

Advanced Data Management



Federico Segala

Anno Accademico: 2025-2026

Appunti del corso di Advanced Data Management

prof. Claudio Silvestri

Sommario

1.	Introduzione	4
1.1.	Proprietà di una Base di Dati	4
1.2.	Componenti di un Database	5
1.3.	Database Management System Relazionali	7
1.3.1.	Progettazione di una Base di Dati	9
1.3.2.	Query Relazionali	10
1.3.3.	Transazioni e Gestione della Concorrenza	11
1.3.4.	Problematiche del Modello Relazionale	12
1.4.	Nuovi Requisiti	12
2.	Key-Value Stores e Document Databases	14
2.1.	Key-Value Stores	14
2.1.1.	Map-Reduce	14
2.2.	Document Database	15
2.2.1.	JavaScript Object Notation	16
2.2.2.	MongoDB	17
2.2.3.	Caso d'uso: Location-Based Application	21
2.2.4.	Operazioni di Aggregazione in MongoDB	26
2.2.5.	Todo Maybe: forse torneremo qui per parlare di sharded deployments	33
3.	Column Stores	34
3.1.	Architettura di un R-DBMS	34
3.2.	Column Stores	36
3.2.1.	Vantaggi e Svantaggi dei Column Stores	37
3.2.2.	Compressione delle colonne	37
3.2.3.	Alcuni Prodotti Commerciali	42
4.	Extensible Record Stores	43
4.1.	Modello Logico dei Dati	43
4.2.	Storage Fisico dei Dati	44
4.2.1.	Flusso delle operazioni di scrittura	44
4.2.2.	Modifica e Cancellazione dei Dati	45
4.2.3.	Flusso di Lettura dei Dati	46
4.2.4.	Ulteriori Accorgimenti	47
4.3.	Alcune Implementazioni	51
5.	Graph Databases	52
5.1.	Teoria dei Grafi	52
5.1.1.	Navigazione in un Grafo	53
5.1.2.	Problemi sui Grafi	54
5.2.	Basi di Dati a Grafo	54
5.2.1.	Modello dei Dati	55
5.2.2.	Memorizzazione di un Grafo	57
5.2.3.	Graph Database Management Systems & API	62
5.3.	Neo4j	64
5.3.1.	Property Graph Model	64
5.3.2.	Graph Querying: Qualche Esempio	64

5.3.3. Introduzione a Cypher	66
5.3.4. Indici e vincoli	68
5.3.5. Scrittura di Query in Cypher: Utilizzi Avanzati	69
5.3.6. Data Ingestion in Neo4j	71
6. Gestione di Dati Distribuiti	73
6.1. Concetti di Base	73
6.2. Guasti nei Sistemi Distribuiti	74
6.3. Protocolli Epidemici	75
6.3.1. Background	75
6.3.2. Varianti di Protocolli Epidemici	76
6.3.3. Hash Trees	76
6.4. Frammentazione («Sharding»)	77
6.4.1. Allocazione	78
6.5. Replicazione dei Dati	81
6.5.1. Replicazione Master-Slave	81
6.5.2. Replicazione Multi-record	82
6.5.3. Replicazione Multi-Master	82
6.5.4. Guasti e Recovery	83
6.6. Gestione della Concorrenza	84
6.7. Proprietà dei Database Distribuiti	85
6.7.1. Congettura di Brewer	86
7. Architetture per Basi di Dati	87
7.1. Persistent Memory Manager	88
7.1.1. Geometria di un Hard Disk	88
7.1.2. Lettura da Disco Magnetico	89
7.2. Buffer Manager	90
7.2.1. Bozza di Interfaccia del Buffer Manager	90
7.2.2. Buffer Pool	91
7.2.3. Tabella delle Pagine Residenti	92
7.2.4. Struttura di una Pagina di Memoria	93
7.3. Strutture per File Management	95
7.3.1. Modello Seriale e Sequenziale	96
7.3.2. Algoritmi di Ricerca e Costi	99
7.4. Problemi di Ordinamento	101

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-strutturetti**, altre forme tipiche sono strutture ad **albero** (XML) o a **grafo**, nel peggio 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 necessitano 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 cui 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 illustra come molti altri componenti nel sistema operativo vadano a interagire tra di loro e con questo importante elemento.

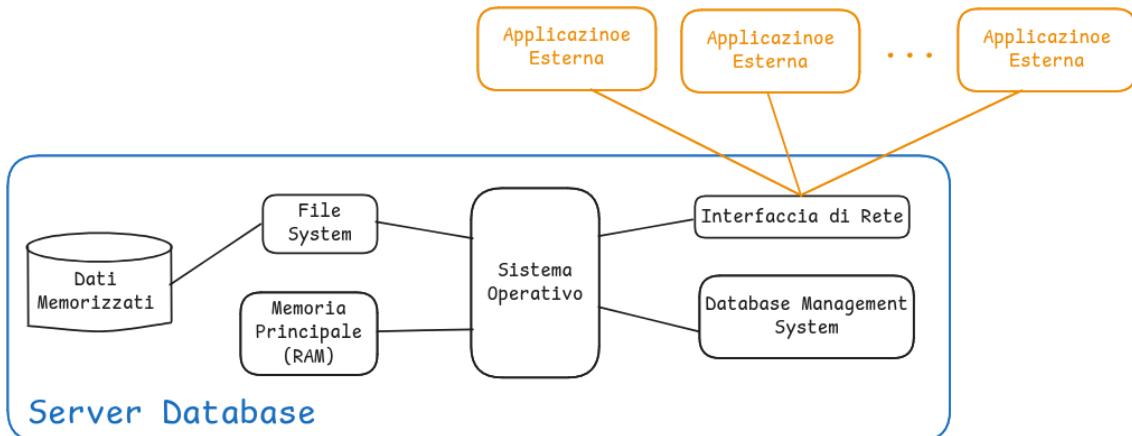


Figura 1: Interazioni del DBMS

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:

- | | |
|--|--|
| <ul style="list-style-type: none"> • T_1, T_2, T_3 • T_2, T_1, T_3 • T_3, T_2, T_1 | <ul style="list-style-type: none"> • T_1, T_3, T_2 • T_2, T_3, T_1 • T_3, T_1, T_2 |
|--|--|

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 varianti 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 tavole* 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, \dots, A_n
- Insieme delle dipendenze locali Σ_R

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

$$R = (\{A_1, \dots, A_n\}, \Sigma_R) \quad (1)$$

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, \dots, R_m
- Insieme di dipendenze globali Σ

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

$$D = (\{R_1, \dots, R_m\}, \Sigma) \quad (2)$$



Esempio: Schema di una Base di Dati

- Schema relazionale 1:

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

- Schema relazionale 2:

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

- 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*: $\text{BookID} \rightarrow \text{BookID}$, $\text{Author}, \text{Title}$ 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 \leq Book.BookID o ancora BookLending.ReaderID \leq 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.

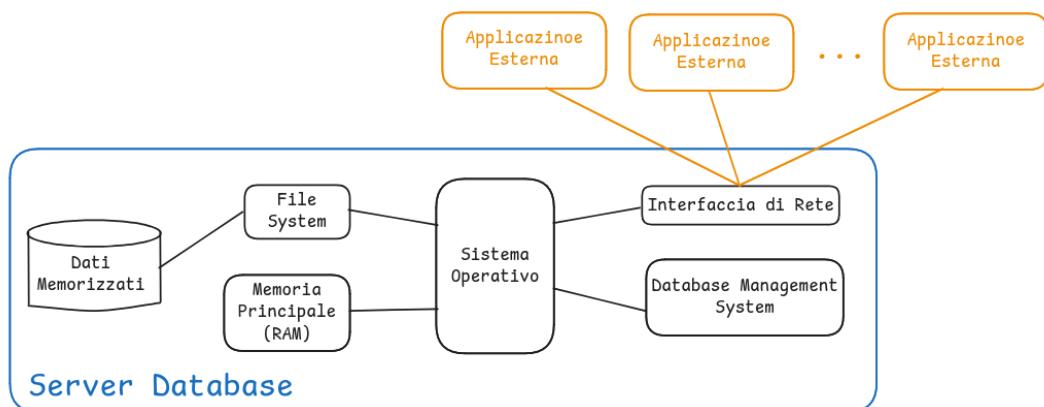


Figura 2: Diagramma Entità-Relazione di una libreria

- *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 operatori 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 $\text{ReturnDate} < 20/10/2016$

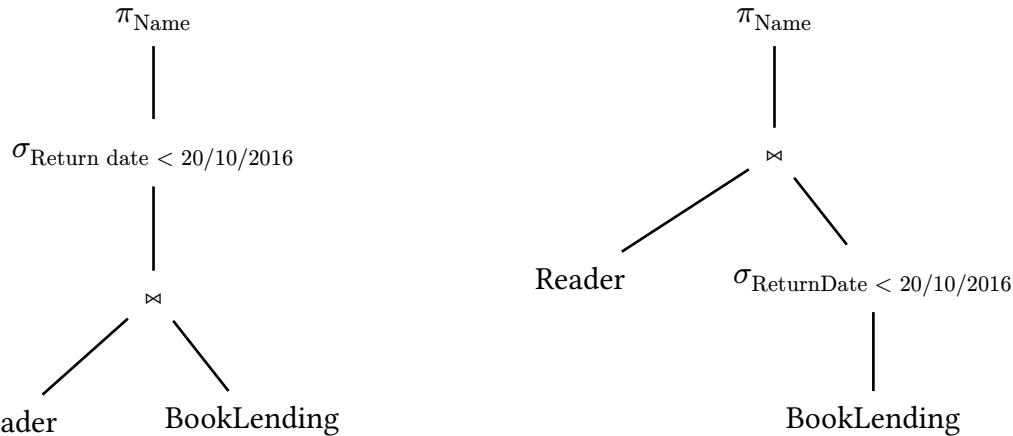


Figura 3: Due diversi piani di query per la stessa richiesta in algebra relazionale.

Possiamo osservare come Figura 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 gestiscono 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 odiere 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 4 illustra chiaramente il funzionamento dei vari passaggi necessari al funzionamento dell'algoritmo MapReduce.

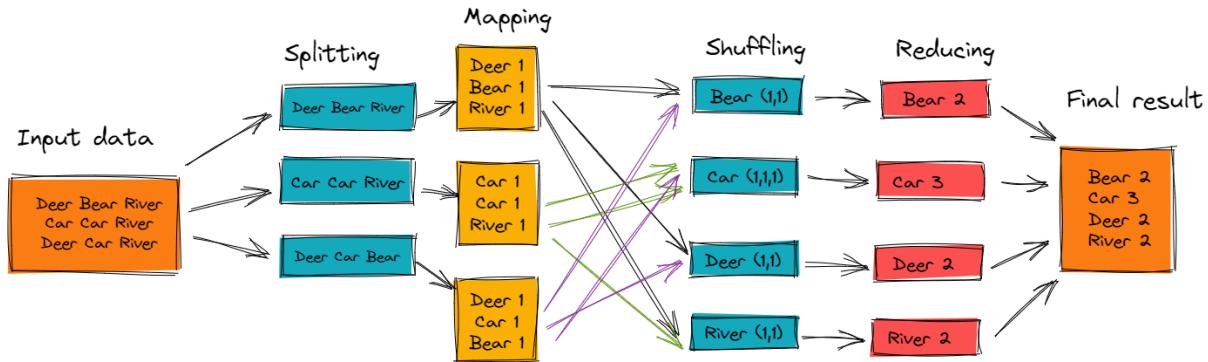


Figura 4: Esempio di applicazione dell'algoritmo MapReduce per contare le occorrenze di ogni parola all'interno di un input

È possibile notare i seguenti aspetti:

- L'approccio è estremamente 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 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 sceglieremo di utilizzare

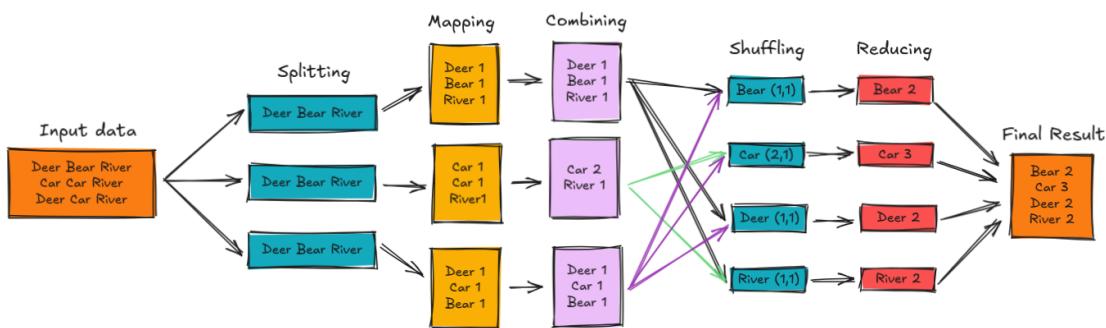


Figura 5: Applicazione della funzionalità di *combine* al metodo MapReduce

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

```
1 {  
2   "firstName": "Alice",  
3   "lastName" : "Smith",  
4   "age"       : 31,  
5 }
```

JSON

Esempio: Oggetto complesso con figli composti e array di valori

```
1 {  
2   "firstName" : "Alice",  
3   "lastName"  : "Smith",  
4   "age"        : 31,  
5   "address"   : {  
6     "street"    : "Main Street",  
7     "number"    : 12,  
8     "city"      : "Newtown",  
9     "zip"       : 31141,  
10    },  
11   "telephone" : [123456, 908077, 2782783],  
12 }
```

JSON

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 oggetto 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 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.

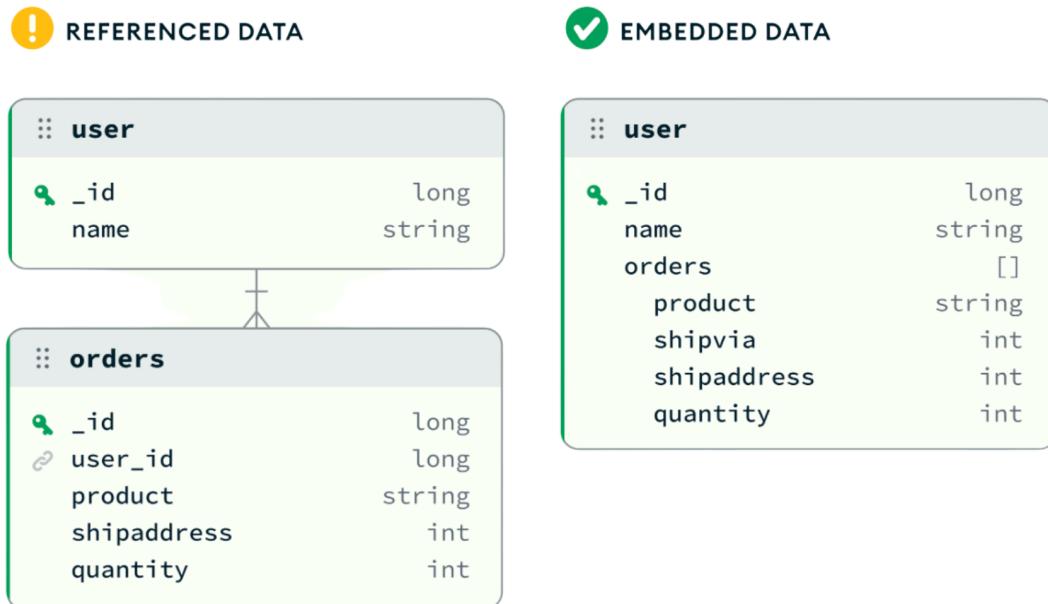


Figura 6: Esempio di de-normalizzazione: a sinistra una struttura normalizzata basata su due entità distinte, a destra una rappresentazione de-normalizzata della stessa relazione.

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.

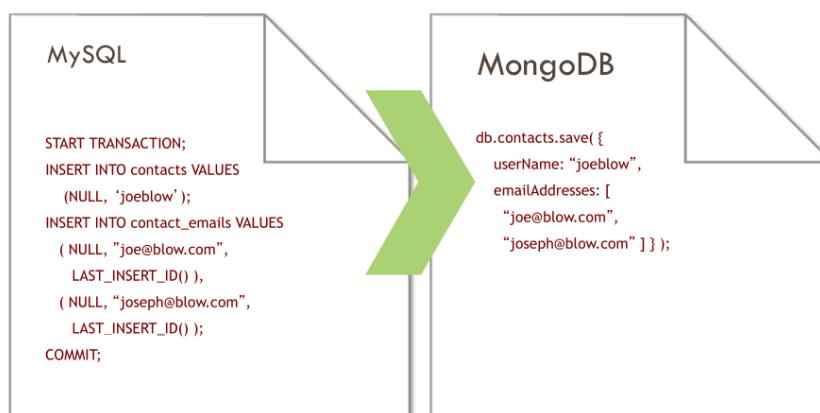


Figura 7: Esempio di Salvataggio atomico

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 aggioran-
re 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

```
1 > db.user.insert({  
2   firstName : "John",  
3   lastName  : "Doe",  
4   age        : 39  
5 })
```

js JavaScript

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

```
1 > db.user.find()  
2  
3 > result : {  
4   "_id"      : ObjectId("51.."), // assegnato automaticamente  
5   "firstName": "John",
```

js JavaScript

```

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(
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({
2   "firstName": /^J/      // regexp per identificare tutti i documenti
3   dove firstName inizia per "J"
4 })
```

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 devono 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

```
1 location = {  
2   name: "10gen East Coast",  
3   address: "134 5th Avenue 3rd Floor",  
4   city: "New York",  
5   zip: "10011",  
6  
7   tags: ["business", "offices"]  
8 }
```

JSON

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

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

JavaScript

```

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

```

1 location = {
2   name: "10gen East Coast",
3   address: "134 5th Avenue 3rd Floor",
4   city: "New York",
5   zip: "10011",
6
7   tags: ["business", "offices"],
8   latlong: [40.0, 72.0]
9 }

```

 JSON

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**:

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

 JavaScript

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**:

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

 JavaScript

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

```
1  location = {  
2      name: "10gen East Coast",  
3      address: "134 5th Avenue 3rd Floor",  
4      city: "New York",  
5      zip: "10011",  
6  
7      tags: ["business", "offices"],  
8      latlong: [40.0, 72.0],  
9      tips: [ // lista di oggetti complessi  
10         {  
11             user: "nosh", date: "6/26/2010",  
12             tip: "stop by for office hours on Thursdays",  
13         },  
14         {...},  
15     ]  
16 }
```

JSON

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

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

JavaScript

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:

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

JavaScript

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**:

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

JavaScript

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:

```

1 db.locations.update(
2   {name: "10gen HQ"}, // query
3   // push è usato per aggiungere elementi ad una lista (tips)
4   {$push: {tips:
5     {
6       user: "nosh", date: "2/26/2010",
7       tip: "stop by for office hours on Thursdays",
8     }
9   }}
10 )
11 )

```

js JavaScript

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 analogo quello per i *tips* nella collection delle *location*.

```

1 user = {
2   name: "nosh",
3   email: "nosh@10gen.com",
4   ... // altre proprietà dell'utente
5   checkins: [
6     {
7       location: "10gen HQ",
8       timestamp: "9/20/2010, 10:12:00",

```

JSON

```

9      ... // altre proprietà
10     },
11     ... // altri check-in dell'utente
12   ]
13 }
```

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

```

1 // estrazione di tutti gli utenti che hanno effettuato un
check-in in una location
2 db.users.find({"checkins.location": "10gen HQ"})
3
4 // estrae i 10 utenti che hanno effettuato più check-in in una location
5 db.users.find({"checkins.location": "10gen HQ"}).sort({ts:-1}).limit(10)
6
7 // estrae quanti utenti hanno effettuato un check-in in una location dopo
un certo timestamp
8 db.users.find({
9   "checkins.location": "10gen HQ",
10  timestamp: {$gt: ...$}}
11 ).count()
```

js JavaScript

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`

```

1 user = {
2   name: "nosh",
3   email: "nosh@10gen.com",
4   ... // altre proprietà
5   checkins = [e4af242f, cfeb950a, a542e63e]
6 }
```

JSON

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

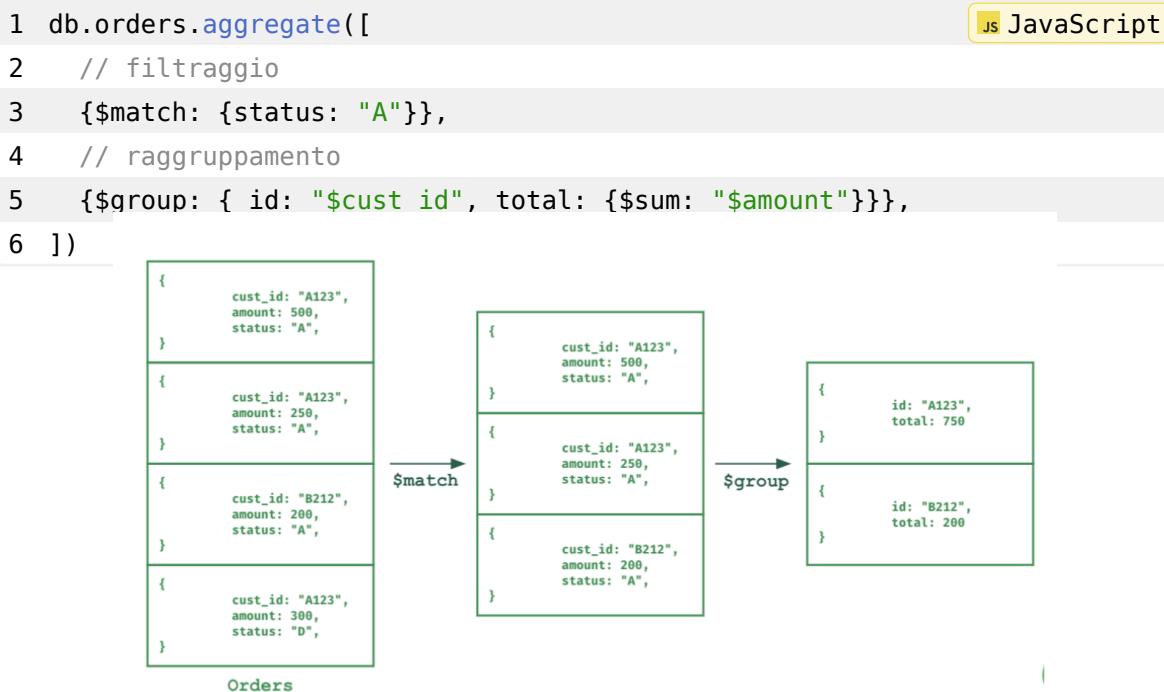


Figura 8: Rappresentazione grafica della pipeline di aggregazione specificata nel codice sopra

Nell'esempio di Figura 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 operazioni 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'

```
1 [  
2   { item: "apple", qty: 10, store: "A", region: "north"},  
3   { item: "apple", qty: 5, store: "B", region: "south"},  
4   { item: "banana", qty: 7, store: "A", region: "north"},  
5   { item: "banana", qty: 8, store: "B", region: "south"},  
6   { item: "banana", qty: 3, store: "C", region: "north"}  
7 ]
```

JSON

Possiamo applicare la seguente pipeline di aggregazione:

```
1 db.sales.aggregate([  
2   // filtriamo i vari documenti  
3   { $match: { region: "north" }, // filtra per regione 'north'  
4     qty: { $gte: 5},           // e quantità maggiore o uguale a 5  
5   },  
6   {  
7     $group: {  
8       _id: "$item", // raggruppa per item  
9       totalQty: { $sum: "$qty" }, // somma le quantità per ogni item  
10      stores: { $push: "$store" }, // crea una lista di store per  
11        ogni item  
12    }  
13  }  
14 ])
```

JavaScript

Il risultato di questa pipeline sarà il seguente:

```

1 [
2   { "_id": "apple", "totalQty": 10, "stores": ["A"] },
3   { "_id": "banana", "totalQty": 10, "stores": ["A", "C"] }
4 ]

```

JSON

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

Esempio: Aggregazione 'by window'

```

1 db.cakeSales.aggregate([
2   $setWindowFields: {    // inizio dell'operazione di windowing
3     partitionBy: "$state", // partiziona per stato
4     sortBy: {orderDate: 1}, // ordina per data
5     output: { // definizione del campo di output
6       cumulativeQuantityForState: {
7         $sum: "$quantity",
8         window: {
9           documents: ["unbounded", "current"] // considera documenti
10          dal primo a quello corrente (somma cumulativa)
11        }
12      }
13    }
14  })

```

JavaScript

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

- `$setWindowFields` serve a permettere di calcolare funzioni *windowed* su ogni documento; permettendo di aggiungere o sostituire campi basati sui valori in una determinata finestra
- `$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

```
1  {
2    $bucket: {
3      groupBy: <expression>, // espressione su cui basare il
4      raggruppamento
5      boundaries: [ <lowerbound1>, <lowerbound2>, ... ], // definizione
6      dei confini dei bucket
7      default: <literal>, // bucket di default per valori fuori dai
8      confini
9      output: { // definizione dei campi di output per ogni bucket
10        <field1>: { <accumulator1> : <expression1> },
11        ...
12        <fieldN>: { <accumulatorN> : <expressionN> }
13      }
14    }
15 }
```

js JavaScript

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

```
1 db.things.aggregate([
2   { $group: { _id: $parity$, sum: {$sum: $value} } }
3 ])
4
5 > { "result" : [
6   { "_id": "even", "sum": 102 },
7   { "_id": "odd", "sum": 97 }
8 ]
9 }
```

js JavaScript

Esempio: Aggregazione 2

```
1 db.zipcodes.aggregate([
2   { $group: {
3     _id: "$state",
4     totalPop: { $sum: "$pop" },
5   }},
6   {
7     $match: { totalPop: { $gte: 10^6 } }
8   }
9 ])
```

js JavaScript

Esempio: Aggregazione 3

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

```
1 db.zipcodes.aggregate([
2   { $group: { _id: {"$state", city: "$city"}, pop: {$sum: "$pop"}}, },
3   { $sort: {pop: 1}},
4   { $group: {
5     _id: "$_id.state",
6     biggestCity: {$last: "$_id.city"},
7     biggestPop: {$last: "$pop"},
8     smallestCity: {$first: "$_id.city"},
9     smallestPop: {$first: "$pop"}
10   }},
11   { $project: {
12     _id: 0, // non mostrare il campo _id
13     state: "$_id",
14     biggestCity: {name: "$biggestCity", population: "$biggestPop"},
15     smallestCity: {name: "$smallestCity", population: "$smallestPop"},
16   }}
17 ])
```

js JavaScript

Esempio: Aggregazione 4

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

```
1 db.places.aggregate([
2   {
3     $geoNear: {
4       near: {type: "Point", coordinates: [ -73.9667, 40.78 ]},
```

js JavaScript

```

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( js 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 8.

Esempio: Algoritmo MapReduce in MongoDB

```
1 db.orders.mapReduce( js 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 9.

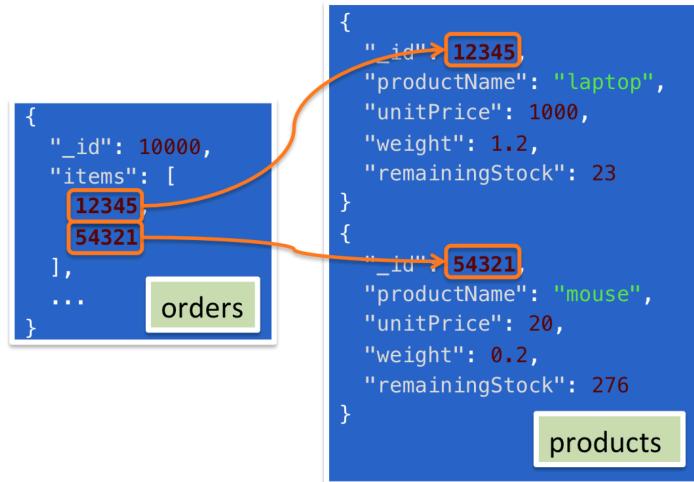


Figura 9: Esempio di utilizzo dell'operatore `$lookup` per effettuare un left-outer join

Di seguito andiamo a mostrare la sintassi del comando

```
1 {  
2   $lookup: {  
3     from: <collection to join>,  
4     localField: <field from the input documents>,  
5     foreignField: <field from the documents of the "from"  
6     collection>,  
7     as: <output array field>  
8   }  
9 }
```

JS JavaScript

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-dокументo**
- **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 parlare 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 10 è una rappresentazione dell'organizzazione interna delle componenti di un DBMS relazionale.

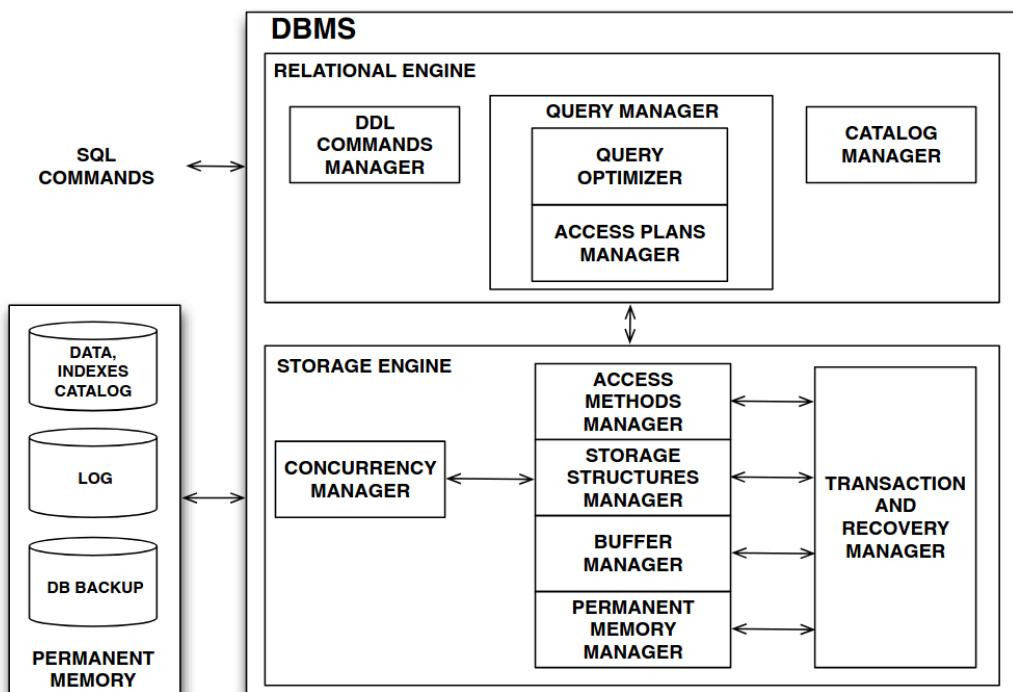


Figura 10: Organizzazione interna delle componenti di un R-DBMS

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 trasferire 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.

Supponiamo 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 a 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 hanno 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 11 mostra un esempio del funzionamento di questo algoritmo.

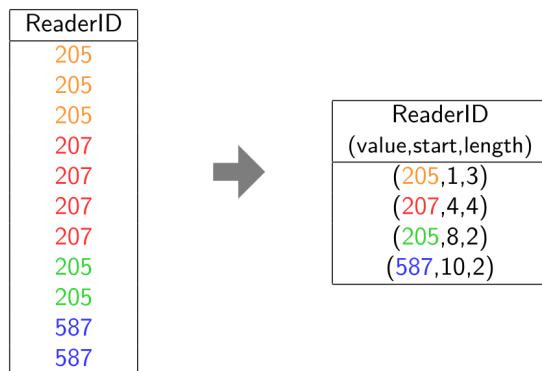


Figura 11: Esempio di Run-Length Encoding

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 12 mostra un esempio del funzionamento di questo algoritmo.

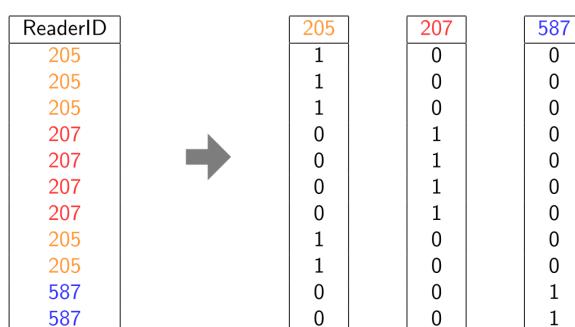


Figura 12: Esempio di Bit-vector encoding

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 13.

ReaderID
205
205
205
207
207
207
207
205
205
587
587

ReaderID
1
1
1
2
2
2
2
1
1
3
3

Figura 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 14

BookID
1002
1010
1004
1008
1006
1002
1010
1004

BookID
1
2
1

Figura 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 15 mostra un esempio del funzionamento di questo algoritmo.

The diagram illustrates the Frame of Reference Encoding process. On the left, a vertical column of dates is shown in a table:

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

An arrow points to the right, indicating the transformation into a compressed representation:

ReturnDate
Reference: 25-10-2016
-3
+2
0
-3
+3
14-10-2016
+1
-4
0

Figura 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 16 mostra un esempio del funzionamento di questo algoritmo.

The diagram illustrates the differential encoding process. On the left, a vertical column of dates is shown in a table:

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

An arrow points to the right, indicating the transformation into a compressed representation:

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

Figura 16: Esempio di encoding differenziale con valori oltre l'offset massimo

Osservazione

Si noti come in Figura 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 colonna. 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:

```
1 SELECT ReaderID, COUNT(*)  
2 FROM BookLending  
3 GROUP BY ReaderID
```

SQL

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. Alcuni Prodotti Commerciali

Di seguito sono elencati alcuni dei più noti prodotti commerciali che implementano uno storage engine colonna:

- **Monet DB**: si tratta del primo database basato su storage engine colonna, supporta il modello relazionale.
- **Maria DB ColumnStore**: si tratta di un'estensione di MariaDB che implementa uno storage engine colonna, supporta il modello relazionale.
- **Apache Parquet**: si tratta di un formato di file che implementa uno storage engine colonna, utilizzato principalmente in ambito big data.

4. Extensible Record Stores

Nello scorso capitolo abbiamo visto come un cambio di paradigma dello **storage engine** possa portare a dei grandi benefici in termini di prestazioni sfruttando la *località dei dati*. Vogliamo provare a spingere questo concetto ancora oltre, andando a considerare un particolare tipo di database NoSQL chiamato **extensible record store** (ERS).

Ipotizziamo di dover gestire una base di dati che memorizzi informazioni riguardanti delle persone, ogni persona avrà delle informazioni che sono particolarmente importanti, come ad esempio la data e il luogo di nascita, la città in cui questa ha residenza e così via. Probabilmente tante di queste informazioni saranno accedute “assieme” (es. difficilmente avremo bisogno di sapere la data di nascita senza sapere anche il luogo di nascita).

4.1. Modello Logico dei Dati

In questa sezione andremo a vedere come è possibile modellare i dati in un sistema di questo tipo. È importante andare a vedere le seguenti regole, che si pongono alla base del funzionamento degli extensible record store:

- Le colonne sono raggruppate in insiemi detti **column families**, le quali hanno lo scopo di raggruppare le colonne che sono spesso accedute assieme
- Ogni column family deve essere creata prima che possa essere utilizzata (similmente a quanto avviene con una tabella SQL)
- All'interno di una column family è possibile aggiungere nuove colonne in maniera arbitraria, senza dover modificare uno schema predefinito
- Ogni riga di una column family può avere un insieme di colonne diverso dalle altre righe della stessa column family

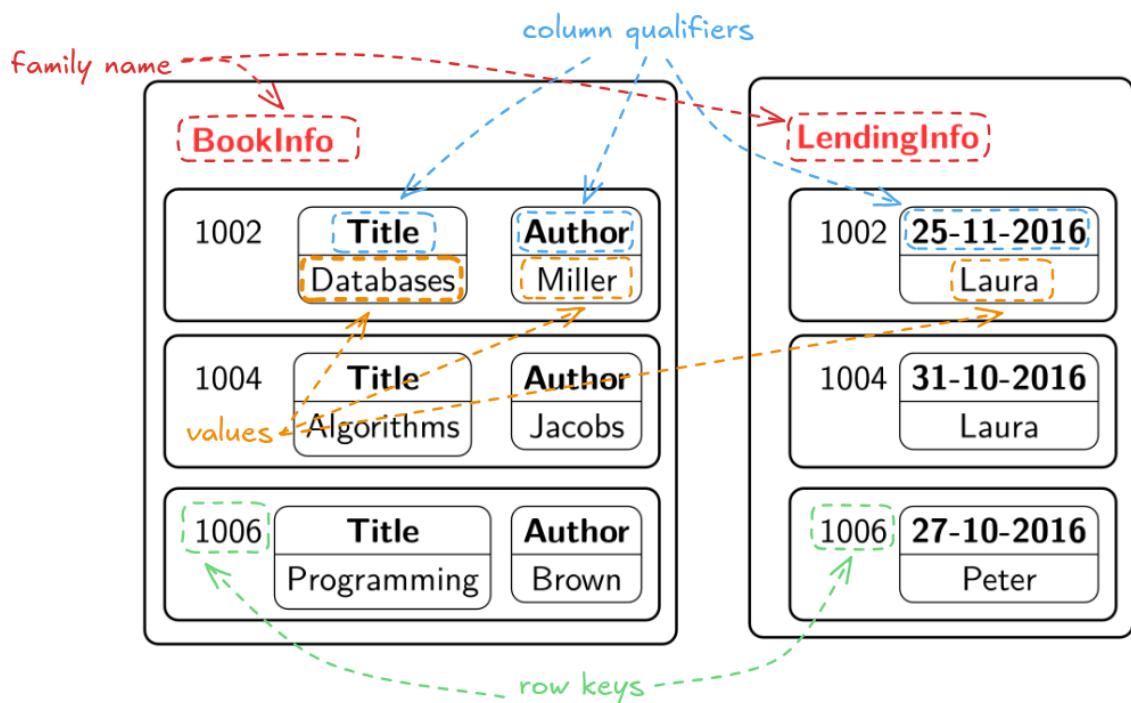


Figura 17: Terminologia degli Extensible Record Stores

Figura 17 mostra con chiarezza i vari elementi che troviamo all'interno un extensible record store: andiamo a definirli brevemente:

- Una **column family** è un insieme di colonne che sono spesso accedute assieme
- Una **row key** è l'identificativo univoco di una riga all'interno di una column family
- Un **column qualifier** è il nome di una colonna all'interno di una column family

Osservazione

Tipicamente i *column qualifiers* sono dei metadati e sono **costanti**, tutta via in Figura 17 è stato scelto di utilizzare una data come qualificatore per una colonna. Questo è possibile dal momento le colonne sono arbitrarie e possono avere valori diversi per ogni riga (quindi nulli nel caso di dati non pertinenti). Il motivo per cui questo è stato fatto è che se possiamo accedere ai dati tramite nome di una colonna, è possibile accedere a tutti i prestiti di un giorno in maniera immediata.

Per quanto riguarda la memorizzazione, fino a questo momento ci basta sapere che data una column family, ogni row viene memorizzata in maniera consecutiva a quella precedente, questo consente di ottenere **località spaziale**.

Accesso ai Valori di una Colonna

L'accesso a un valore di una colonna è possibile tramite l'utilizzo della sua **full key**:

```
<row key>:<column family>:<column qualifier>
```

Per esempio, con riferimento all'immagine in Figura 17, la full key `1006.BookInfo.Author` identifica in maniera univoca il valore `Brown`.

È inoltre possibile andare ad aggiungere una dimensione ulteriore, quella **temporale**, aggiungendo alla full key un *timestamp*, questo garantisce che sia possibile avere diverse versioni dello stesso dato, ad esempio per tenere traccia delle modifiche avvenute nel tempo.

4.2. Storage Fisico dei Dati

In questa sezione andremo a vedere più nel dettaglio, dato il modello logico precedentemente descritto, come i dati vengano effettivamente memorizzati sul disco. Il principio fondamentale di questa tipologia di basi di dati è quello di consentire alta velocità in scrittura, anche al costo di avere delle letture un po' più lente.

4.2.1. Flusso delle operazioni di scrittura

Tipicamente i dati più recenti vengono scritti all'interno di una struttura chiamata **memtable** che risiede in memoria principale. Tipicamente viene mantenuta una memtable per ogni column family. Ogni record nella memtable viene identificato tramite la sua full key (`row + column family + qualifier + timestamp[ms]`).

Una volta che una memtable è completa, avviene un'operazione di **flushing**, che consiste nel salvare i dati in memoria su disco. Potremmo tranquillamente scrivere i dati in un unico file, mettendo i dati nuovi in coda a quelli inseriti per ultimi, ma questo porterebbe a dover scorrere tutto il file per trovare un dato specifico, risultando in un costo estremamente elevato in

lettura. Questo costo potrebbe essere notevolmente abbassato memorizzando i dati in maniera ordinata, ma questo, utilizzando un singolo file, comporterebbe il dover ogni volta cercare la posizione di ogni dato da scrivere e il dover spostare i dati in avanti per fare spazio a quelli nuovi, risultando in *scritture lente*.

Per mitigare il costo in lettura, viene scelto di effettuare un'operazione di **ordinamento** su ogni memtable e di memorizzare la tabella ordinata in un singolo file in memoria. Questo è il motivo per cui sostanzialmente ha senso avere una memtable per ogni column family: il fatto che l'ordinamento possa essere effettuato sulla singola column family. In Figura 18 è mostrato il flusso delle operazioni di scrittura in un extensible record store.

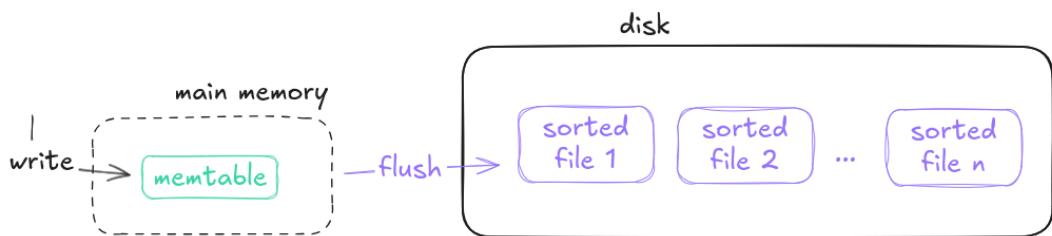


Figura 18: Flusso delle operazioni di scrittura in un Extensible Record Store

Osservazione

Si noti come una struttura di questo genere possa essere estesa in maniera estremamente efficace ad un architettura distribuita e ad un approccio parallelizzabile, rendendo di fatto la ricerca ancora più efficiente, lasciando ogni server a lavorare sul suo o i suoi file ordinati.

4.2.2. Modifica e Cancellazione dei Dati

Per garantire che le scritture siano il più veloci possibili, è chiaro che anche le operazioni di modifica e/o cancellazione debbano essere diverse. Sarebbe impensabile andare a cercare il record da modificare in memoria principale e cambiarlo, dal momento che questo potrebbe essere scritto in qualsiasi posto del disco.

Per questo motivo viene imposto il vincolo che i dati memorizzati siano **immutabili** e che le modifiche di un record possano soltanto essere “simulate” andando a scrivere un nuovo record con la stessa full key e un timestamp più recente. In questo modo, quando si andrà a leggere un record, si prenderà sempre quello con il timestamp più recente, che sarà quello valido.

Osservazione

Questa modalità di effettuare l'aggiornamento dei dati, permette di avere senza costi aggiuntivi l'implementazione di una sorta di **versioning** dei record.

Se per andare a **modificare** dei dati basta semplicemente scrivere un nuovo record con un nuovo timestamp, per effettuare la **cancellazione** è possibile utilizzare un meccanismo simile, andando ad aggiungere un nuovo record con la stessa full key e un timestamp più recente, ma con un particolare valore chiamato **tombstone** che indica che quel record è stato cancellato. Di seguito andiamo ad elencare le varie caratteristiche di questi valori:

- I tombstone vanno a *mascherare* tutti i record precedenti con la stessa full key e timestamp precedente a quello del tombstone
- Esistono diversi tipi di tombstone, a seconda del tipo di dato che si vuole cancellare:
 - si può cancellare una singola **versione** di una colonna, andando a scrivere il **timestamp** relativo alla versione che si vuole cancellare
 - è possibile cancellare l'intera **colonna** con tutte le versioni
 - è possibile andare a cancellare un'intera **column family** andando di fatto a rimuovere tutte le versioni di tutte le colonne
- Nuove versioni per una stessa chiave possono essere scritte anche dopo la cancellazione, in questo caso il nuovo record non sarà mascherato dal tombstone

4.2.3. Flusso di Lettura dei Dati

Come già anticipato, le operazioni di lettura in un sistema di questo genere sono più lente e complesse rispetto a quelle di scrittura. Questo perché i dati sono memorizzati in maniera *sparsa* su disco, e per trovare un dato specifico potrebbe essere necessario andare a leggere più file.

In particolare, per leggere un dato specifico si rende necessario **combinare** i dati che provengono sia dalla memtable (dati più recenti) sia dai file memorizzati su disco. La versione corretta dei dati da recuperare è quella più recente, informazione che sarà accessibile tramite timestamp. Figura 19 mostra chiaramente questo processo.

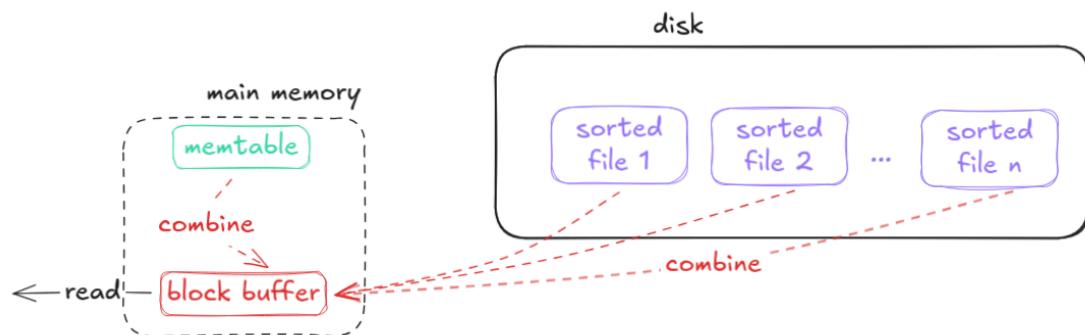


Figura 19: Flusso delle operazioni di lettura in un Extensible Record Store

Formato dei File

Andiamo ora a concentrare l'attenzione su come i dati vengono rappresentati all'interno dei file ordinati in memoria secondaria. Iniziamo a vedere come è strutturato un **file** (struttura ordinata proveniente dal flush della memtable):

- Ogni file si può vedere come composto da molti **blocchi** (data blocks)
- Dal momento che all'interno di un file sono presenti molti blocchi viene tenuto un **indice** che consente di trovare rapidamente il blocco che contiene il dato cercato
- Insieme all'indice viene mantenuto un **trailer** che consente di memorizzare informazioni per la gestione del file, come ad esempio la posizione dell'indice stesso

Osservazione

Il motivo per cui l'indice viene inserito in fondo al file, è legato al fatto che questa scelta consente una scrittura su disco più veloce: possiamo costruire l'indice man mano che scriviamo i record uno dopo l'altro, invece di dover prima leggere tutti i record da scrivere per poter inserire l'indice in testa.

Ogni **data block** è sostanzialmente composto da una lista di **key-value** pairs. Ogni key è la full key del record (row key + column family + column qualifier + timestamp) e il value è il valore associato a quella specifica chiave. Andando più nel dettaglio, queste key-value pairs sono memorizzate assieme ad un campo **type** che ci serve a distinguere se il record è un record normale *inserimento*, una *modifica* o una *cancellazione* (tombstone). Figura 20 mostra la gerarchia appena descritta.

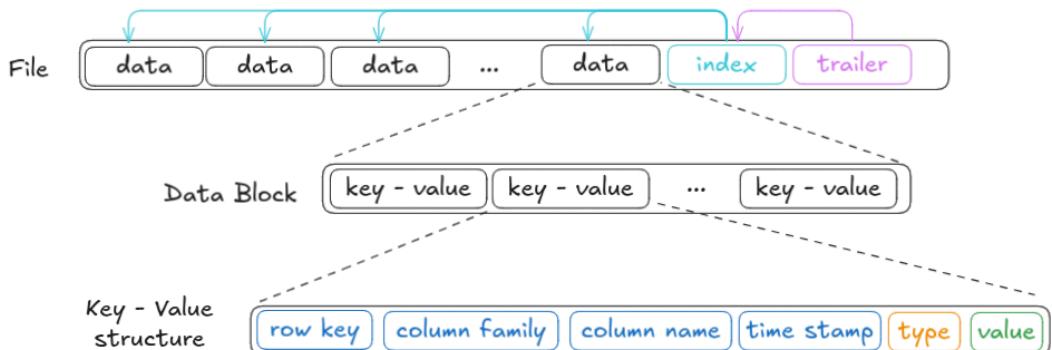


Figura 20: Formato di un file in un Extensible Record Store

4.2.4. Ulteriori Accorgimenti

Write-Ahead Logging

Il sistema progettato fino a questo momento è estremamente vulnerabile nel caso di **crash improvvisi**: tutto ciò che è presente in memtable non sarebbe infatti recuperabile. Per garantire che i dati non vadano persi in questi casi, viene utilizzata una tecnica chiamata **write-ahead logging**. Sostanzialmente ogni operazione di scrittura viene effettuata due volte: una volta su un **file di log** memorizzato su disco, e una seconda volta nella **memtable**.

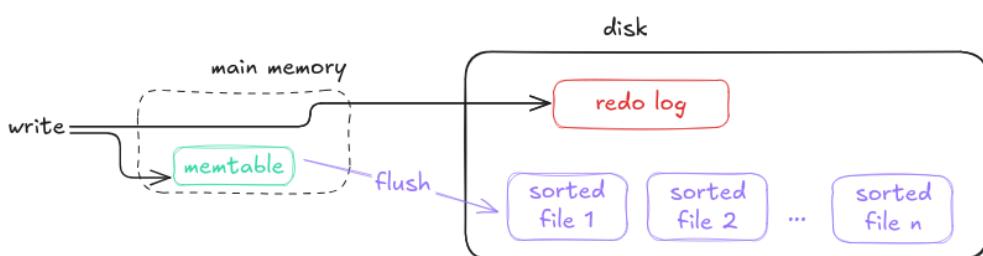


Figura 21: Write-Ahead Logging in un Extensible Record Store

Utilizzando questo approccio, in caso di system crash, è possibile recuperare tutte le operazioni di scrittura effettuate leggendo il file di log e reinserendole nella memtable. Figura 21

mostra questo processo. In pratica, il **redo log** viene svuotato ogni qualvolta che avviene un'operazione di flushing della memtable su disco. Se al verificarsi di un crash di sistema il log contiene dei record, significa che è necessario ripristinare lo stato.

Compattazione dei File

Dopo che si sono verificate **molte operazioni di flushing**, si verranno a creare **molti file** ordinati su disco. Questo comporta che le *operazioni di lettura* diventino *sempre più lente*. Questo perché per leggere un dato specifico, potrebbe essere necessario andare a leggere molti file diversi, andando così a vanificare il vantaggio di avere i dati ordinati all'interno di ogni singolo file.

Per capire l'idea alla base di questo approccio, è necessario considerare i seguenti punti:

- Nel momento in cui cerchiamo una chiave, dobbiamo cercarla all'interno di ogni file presente all'interno del disco
- Il costo di ricerca della chiave all'interno di un file consiste nella ricerca del trailer (1 accesso a pagina), della ricerca dell'indice (1 accesso a pagina) e nel caso in cui la chiave sia presente nel file, il costo di ricerca all'interno del blocco dati (1 accesso a pagina), per un totale di massimo 3 pagine per file
- Possiamo concludere che il costo della ricerca di una chiave sia dominato dal numero di file presenti su disco $O(n)$, dove n è il numero di file (o di operazioni di flushing effettuate fino ad un certo punto)

L'idea potrebbe essere quella di andare a **ridurre il numero di file**, per fare ciò, senza perdita di informazione, la soluzione consiste nel **combinare** più file all'interno di un unico file più grande. La creazione di file più grandi nativi non è supportata, dal momento che la dimensione di questi dipende dalla dimensione della *memtable*. Dal momento che effettuare ordinamento su disco è particolarmente costoso, questa operazione viene effettuata soltanto l'effort necessario è compensato dall'aumento di velocità in lettura.

Durante l'operazione di **compaction** è possibile anche decidere di eliminare i record che sono stati marcati come cancellati tramite tombstone, così da liberare spazio su disco.

Osservazione

Possiamo associare questa operazione di compattazione a quella di **deframmentazione di un disco** o a quella di **ricostruzione dell'indice** di una base di dati relazionale.

Osservazione

Dal momento che i file che stiamo riunendo in un unico file ordinato sono già rispettivamente ordinati, possiamo applicare un comunissimo algoritmo di ordinamento: il **merging**, che corrisponde alla seconda fase dell'algoritmo *merge-sort*.

Figura 22 mostra il flusso di operazioni che avvengono durante una compattazione di file in un extensible record store.

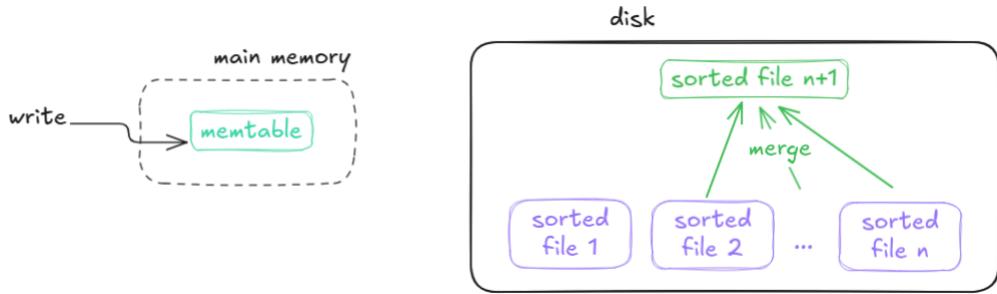


Figura 22: Compattazione dei file in un Extensible Record Store

Bloom Filters

Per migliorare ulteriormente le prestazioni in lettura, è possibile applicare una nuova tecnica chiamata **bloom filter**. Si tratta di un *meccanismo probabilistico* utilizzato per determinare l'*appartenenza ad un insieme*. In questo specifico caso viene utilizzato per determinare se una **chiave** è presente o meno all'interno di un **file specifico** o direttamente in un **data block**.

Questo filtro viene posizionato tra il trailer e l'indice all'interno del file contenente i vari data block e viene memorizzata la sua posizione all'interno del trailer in modo che sia di facile accesso. In Figura 23 è mostrata la nuova struttura.



Figura 23: Bloom Filter all'interno dei file in un Extensible Record Store

Il workflow di lettura di un dato specifico con l'utilizzo del bloom filter è il seguente:

- Viene *letto il trailer* per recuperare la posizione dell'indice e del bloom filter
- L'indice indica quali siano le chiavi minima e massima di ogni data block
- Se esiste un range che può contenere la chiave cercata, si legge il data block, cercando la chiave al suo interno
- Il bloom filter viene utilizzato per determinare se la chiave è presente o meno all'interno del blocco che contiene chiavi nel range prestabilito

Ipotizziamo di avere un **oracolo** che ci dica se una chiave è presente o meno all'interno di un data block, possiamo avere i seguenti casi:

- L'oracolo indica che la chiave esiste, ma non è vero, abbiamo un **true positive**, in questo caso andremmo a leggere il blocco dati e non troveremo la chiave cercata, andando a “sprecare” computazione, ma è una situazione accettabile
- L'oracolo indica che la chiave esiste, ed in effetti esiste, siamo nel caso di un **true positive**. In questo caso andremo a leggere il blocco dati e troveremo la chiave cercata
- Il filtro indica che la chiave non esiste, ma in realtà esiste, siamo nel caso di un **false negative**. Se ci fidassimo dell'oracolo, non andremmo a leggere il blocco dati e perderemmo l'informazione, questa situazione è inaccettabile
- Il filtro indica che la chiave non esiste, ed effettivamente non esiste, siamo nel caso di un **true negative**. In questo caso potremmo evitare di leggere il blocco dati, risparmiando tempo di computazione

Alla luce delle considerazioni appena fatte, i bloom filters sono progettati in modo tale da *poder restituire dei falsi positivi*, che sono appunto tollerabili, ma in modo tale da *non restituire mai dei falsi negativi*, che sono non accettabili.

L'idea dietro ad un bloom filter è quella di usare una piccola quantità di memoria per andare a tracciare appartenenza o non appartenenza ad un insieme. Quando andiamo a memorizzare un record, questo ha delle determinate caratteristiche, alcune di queste possono essere condivise da altri record. Alla luce di questo concetto, sappiamo che nel momento in cui cerchiamo un record, con delle date caratteristiche, se un certo insieme ha elementi con queste caratteristiche, potrebbe (**falsi positivi accettabili**) contenere record di interesse, al contrario sicuramente non conterrà il record cercato (**falsi negativi inaccettabili**).

Esempio: Applicazione di Bloom Filtering ad un semplice contesto

Supponiamo di essere in una classe, e voler cercare di stimare se un generico studente è presente al suo interno. Possiamo considerare un array di 26 elementi (uno per lettera dell'alfabeto), quando una persona entra in classe andiamo a memorizzare che uno studente con una certa iniziale è entrato in aula.

Supponiamo che entrino in aula gli studenti “Bob”, e “Alice”, andremo a settare gli elementi corrispondenti alle lettere “A” e “B” nell'array. Ora, se vogliamo sapere se “Charlie” è presente in aula, andremo a controllare l'elemento corrispondente alla lettera “C”. Dal momento che questo elemento non è settato, possiamo concludere che “Charlie” non è presente in aula (true negative).

Osservazione

Possiamo notare come la scelta delle caratteristiche da utilizzare nel bloom filter sia fondamentale per garantire che il filtro non produca troppi falsi positivi. Nel caso di sopra, stiamo inoltre introducendo un **bias**, dal momento che ci sono iniziali che sono molto più comuni di altre.

Alla luce dell'osservazione precedente, desidereremmo che la distribuzione delle caratteristiche fosse il più **uniforme** possibile. Per fare ciò i bloom filter utilizzano **funzioni di hashing** per andare a mappare i valori delle caratteristiche in indici di un array di bit. In questo modo, anche se le caratteristiche non sono uniformemente distribuite, gli indici generati dalle funzioni di hashing lo saranno. Il filtro che progetteremo avrà lo scopo di decidere se un certo valore **non è presente** in un insieme.

Di seguito andiamo ad elencare i passi per la ricerca di un valore all'interno di un bloom filter:

- Viene richiesto al bloom filter se un certo valore c è presente nell'insieme S
- Vengono utilizzate k funzioni di hashing h_1, \dots, h_k
- Ogni funzione di hashing h_i mappa il valore c in maniera casuale in un indice dell'array di bit di dimensione m : $h_i : c \rightarrow [0, m - 1]$, ossia $h_{i(c)} \in \{1, \dots, m\}$

Nel caso in cui due diversi valori c, c' vengano mappati nello stesso valore: ($h_{i(c)} = h_{i(c')}$), si parla di **collisione**. Chiaramente le collisioni sono la causa principale dei falsi positivi.

Di andiamo ad approssimare il numero di falsi positivi che possiamo aspettarci da un bloom filter. Supponiamo di avere a disposizione k *funzioni di hashing*, un array di bit di dimensione m e di voler memorizzare n elementi nel filtro. Il numero di falsi positivi si può approssimare tramite la formula in Equazione 3.

$$\Pr[\text{false positive}] = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-\frac{kn}{m}}\right)^k \quad (3)$$

Figura 24, mostra come si comporta un bloom filter nel caso in cui sia necessario leggere dei dati, sia nel caso di chiave presente sia nel caso di chiave assente.

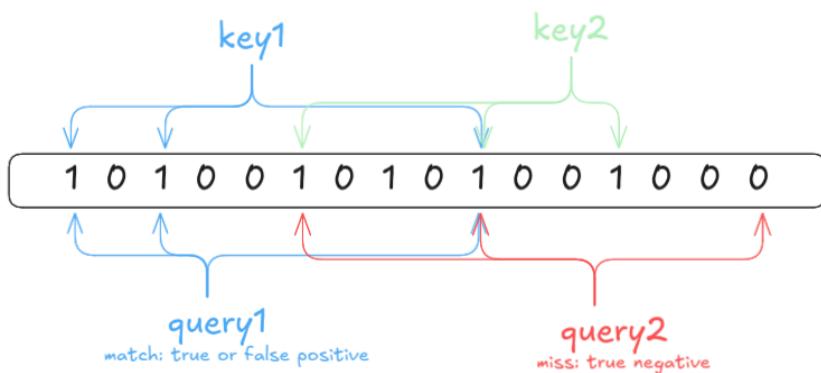


Figura 24: Due esempi di utilizzo di un Bloom Filter durante la lettura in un Extensible Record Store, `query1` mostra il caso in cui la chiave cercata è presente, `query2` mostra il caso in cui la chiave cercata non è presente

4.3. Alcune Implementazioni

Tra le implementazioni più famose di extensible record store troviamo:

- Apache Cassandra, che utilizza CQL, un linguaggio **SQL-like**, nel quale un inserimento avviene nel modo seguente:

```
1 INSERT INTO bookinfo (book_id, title, author)
2 VALUES (1234, 'Foo', 'Bar', '
```

CQL

- HBase, che è costruito sopra a Hadoop HDFS e utilizza il framework MapReduce per l'elaborazione distribuita dei dati

5. Graph Databases

Nel corso di questo capitolo andremo ad affrontare basi di dati organizzate come **grafo**. Nella prima parte andremo a vedere i concetti fondamentali, per poi passare a vedere una delle implementazioni più diffuse: **Neo4j**.

5.1. Teoria dei Grafi

In questa sezione andiamo a porre le basi teoriche per comprendere il concetto al fondamento di questa categoria di base di dati: i **grafi**, di cui andiamo a dare una definizione.

Definizione 5.1 (Grafo)

Un grafo G è una struttura costituita da un insieme di **vertici** e di **archi** $G = (V, E)$, dove:

- ogni *vertice* $v \in V$ rappresenta un'entità o un oggetto;
- ogni *arco* $e \in E$ rappresenta una relazione o connessione tra due vertici.
- Gli archi possono essere **diretti** o indiretti, a seconda che la relazione abbia una direzione specifica o meno.

Un grafo è detto diretto se tutti i suoi archi hanno una direzione associata, al contrario, se nessun arco ha direzione, il grafo si dice indiretto. Figura 25 mostra un esempio di grafo diretto e indiretto.

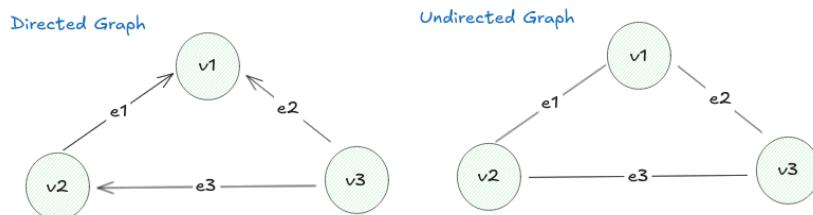


Figura 25: Esempio di grafo diretto (a sinistra) e indiretto (a destra).

Un grafo si dice un **multigrafo** se ha una coppia di nodi che è connessa tramite più di un arco. Questa caratteristica è utile per rappresentare relazioni complesse tra entità, come ad esempio in una rete di trasporti dove più linee possono collegare due stazioni. Figura 26 mostra un esempio di multigrafo.

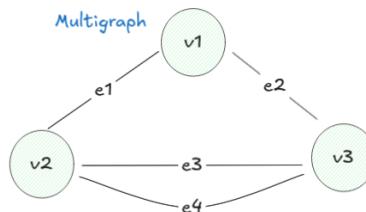


Figura 26: Esempio di multigrafo con più archi tra alcuni nodi. e_3, e_4 sono archi multipli tra i nodi v_2 e v_3 .

Possiamo avere anche degli **ipergrafi**, in cui un arco può connettere più di due vertici. Tuttavia, per i nostri scopi, ci concentreremo su grafi più semplici.

Definizione 5.2 (Adiacenza tra Nodi)

Due vertici v_1, v_2 sono detti **adiacenti**, se sono “vicini”, ovvero se esiste un arco che li connette in maniera diretta.

Definizione 5.3 (Incidenza)

Un arco e è detto **incidente su un vertice** se è connesso a quel vertice. Nel caso di grafi diretti distinguiamo due casi:

- **positivamente** incidente se l’arco parte dal vertice,
- **negativamente** incidente se l’arco arriva al vertice.

Vedremo in seguito come i concetti introdotti da Definizione 5.2° e Definizione 5.3° siano fondamentali per comprendere come andare a memorizzare in maniera efficiente un grafo all’interno della memoria.

5.1.1. Navigazione in un Grafo

La navigazione di un grafo è nota anche con il nome di **graph traversal** e sostanzialmente è alla base di tutte le operazioni che possiamo effettuare su un grafo.

Un **path** (o percorso) in un grafo è una sequenza di vertici e archi che collega un vertice di partenza a un vertice di arrivo.

Esempio: Path in un Grafo

Consideriamo il grafo in Figura 27. Possiamo notare che esiste un path tra il nodo “Alice” e il nodo “Marcus” che passa per “Bob”. Il fatto che possiamo giungere da Alice a Marcus, passando per Bob, implica che Alice e Marcus sono “conoscenti di conoscenti”.

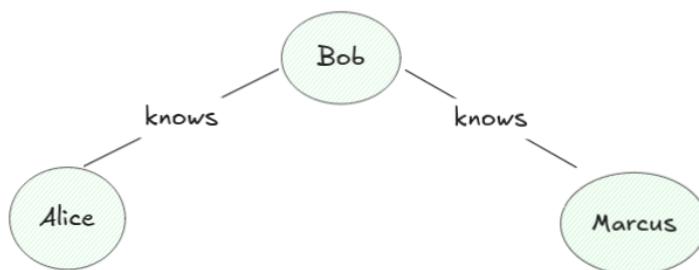


Figura 27: Rappresentazione delle relazioni ‘conoscenti di conoscenti’ tramite un grafo

In riferimento all’esempio precedente, possiamo notare come alla definizione di grafo introdotta da Definizione 5.1° manchi un concetto fondamentale nel contesto delle basi di dati: il dare un nome agli archi, che rappresentano relazioni trai nodi.

Quando andiamo a visitare un grafo, possiamo porci una domanda fondamentale: «**In che ordine dovremmo visitare i nodi?**». Esistono diverse alternative per rispondere a questa domanda, le più comuni sono:

- **Depth-First Search (DFS)**: questa strategia esplora il più lontano possibile lungo ogni ramo prima di tornare indietro. In pratica, si visita un nodo, poi si visita uno dei suoi vicini, e così via, fino a quando non si raggiunge un nodo senza vicini non visitati. A quel punto, si torna indietro e si esplorano gli altri vicini.
- **Breadth-First Search (BFS)**: questa strategia esplora tutti i vicini di un nodo prima di passare ai nodi di livello successivo. In pratica, si visita un nodo, poi si visitano tutti i suoi vicini, poi i vicini dei vicini, e così via.
- **Altri algoritmi specializzati**: esistono molti altri algoritmi di navigazione dei grafi, come Dijkstra per il calcolo del percorso più breve, A*, Dijkstra, Bellman-Ford, e molti altri, ciascuno con le proprie caratteristiche e casi d'uso specifici.

5.1.2. Problemi sui Grafi

I grafi sono utilizzati per modellare una vasta gamma di problemi nel mondo reale. Sono stati dunque sviluppati numerosi problemi classici sui grafi, tra cui:

- **Graph Traversal**: anche se già menzionato in precedenza è importante sottolineare che la navigazione di un grafo è un problema fondamentale, con molte varianti e applicazioni. Consiste nella visita di *tutti i nodi* al suo interno
- **Eulerian Path**: trovare un percorso che attraversa tutti i nodi passando per ogni arco esattamente una volta.
- **Eulerian Cycle**: iniziando da un path euleriano, vorremmo iniziare e terminare nello stesso nodo.
- **Hamiltonian Path**: trovare un percorso che visita ogni nodo esattamente una volta.
- **Hamiltonian Cycle**: in maniera simile a prima, vorremmo un path hamiltoniano che inizi e termini nello stesso nodo.
- **Minimum Spanning Tree**: trovare un albero che oltre a collegare tutti i nodi, minimizza il costo totale degli archi utilizzati.

5.2. Basi di Dati a Grafo

L'idea alla base delle basi di dati a grafo è quella che sia estremamente importante rappresentare i **collegamenti** tra i dati memorizzati. In una base di dati relazionale, le relazioni tra entità sono rappresentate tramite chiavi esterne, che per quanto funzionali, non sono così intuitive, l'utilizzo di un grafo è quanto di più naturale per rappresentare queste relazioni. Di seguito Qualche esempio in cui basi di dati di questo tipo sono particolarmente utili:

- **Social Networks**: le relazioni tra utenti, amici, follower, ecc. sono naturalmente rappresentate come grafi.
- **Recommendation Systems**: le connessioni tra utenti e prodotti possono essere modellate come graf
- **Web Semantico**: le relazioni tra pagine web, link, e contenuti possono essere rappresentate come grafi.

- **Sistemi Informativi Geografici**: le reti stradali, i percorsi di trasporto, e le connessioni geografiche sono spesso modellate come grafi.
- **Bioinformatica**: le reti di interazioni tra proteine, geni, e altre entità biologiche sono spesso rappresentate come grafi.

Figura 28 mostra un esempio di come una base di dati a grafo possa essere utilizzata per rappresentare le relazioni tra utenti in un social network e in un sistema geospaziale.

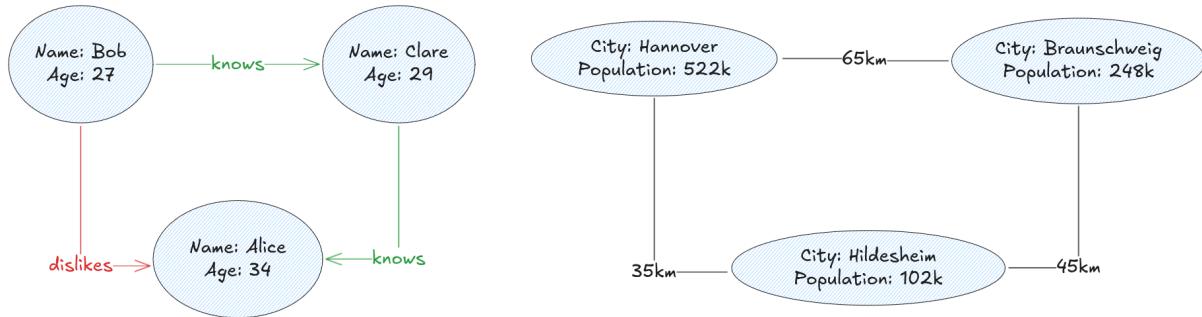


Figura 28: Esempi di basi di dati a grafo in un social network (a sinistra) e in un sistema geospaziale (a destra).

5.2.1. Modello dei Dati

Le basi di dati a grafo utilizzano un modello di dati chiamato **Property Graph Model**, che andiamo di seguito a definire.

Definizione 5.4 (Property Graph Model)

Un **property graph** è un **multigrafo diretto** che memorizza informazioni (proprietà) sia sui nodi che sugli archi. Una **proprietà** è una coppia chiave-valore (es. `Name: Alice`). A volte è possibile avere proprietà *multi-valore*: una chiave e un insieme di elementi associati (es. `Hobbies: {Reading, Hiking, Coding}`).

Per ogni nodo e arco viene definita una proprietà di default chiamata **Id** che identifica univocamente l'elemento all'interno del grafo.

In Figura 29 è mostrato un esempio di property graph che rappresenta una piccola rete sociale.

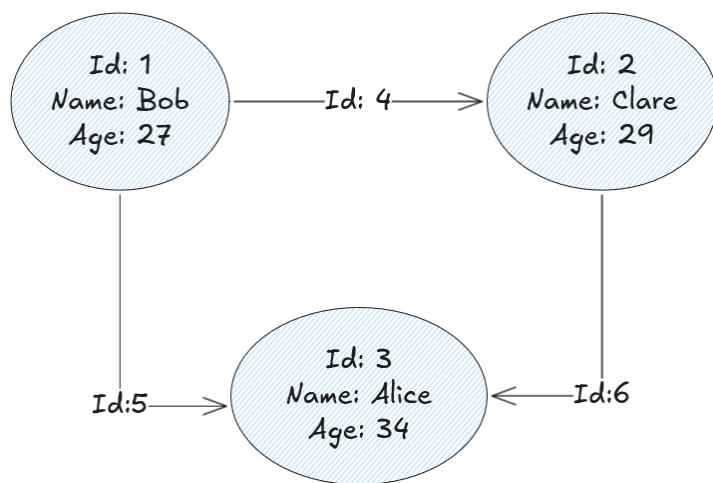


Figura 29: Esempio di Property Graph che rappresenta una rete sociale.

In generale non viene imposto alcun vincolo sul tipo di dati, che possono essere coppie chiave-valore di **tipo arbitrario**. Ad ogni modo è buona pratica utilizzare tipaggio per dare valore semantico ai valori memorizzati:

- Per i *vertici* viene utilizzata una proprietà di *default*, chiamata **Type**
- Per gli *archi* viene utilizzata una proprietà di *default*, chiamata **Label**, optionalmente è anche possibile andare a definire quali edge labels possono connettere certi nodi in base al type che viene specificato per questi

Possiamo notare che l'utilizzo delle proprietà **Type** e **Label** corrisponde circa a dire che un certo nodo o una certa relazione tra nodi fanno parte di una certa **classe**. In questo modo, l'esempio in Figura 29 può essere riformulato reintroducendo il valore semantico delle relazioni nel grafo come viene fatto in Figura 30.

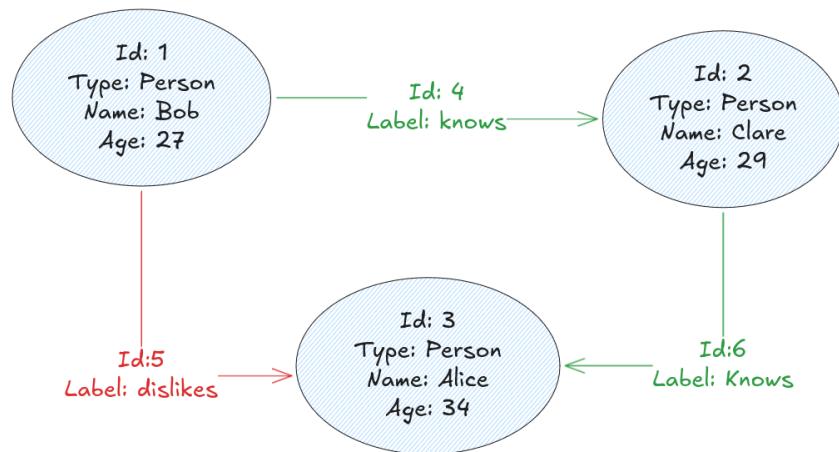


Figura 30: Esempio di Property Graph con tipi e label che rappresenta una rete sociale.

Non è obbligatorio, ma in generale possiamo aspettarci che nodi e archi che fanno parte di una stessa “classe” condividano le stesse proprietà. Introduciamo un ultimo particolare vincolo, che riguarda la presenza di **multi-edge**, questi non possono connettere gli stessi nodi con la stessa «Label» di relazione. Vogliamo evitare questa situazione per evitare di avere **ambiguità** nell'interpretazione del grafo. Figura 31 mostra un esempio di multi-edge non ammesso.

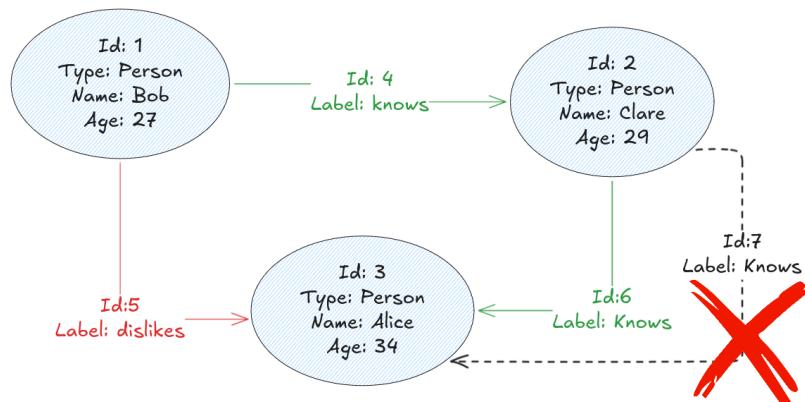


Figura 31: Esempio di multi-edge non ammesso dal momento che ha la stessa label di un altro arco già presente per collegare gli stessi nodi.

5.2.2. Memorizzazione di un Grafo

Questa sezione è dedicata a illustrare come viene salvata in memoria la struttura del grafo. Se per quanto riguarda i vari nodi, potrebbe risultare intuitivo come andare a memorizzarli, la questione sia complica quando andiamo a cercare di rappresentare gli **archi**.

Abbiamo infatti bisogni di raggiungere un buon **tradeoff** tra **memorizzazione rapida** e **possibilità di utilizzare dati in modo efficiente** (es. graph traversal).

Strategia Naive

Una possibile strategia molto semplice è quella di memorizzare l'insieme dei nodi V e l'insieme degli archi E come liste separate. Ogni arco conterrà un riferimento ai nodi che connette. Di seguito andiamo ad elencare i pro e contro di questa strategia:

- L'**inserimento** è molto rapido, dal momento che basta aggiungere un nuovo nodo o un nuovo arco alla rispettiva lista, l'unica cosa di cui abbiamo bisogno è un puntatore alla prossima posizione in cui andiamo a scrivere ✓
- La determinazione delle **adiacenze** è particolarmente inefficiente, dal momento che dobbiamo scorrere l'intera lista degli archi per trovar quelli di interesse ✗
- La gestione delle operazioni di **cancellazione** è particolarmente inefficiente, dal momento che dobbiamo scorrere l'intera lista di interesse, cancellare il dato e ricompattare il tutto ✗

Matrice di Adiacenza

Una possibile strategia possibilmente più efficace è quella di utilizzare una **matrice di adiacenza**. In questo caso andremo a creare una matrice quadrata di dimensione $|V| \times |V|$. Ogni riga e colonna andrà a rappresentare i vertici v_1, \dots, v_n .

Nel caso di grafi senza archi, la matrice di adiacenza conterrà unicamente zeri. Nel caso in cui siano presenti degli archi abbiamo due possibilità in base al tipo di grafo:

- Per **grafi non diretti a-ciclici**: se esiste un arco tra il nodo v_i e v_j , allora andremo a scrivere uno nella cella (i, j) e nella cella (j, i) della matrice di adiacenza.
- Per **grafi non diretti con cicli**: nel caso di un loop tra il v_i e sé stesso, andremo a scrivere 2 nella cella (i, i) della matrice di adiacenza.
- Per **grafi diretti a-ciclici**: se esiste un arco dal nodo v_i al nodo v_j , allora andremo a scrivere uno nella cella (i, j) della matrice di adiacenza.
- Per **grafi diretti con cicli**: nel caso di un loop tra il v_i e sé stesso, andremo a scrivere 1 nella cella (i, i) della matrice di adiacenza.

Osservazione

Nel caso di grafi non diretti la matrice di adiacenza sarà sempre simmetrica, mentre non lo sarà necessariamente nel caso di grafi diretti.

Osservazione

Il motivo per cui in un grafo non diretto con ciclo andiamo a scrivere due nella cella (i, i) sta nel fatto che in grafi non diretti ogni arco è contato in maniera simmetrica, dunque un loop viene contato due volte. Ciò non avviene nel caso di grafi diretti.

Andiamo ora a mostrare cosa succede nel caso di **multigrafi non diretti**:

- Nel caso di multigrafo diretto **a-ciclico**: se esistono k archi tra il nodo v_i e v_j , allora andremo a scrivere k nella cella (i, j) e nella cella (j, i) della matrice di adiacenza.
- Nel caso di multigrafo diretto **con cicli**: nel caso in cui si verifichino k loops del tipo v_i, v_i , andremo a scrivere $2 \cdot k$ nella cella (i, i) della matrice di adiacenza.

Nel caso in cui di **multigrafi diretti** avremo la seguente situazione:

- Nel caso di multigrafo diretto **a-ciclico**: se esistono k archi dal nodo v_i al nodo v_j , allora andremo a scrivere k nella cella (i, j) della matrice di adiacenza.
- Nel caso di multigrafo diretto **con cicli**: nel caso in cui si verifichino k loops del tipo v_i, v_i , andremo a scrivere k nella cella (i, i) della matrice di adiacenza

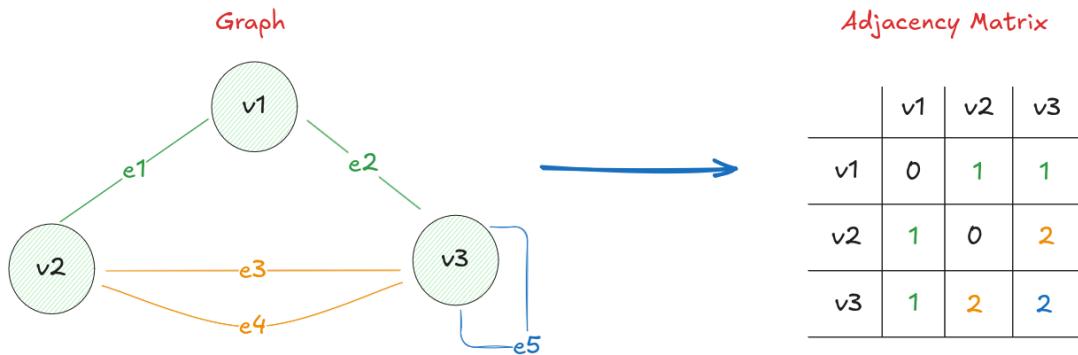


Figura 32: Esempio di multigrafo non diretto con cicli (a sinistra) e la sua matrice di adiacenza (a destra).

Di seguito andiamo ad elencare vantaggi e svantaggi di questo approccio:

- La determinazione delle **adiacenze** è molto rapida, dal momento che basta accedere alla cella di interesse della matrice ✓
- È estremamente facile **inserire** un nuovo arco, dal momento che basta incrementare il valore nella cella di interesse ✓
- È particolarmente **costoso** dal punto di vista della memoria, dal momento che dobbiamo memorizzare una matrice di dimensione $|V| \times |V|$, specialmente quando i nodi sono molti la dimensione della matrice potrebbe diventare di difficile gestione ✗
- Spesso e volentieri i grafi sono **sparsi**, ovvero hanno pochi archi rispetto al numero di nodi, in questo caso la matrice di adiacenza conterrà molti zeri, andando così a sprecare memoria ✗
- Le operazioni di **inserimento di nuovi nodi** sono particolarmente costose, dal momento che dobbiamo aumentare la dimensionalità della matrice ✗

- Non è possibile memorizzare iper-grafi, dal momento che la matrice di adiacenza può rappresentare solamente archi che connettono due nodi X
- Determinare **tutte le adiacenze** di un nodo è particolarmente inefficiente, dal momento che dobbiamo scorrere l'intera riga o colonna della matrice di interesse, che nel caso di grafi con molti nodi potrebbe essere un'operazione lenta X

Matrice di Incidenza

Un possibile alternativa alla matrice di adiacenza è la **matrice di incidenza**. In questo caso andremo a creare una matrice di dimensione $|V| \times |E|$. Ogni riga andrà a rappresentare i vertici v_1, \dots, v_n , mentre ogni colonna andrà a rappresentare gli archi e_1, \dots, e_m .

I valori nelle celle della matrice di incidenza saranno assegnati in maniera simile a quanto visto per la matrice di adiacenza, con la differenza che dovremo tenere conto della positività e della negatività delle incidenze.

Figura 33 e Figura 34 mostrano le differenze nella matrice di adiacenza nel caso di grafi diretti e indiretti.

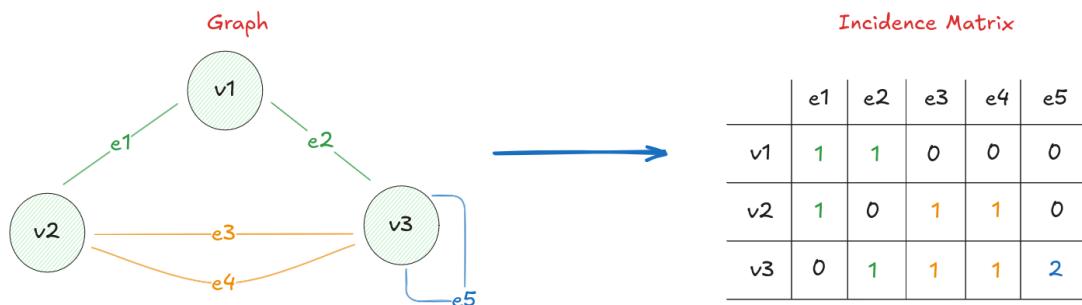


Figura 33: Esempio di multigrafo diretto con cicli (a sinistra) e la sua matrice di incidenza (a destra).

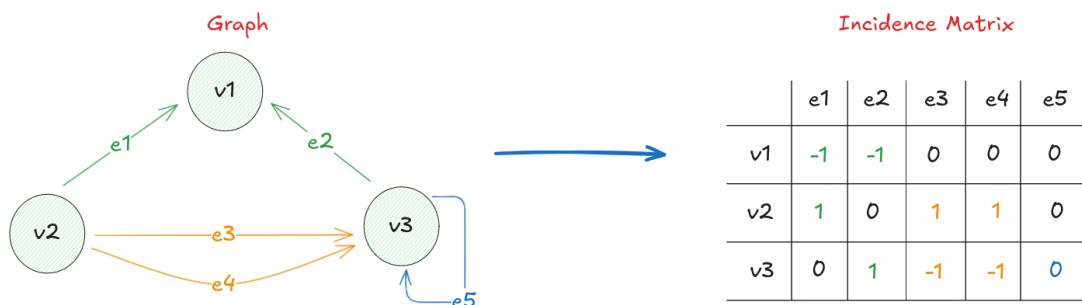


Figura 34: Esempio di multigrafo diretto con cicli (a sinistra) e la sua matrice di incidenza (a destra).

Di seguito andiamo a riportare vantaggi e svantaggi di questo approccio:

- Non avremo colonne di soli zeri, dal momento che queste rappresentano gli archi e usiamo colonne solo per gli archi esistenti ✓
- È possibile memorizzare iper-archi ✓
- Dal punto di vista dello **storage** si tratta di un approccio particolarmente **intensivo** ($n \times m$) X

- Nel caso di grafi con molti nodi, le **colonne** avranno comunque **molti zeri**, dal momento che una colonna rappresenta un arco
- Determinare le **adiacenze** per un vertice richiede il lookup di tutta la riga corrispondente **X**
- L'**inserimento** di un nuovo nodo richiede l'aggiunta di una nuova riga alla matrice, risultando **costoso X**

Invece di utilizzare **matrici** per rappresentare i grafi, un'alternativa più efficiente è quella di utilizzare delle **liste di adiacenza**, questo permette di risparmiare memoria fondamentale e di velocizzare l'esecuzione di operazioni comuni sui grafi.

Liste di Adiacenza

L'idea alla base di una lista di adiacenza è quella di memorizzare per ogni nodo, una lista di nodi adiacenti. In questo modo, possiamo rappresentare un grafo come un array di liste. Figura 35 mostra come questo approccio possa essere implementato nel caso di un grafo indiretto a sinistra e di uno diretto a destra.

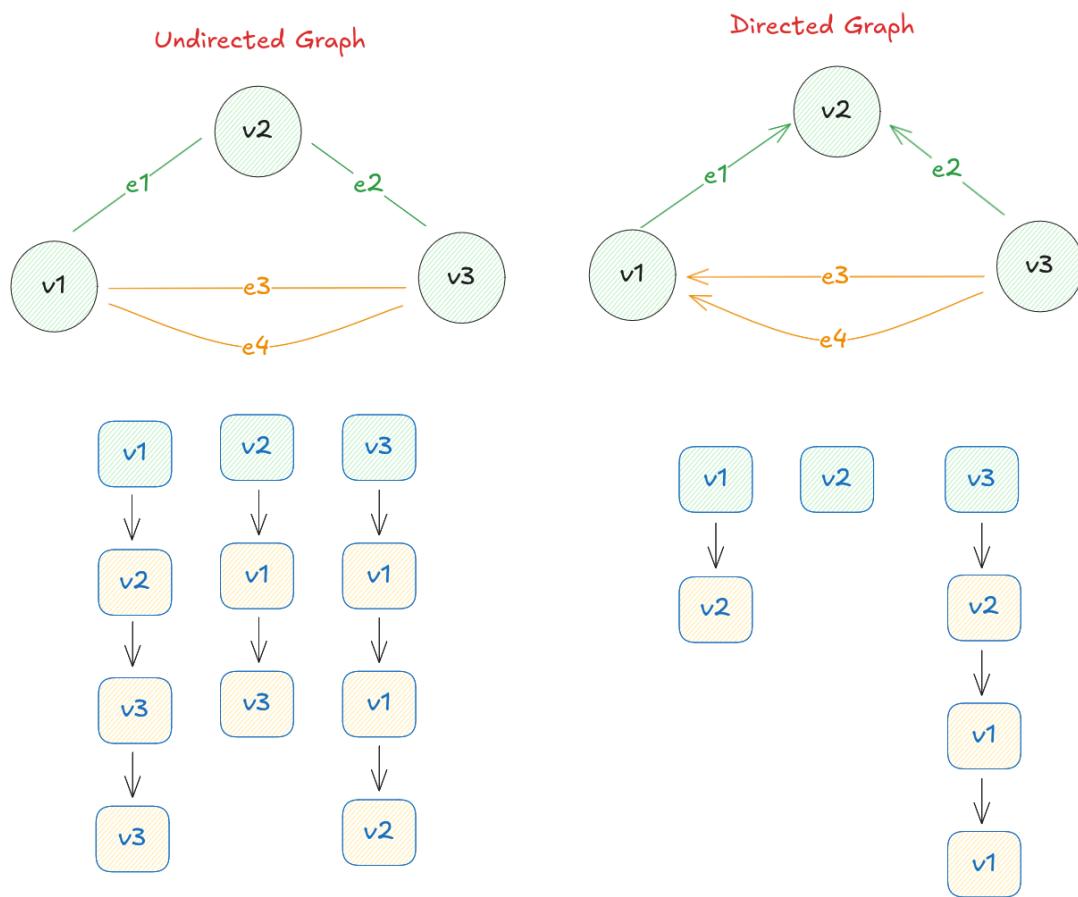


Figura 35: Esempio di rappresentazione di un grafo indiretto (a sinistra) e diretto (a destra) tramite liste di adiacenza.

Di seguito andiamo ad elencarne vantaggi e svantaggi:

- Effettuare un **lookup** di tutti i vertici adiacenti a un nodo è molto **efficiente**, dal momento che basta accedere alla lista corrispondente **✓**

- Vengono memorizzato soltanto le informazioni rilevanti, non abbiamo **overhead** di memoria ✓
- L'**inserimento** di un nuovo nodo è **efficiente**, dal momento che basta aggiungere una nuova lista all'array ✓
- L'**inserimento** di un nuovo arco è **efficiente**, dal momento che basta aggiungere il nodo di destinazione alla lista del nodo/i di partenza/arrivo ✓
- È possibile memorizzare iper-archi ✓
- Determinare l'esistenza di un **arco specifico** può essere costoso, dal momento che potrebbe essere necessario scorrere l'intera lista di adiacenza ✗

Osservazione

Questa struttura per quanto efficiente non consente ancora di andare a memorizzare le proprietà associate a nodi ed archi, ma non si tratta di una operazione complicata, basta infatti memorizzare le **proprietà di un nodo** possiamo memorizzarle nel punto di **partenza della lista**, mentre le **proprietà di un arco** possono essere memorizzate nel **nodo di destinazione** all'interno della lista di adiacenza.

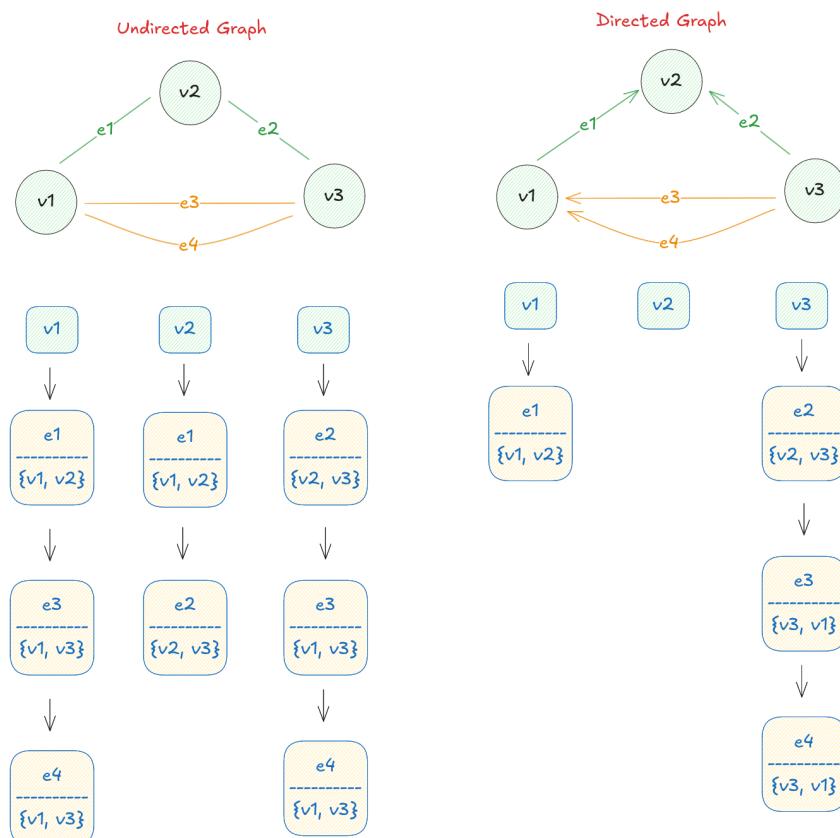


Figura 36: Esempio di rappresentazione di un grafo indiretto (a sinistra) e diretto (a destra) tramite liste di incidenza.

Liste di Incidenza

Un'ulteriore possibile strategia per memorizzare un grafo è quella di utilizzare delle **liste di incidenza**. In questo caso, per ogni nodo andremo a memorizzare una **lista di archi** che sono

incidenti su quel nodo. Figura 36 mostra come questo approccio possa essere implementato nel caso di un grafo indiretto a sinistra e di uno diretto a destra.

Osservazione

Nel caso in cui volessimo andare a memorizzare degli **iper-grafi** questo sarebbe possibile, basterebbe andare ad aggiungere alla lista di incidenza relativa all'iper-arco tutti i nodi che questo coinvolge.

5.2.3. Graph Database Management Systems & API

Il focus di questa sezione è quello di andare ad illustrare come possiamo **accedere** ai dati memorizzati in un graph database. In realtà abbiamo diversi sistemi per farlo. Di seguito andiamo a mostrarne alcuni ognuno con le sue caratteristiche.

TinkerPop

Si tratta di un framework open-source per la gestione di grafi che fornisce un API standardizzata per interagire con un graph db.

```
1 Graph g = TinkerGraph.open();  
2 Vertex alice = g.addVertex("name", "Alice");  
3 alice.property("age", 34);  
4 Vertex bob = g.addVertex("name", "Bob");  
5 alice.addEdge("knows", bob, "since", 2020);
```

Gremlin

Di seguito andiamo a vedere un semplice codice per effettuare traversal del grafo creato tramite *method chaining*:

```
1 g.traversal().V() // prendiamo la lista dei vertici  
2   .has("name", "Alice") // filtriamo per il vertice con nome "Alice"  
3   .out("knows") // andiamo agli archi in uscita con label "knows"  
4   .values("name"); // prendiamo i nomi dei vertici raggiunti
```

Gremlin

Cypher

Un alternativa all'approccio di TinkerPop è quello di utilizzare un linguaggio di query tramite il quale andare a **descrivere pattern** riguardo **nodi** o **paths**. Il modo in cui possiamo definire questi pattern avviene tramite l'utilizzo del linguaggio **Cypher**. Di seguito andiamo a vederne alcuni elementi sintattici:

- **START** : viene utilizzato per specificare i *nodi* di partenza per una query
- **MATCH** : viene utilizzato per la specifica dei *pattern* da ricercare
- **WHERE** : viene utilizzato per specificare delle *condizioni* sui *nodi* o sugli *archi*
- **RETURN** : viene utilizzato per specificare quali dati vogliamo vengano restituiti dalla query

In riferimento al grafo giocattolo costruito nell'esempio di prima possiamo costruire una query come quella descritta di seguito per trovare tutti i nodi adiacenti ad "Alice":

```
1 START alice = (people_idx, name, "Alice")    // selezioniamo il  
nodo di partenza  
2 MATCH (alice)-[:knows]->(aperson) // definiamo il pattern da cercare  
3 return (aperson)
```

Cypher

Quello che viene fatto nella query di sopra è:

- Selezionare il nodo di partenza “Alice” tramite l’uso dell’indice `people_idx`
- Definire il pattern da cercare, ovvero tutti i nodi connessi ad “Alice” tramite un arco con label “knows”
- Restituire i nodi trovati.

Osservazione

Si noti come, dal momento che non sono stati specificati vincoli sui nodi o sugli archi, la query restituirà tutti i nodi adiacenti ad “Alice” tramite un arco con label “knows”, non necessariamente nodi con il suo stesso tipo (es. `Person`).

5.3. Neo4j

Al contrario di moltissime altre tecnologie per la gestione dei dati, Neo4j è un **Graph Database nativo**, ovvero è stato progettato fin dall'inizio per memorizzare e gestire dati in forma di grafo.

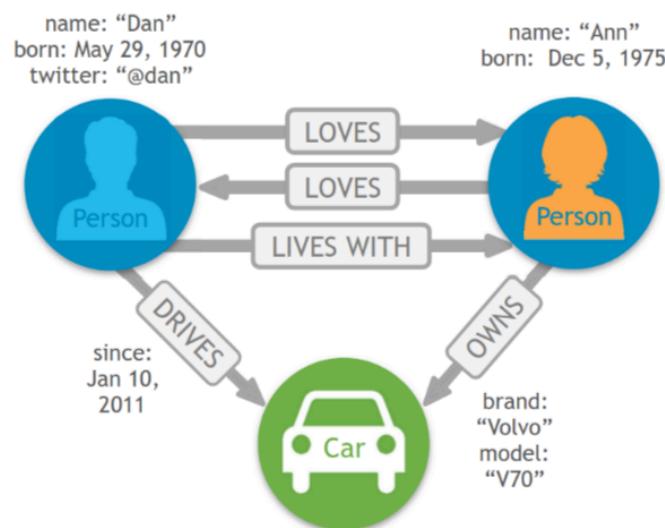
Anche un database relazionale può essere visto come una struttura a grafo, ma in questo caso i grafi sono **derivati** dalle tabelle e dalle relazioni tra esse. Questo comporta che le operazioni su database relazionali possano diventare estremamente complesse e inefficienti quando si tratta di navigare attraverso molteplici relazioni.

5.3.1. Property Graph Model

Questa sezione va velocemente a riassumere i concetti fondamentali alla base di un qualsiasi database basato su un grafo, ovvero il *property graph model*. Di seguito andiamo a rielencare i concetti principali:

- **Nodi**: rappresentano gli oggetti (o entità) del grafo, ogni nodo può avere una **label** (o più di una) che ne definisce il tipo (es. Person, Movie, ecc.)
- **Relazioni**: rappresentano le connessioni (archi) tra i nodi, ogni relazione ha una **direzione** e una **label** che ne definisce il tipo (es. KNOWS, ACTED_IN, ecc.)
- **Proprietà**: sia i nodi che le relazioni possono avere proprietà, che sono coppie chiave-valore che memorizzano informazioni aggiuntive (es. name: "Alice", since: 2020, ecc.)

L'immagine che segue mostra un esempio di un property graph model che contiene molteplici relazioni tra nodi, molte proprietà e tipi di nodi e relazioni diversi:



L'idea alla base di Neo4j è quella di fornire uno strumento il più intuitivo e vicino possibile a ciò che un utente si aspetta quando pensa a un grafo.

5.3.2. Graph Querying: Qualche Esempio

Dato un grafo abbiamo bisogno di un modo per poter interrogare i dati in esso contenuti. Neo4j fornisce un linguaggio di query chiamato **Cypher**, che è stato progettato specificamente per lavorare con grafi. Si tratta di un linguaggio **dichiarativo** che pone alla sua base il concetto di

pattern matching sui grafi, in maniera simile a quello che facciamo con le *espressioni regolari* sui testi. Figura 37 mostra un esempio di come possiamo rappresentare dei pattern in Cypher.

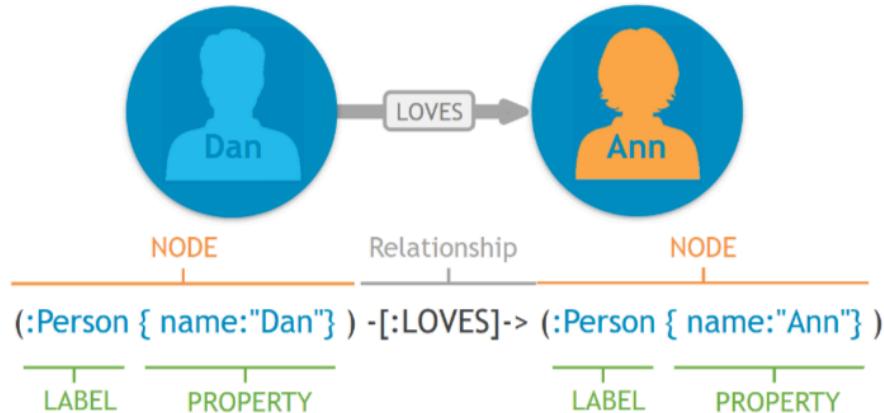


Figura 37: Esempio di pattern matching in Cypher per trovare nodi e relazioni specifiche all'interno di un grafo.

Dato il pattern rappresentato in Figura 37 possiamo andare a **creare** o a **ricercare** nodi e relazioni all'interno del grafo.

Esempio: Creazione della relazione 'Loves' tra 'Dan' e 'Ann'

```

1 CREATE
2   (:Person {name: "Dan"})
3   - [:LOVES] ->
4   (:Person {name: "Ann"})
    
```

Cypher

Esempio: Ricerca delle persone amate da 'Dan'

```

1 MATCH
2   (:Person{name: "Dan"})
3   - [:LOVES] -> (whom)
4 RETURN whom
    
```

Cypher

Nell'esempio di sopra non è stato specificato che l'entità amata da “Dan” dovesse essere di tipo `Person`, dunque la query restituirà qualsiasi entità connessa a “Dan” tramite una relazione di tipo `LOVES`. È tuttavia possibile aggiungere ulteriori vincoli per restringere i risultati ottenuti.

Vediamo ora un esempio leggermente più complesso. Consideriamo il grafo in Figura 38, che rappresenta una rete sociale in cui gli utenti sono connessi tramite relazioni di amicizia e hanno ristoranti preferiti.

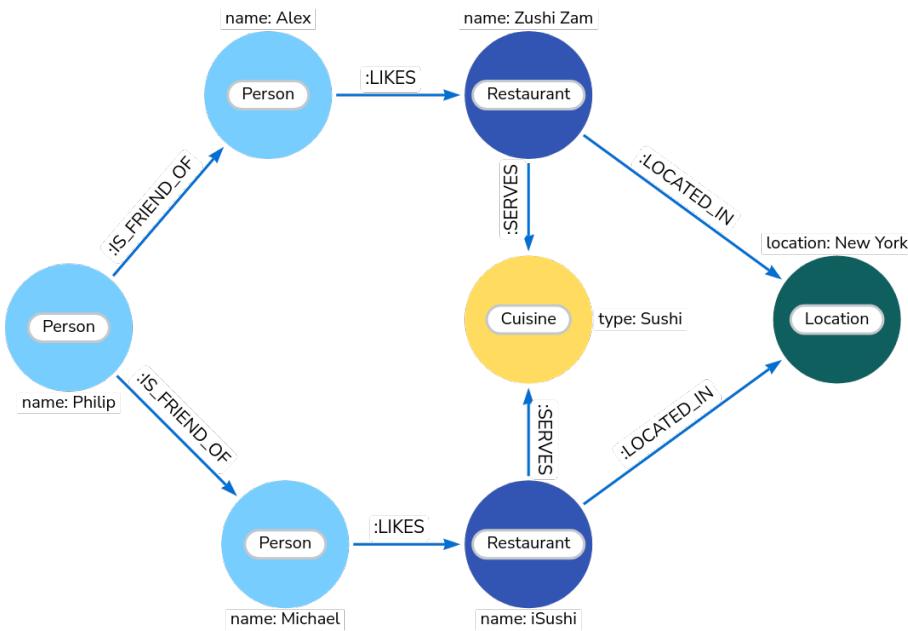


Figura 38: Grafo che rappresenta una rete sociale con relazioni di amicizia e ristoranti preferiti.

Vorremo andare a trovare i ristoranti che sono graditi dagli amici di “Philip” che servono “sushi” locati a “New York”. La query in Cypher per ottenere queste informazioni è la seguente:

```

1 MATCH (person:Person) -[:IS_FRIEND_OF]-> (friend),
2     (friend) -[:LIKES]-> (restaurant),
3     (restaurant) -[:LOCATED_IN]-> (loc:Location),
4     (restaurant) -[:SERVES]-> (type:Cuisine),
5 WHERE person.name = 'Philip'
6 AND loc.location = 'New York'
7 AND type.cuisine = 'Sushi'
8 RETURN restaurant.name

```

Cypher

5.3.3. Introduzione a Cypher

Dopo aver visto qualche esempio di query eseguita in Cypher andiamo a vedere più nel dettaglio la **sintassi** e gli **elementi fondamentali** per costruire query con questo linguaggio.

Sintassi

- un **nodo** viene rappresentato tramite delle parentesi tonde `()`, al cui interno possiamo specificare una **label** e delle **proprietà**. Esempio: `(n:Person {name: "Alice", age: 30})`
- una **relazione** viene rappresentata tramite una freccia `-->` o `<--`, al cui interno possiamo specificare una **label** e delle **proprietà** tra parentesi quadre `[]`. Esempio: `-[:KNOWS {since: 2020}]->`
- un **pattern** viene di solito costruito tramite una combinazione di nodi e relazioni: `()-[]-()`, `()-[]->()`, `()<-[]-()`.

Componenti di una Query

I due componenti principali di una query in Cypher sono sostanzialmente due:

- **MATCH** : viene utilizzato per specificare i pattern da cercare all'interno del grafo.

- **RETURN** : viene utilizzato per specificare quali dati che hanno registrato una corrispondenza nella clausola `MATCH` vogliamo vengano restituiti dalla query

Esempio: Basica Query in Cypher 2

```
1 MATCH (m:Movie) // considera tutti i nodi m      Cypher
    di tipo Movie
2 RETURN m          // restituisce tutti i nodi m trovati
```

Esempio: Basica Query in Cypher 2

```
1 MATCH (p:Person)-[r:ACTED_IN]-(m:Movie)      Cypher
2 RETURN p, r, m
```

In questo caso `MATCH` e `RETURN` sono due parole chiave del linguaggio; `p`, `r` e `m` sono delle **variabili** che rappresentano rispettivamente i nodi di tipo `Person`, le relazioni di tipo `ACTED_IN` e i nodi di tipo `Movie`.

È anche possibile utilizzare interi pattern come variabili, per esempio nel caso in cui vogliamo vedere dei **path** completi. Lo vediamo nell'esempio che segue.

Esempio: Query di Path in Cypher

```
1 MATCH p = (p:Person)-[r:ACTED_IN]-(m:Movie)      Cypher
2 RETURN p
```

In questo caso la **variabile** `p` rappresenta l'intero path che va dal nodo di tipo `Person`, tramite la relazione di tipo `ACTED_IN`, fino al nodo di tipo `Movie`.

È anche possibile andare ad accedere in maniera selettiva alle **proprietà** dei nodi e delle relazioni. Per fare ciò la sintassi è `{variable}.{property_key}`. Lo vediamo nell'esempio che segue.

Esempio: Query con Proprietà in Cypher

```
1 MATCH (p:Person)-[r:ACTED_IN]-(m:Movie)      Cypher
2 RETURN p.name, m.title
```

In questo caso stiamo restituendo solamente le proprietà `name` del nodo di tipo `Person` e `title` del nodo di tipo `Movie`.

Osservazione

Si noti che questa operazione è molto simile a quella di eseguire una **proiezione** in algebra relazionale o ad applicare costrutti equivalenti in SQL.

Un'altra questione che seppur di minore importanza è necessario menzionare è il funzionamento e la gestione del **casing** in questo linguaggio:

- le *label* di nodi, i *tipi* delle relazione, le *property key* sono sempre **case sensitive**
- tutte le *keyword* di Cypher sono **case insensitive**

5.3.4. Indici e vincoli

Un costrutto molto importante è quello dei **vincoli** (constraints), che permettono di imporre delle regole sui dati memorizzati all'interno del grafo, in modo da garantire l'integrità, coerenza dei dati e altre interessanti proprietà.

All'interno di Neo4j e in Cypher è inoltre possibile andare a specificare delle maniere per **ottimizzare** le performance di alcune operazioni, lo strumento utilizzato, similmente a quanto avviene con altre tecnologia è quello degli **indici**.

Vincoli (Constraints) Unique

Andiamo a vedere ora il principale vincolo che possiamo andare a specificare in Neo4j, ovvero il vincolo di **unicità**. Questo assolve a due funzioni fondamentali:

- Garantisce che non esistano due nodi con la stessa proprietà chiave-valore per una certa label
- Permette un **accesso molto veloce** a nodi che corrispondono certe coppie «label-property»

Per la creazione di questo vincolo andiamo a utilizzare la seguente sintassi:

```
1 CREATE CONSTRAINT ON (label:Label)          Cypher
2 ASSERT label.property_key IS UNIQUE
```

Esistono in verità **tre tipologie** di vincoli di unicità che andiamo di seguito ad illustrare:

- **Unique Node Property**: garantisce che non esistano due nodi con la stessa proprietà chiave-valore per una certa label.

```
1 CREATE CONSTRAINT ON (label:Label)          Cypher
2 ASSERT label.property_key IS UNIQUE
```

- **Node Property Existence**: impedisce la creazione di nodi che per una certa label non abbiano valori

```
1 CREATE CONSTRAINT ON (label:Label)          Cypher
2 ASSERT exists(label.name)
```

- **Relationship Property Existence**: impone che per un certo tipo di relazione esista sempre una certa proprietà

```
1 CREATE CONSTRAINT ON ()-[rel:REL_TYPE]->()      Cypher
2 ASSERT exists(rel.name)
```

Indici

Come già visto in altri sistemi e linguaggi, l'utilizzo di **indici** serve a garantire un **lookup** veloce di nodi che soddisfano una certa condizione di tipo «label-property». In Neo4j possiamo creare un indice tramite la sintassi che segue:

```
1 CREATE INDEX ON :Label(property_key)
```

Cypher

Il lookup efficiente e rapido è garantito quando andiamo ad utilizzare i seguenti **predicati** che utilizzano gli indici by design:

- *Uguaglianza*: =, <>
- STARTS WITH
- CONTAINS
- ENDS WITH
- Ricerche su **range** di valori
- Ricerche su **valori null**

È importante fare una distinzione tra l'utilizzo di indici in questo contesto e in altri sistemi come quello relazionale: se nel mondo relazionale, l'uso dell'indice è fatto per **cercare righe** di certe tabelle, in un graph db l'indice serve a **cercare nodi** di partenza per una certa query.

5.3.5. Scrittura di Query in Cypher: Utilizzi Avanzati

Dopo aver visto le basi del linguaggio Cypher andiamo ora a vedere qualche costrutto più avanzato che ci consentirà di scrivere query più complesse e potenti.

Clausola CREATE

La clausola **CREATE** viene utilizzata per andare a creare nuovi nodi e relazioni all'interno del grafo. La sintassi è molto simile a quella utilizzata per specificare i pattern nella clausola **MATCH**:

```
1 CREATE (m:Movie(title: 'Mystic River', released:2003))  
2 RETURN m
```

Cypher

Nell'esempio di sopra stiamo andando a creare un'entità **Movie** con le proprietà **title** e **released** e ritorniamo all'utente l'entità appena creata.

È possibile andare a utilizzare la clausola **CREATE** anche per creare relazioni tra nodi esistenti:

```
1 MATCH (m:Movie {title: 'Mystic River'})  
2 MATCH (p:Person {name: 'Kevin Bacon'})  
3 CREATE (p)-[r:ACTED_IN {roles: ['Sean']}]->(m)  
4 RETURN p, r, m
```

Cypher

In questo caso vediamo come sia stato prima effettuato il **lookup** dei nodi di interesse tramite la clausola **MATCH**, per poi andare a creare la relazione **ACTED_IN** tra i due nodi, con una proprietà **roles** che è una lista che conterrà i ruoli interpretati dall'attore nel film.

Osservazione

Una volta che andiamo ad utilizzare la clausola `CREATE` questa provvederà a creare tutto ciò che viene specificato al suo interno. Nel caso in cui vogliamo creare una relazione tra due nodi abbiamo due possibilità diverse:

- *entrambi i nodi sono già esistenti*: in tal caso andremo soltanto a creare la relazione
- *solo uno dei nodi o nessuno esiste*: ipotizzando che la persona con nome `Kevin Bacon` non fosse già esistente, la clausola `CREATE` la avrebbe creata per fare in modo che il path desiderato fosse coerente

Clausola `SET`

Oltre a creare entità, è possibile anche andare a modificare le proprietà delle entità, per questo scopo utilizziamo la clausola `SET`:

```
1 MATCH (m:Movie {title: 'Mystic River'})  
2 SET m.released = 2004  
3 SET m.tagline = 'A movie about life and loss'  
4 RETURN m
```

Cypher

La query presentata sopra va sia a modificare una proprietà già esistente (`released`), sia ad aggiungerne una nuova (`tagline`).

Clausola `MERGE`

La clausola `MERGE` viene utilizzata per creare nodi o relazioni in modalità “**upsert**”, combinando le funzionalità di `MATCH` e `CREATE`. In pratica, `MERGE` cerca di trovare un nodo o una relazione che corrisponde al pattern specificato; se non lo trova, lo crea.

```
1 MERGE (p:Person {name: 'Tom Hanks'})  
2 RETURN p
```

Cypher

Avvertenza

È importante considerare che l'utilizzo di questa clausola potrebbe portare a risultati inattesi se utilizzata senza la dovuta attenzione. Supponiamo di voler andare a rivisitare la l'operazione di upsert presentata sopra, ma questa volta aggiungendo il la proprietà `oscar` alla persona `Tom Hanks`. A una prima occhiata potrebbe venirci in mente di utilizzare la seguente query:

```
1 MERGE (p:Person {name: 'Tom Hanks', oscar: true})  
2 RETURN p
```

Cypher

Tuttavia, questa query non funzionerà come ci aspettiamo. Se esiste già un nodo `Person` con il nome “Tom Hanks” ma senza la proprietà `oscar`, la clausola `MERGE` non lo troverà come corrispondente e creerà un nuovo nodo con la proprietà `oscar` impostata a `true`.

Di conseguenza, ci ritroveremo con due nodi distinti per “Tom Hanks”, uno con la proprietà `oscar` e uno senza. La query giusta da utilizzare in questo caso è la seguente:

```
1 MERGE (p:Person {name: 'Tom Hanks'})  
2 SET p.oscar = true  
3 RETURN p
```

Cypher

È possibile utilizzare la clausola `MERGE` anche per creare relazioni in modalità upsert. Oltre a tutte queste funzionalità, `MERGE` supporta la possibilità di specificare delle **azioni** da eseguire in base al fatto che l’entità sia stata trovata o creata, tramite l’uso delle clausole `ON MATCH` e `ON CREATE`.

```
1 MERGE (p:Person {name: 'Your Name'})  
2 ON CREATE SET p.created = timestamp(), p.updated = 0  
3 ON MERGE SET p.updated = p.updated + 1  
4 RETURN p.created, p.updated
```

Cypher

5.3.6. Data Ingestion in Neo4j

Dopo aver visto come interrogare la nostra base di dati, andiamo a vedere come effettuare una delle operazioni più importanti nel mondo del data management, ovvero l’**ingestione** dei dati all’interno del database.

Cypher è dotato di una clausola specifica per questo scopo, che permette di partire da un file `.csv`, si chiama appunto `LOAD CSV`. Di seguito ne mostriamo alcune caratteristiche:

- Permette di caricare dati in formato `.csv` da un file locale o da un URL
- Preso in input un file `.csv` fornisce uno **stream** di record che possono essere processati tramite le classiche clausole messe a disposizione da Cypher (`MATCH`, `CREATE`, `SET`, ecc.)
- Supporta l’esecuzione di **operazioni transazionali** sul grafo esistente
- È in grado di convertire i valori letti dal file `.csv` in tipi di dati nativi di Neo4j (es. stringhe, numeri, booleani, ecc.)

Ipotizziamo che il nostro property graph model sia descritto in Figura 39.

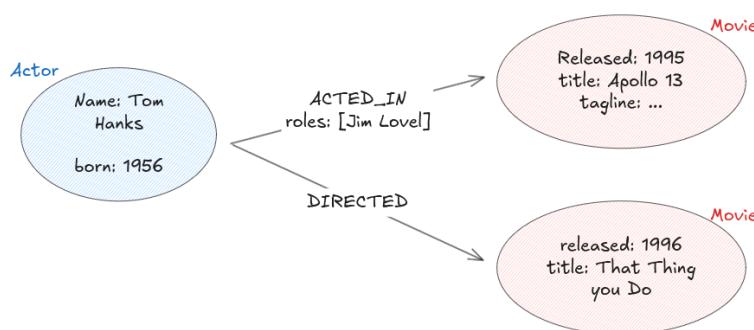


Figura 39: Esempio di property graph model che rappresenta persone e film con relazioni

Per ogni record del file `.csv` da utilizzare per l'ingestion vogliamo distinguere i casi seguenti:

- Creiamo un nodo di tipo `Person` o `Movie` se questo non esiste già:

```
1 CREATE (:Person {  
2   name:row.name,  
3   born:toInt(row.born)  
4 });
```

Cypher

- Cerchiamo nodo di inizio e nodo di fine e creiamo tra loro una relazione:

```
1 MATCH (m:Movie {title:row.movie}),  
2       (p:Person {name: row.person}),  
3 CREATE (p)-[:ACTED_IN {roles:split(r.roles)}]->(m);
```

Cypher

Una volta definiti i classici statements da utilizzare per fare ingestion, è necessario andare a mostrare in quale maniera dire a Neo4j di effettuare ingestion:

```
1 [USING PERIODIC COMMIT]    // attiva le transazioni per      Cypher  
batch  
2  
3 LOAD CSV    // statement per iniziare a effettuare ingestion  
4  
5 WITH HEADERS // opzionalmente usare la prima riga del file come  
chiave per gli elementi letti  
6  
7 FROM "url"    // path o url da cui leggere il file csv  
8  
9 AS row        // ritorna ogni riga come lista di stringhe o mappa  
10  
11 FILETERMINATOR ";"    // specifica qual è il separatore dei  
valori  
12  
13  
14 ... gli altri statement che specificano come fare ingestion ...
```

6. Gestione di Dati Distribuiti

Dopo aver introdotto varie famiglie di basi di dati, e aver visto come in moltissimi contesti, la loro nascita è dovuta al fatto di permettere una scalabilità orizzontale, andremo ora a vedere come effettivamente queste basi di dati permettono di gestire **dati distribuiti** su diversi nodi.

6.1. Concetti di Base

Come già menzionato, il principio alla base delle basi di dati distribuite è quello della **scalabilità**. Nel particolare, esistono due tipologie di scalabilità di cui si può parlare:

- **Scale up** (o *verticale*): consiste nell'aggiunta di più potenza computazionale ad un singolo server che ospita la base di dati. Questo consente tipicamente di migliorare le prestazioni, ma ha dei limiti fisici e di costo.
- **Scale out** (o *orizzontale*): consiste nell'aggiunta di più nodi (server) al sistema di database. Questo approccio consente di distribuire il carico di lavoro e i dati.

Quando andiamo a parlare di basi di dati distribuite, andremo a riferirci alle seguenti componenti fondamentali:

- **Database distribuito**: si tratta di un insieme di dati che sono *logicamente interconnessi*
- **DBMS distribuito**: si tratta di un sistema per la gestione di *database distribuiti*, che ha la fondamentale capacità di rendere la **distribuzione trasparente**

Trasparenza

Proprio questo concetto di **trasparenza** è fondamentale da approfondire. In generale per un utente non dovrebbe essere rilevante come internamente il DBMS gestisce lo storage dei dati e l'esecuzione di query. Esistono diverse tipologie di trasparenza:

- **Access Transparency**: l'accesso ai dati è indipendente dalla struttura della rete o dall'organizzazione fisica dei dati.
- **Location Transparency**: gli utenti non devono conoscere come i dati sono distribuiti.
- **Replication Transparency**: gli utenti non devono conoscere se stanno accedendo a dati replicati o meno.
- **Fragmentation Transparency**: la frammentazione e lo sharding sono tipicamente gestiti internamente. Un utente dovrebbe poter effettuare query alla base di dati come se fosse un'unica entità, senza preoccuparsi di come i dati sono stati frammentati.
- **Migration Transparency**: i dati possono essere spostati tra nodi senza che gli utenti ne siano consapevoli.
- **Concurrency Transparency**: il sistema deve gestire l'accesso concorrente ai dati in modo che gli utenti non debbano preoccuparsi di conflitti o problemi di coerenza.
- **Failure Transparency**: il sistema deve essere in grado di gestire guasti dei nodi o della rete senza che gli utenti ne siano consapevoli.

6.2. Guasti nei Sistemi Distribuiti

È importante andare a considerare come in un sistema distribuito, i guasti sono inevitabili. Esistono diverse tipologie di guasti che possono verificarsi. Lo scopo di questa sezione è quello di andare a presentarli brevemente assieme alle tecniche più comuni per mitigarli.

Il primo caso, più semplice è quello in cui a guastarsi sia un singolo nodo (server del sistema). In questo caso andremo a parlare di **server failure**. Figura 40 mostra un'illustrazione di questo tipo di guasto.

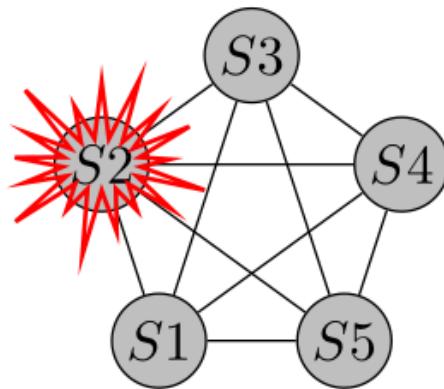


Figura 40: Illustrazione di un guasto su un nodo singolo in un sistema distribuito.

Un'altra tipologia di guasto è quella di **network failure**, in cui la comunicazione tra nodi viene interrotta. Questo può essere causato da problemi di rete, congestione o guasti hardware. Figura 41 mostra un'illustrazione di questo tipo di guasto.

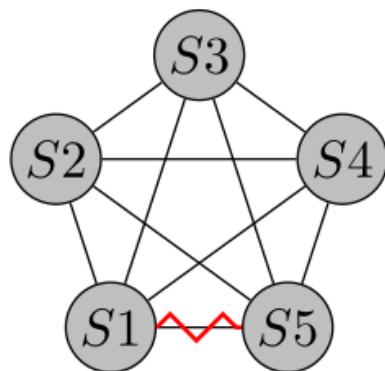


Figura 41: Illustrazione di un guasto di rete in un sistema distribuito.

Nell'eventualità in cui si verifichino **guasti multipli**, potremmo trovarci in una condizione chiamata **network partitioning**, in cui la rete di comunicazione tra i server risulta divisa in più segmenti che non possono comunicare tra di loro. Figura 42 mostra un'illustrazione di questo tipo di guasto.

Osservazione

È importante considerare che un segmento di partizionamento potrebbe essere composto anche da un singolo nodo, nel caso in cui questo risulti isolato dal resto della rete.

Si noti come spesso e volentieri le situazioni di **network partitioning** siano estremamente difficili da distinguere rispetto a quelle di **server failure**, in quanto da un nodo potrebbe sembrare che un altro nodo sia semplicemente non raggiungibile.

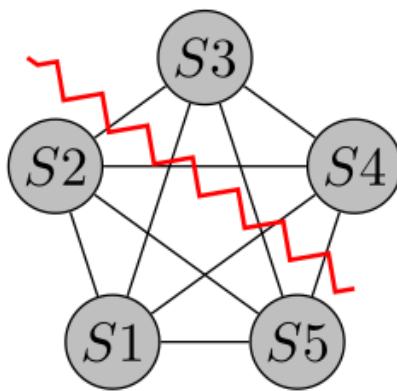


Figura 42: Illustrazione di un partizionamento di rete in un sistema distribuito.

6.3. Protocolli Epidemici

6.3.1. Background

Uno dei punti cruciali all'interno di un sistema distribuito è la **propagazione** delle **informazioni**. In un sistema distribuito, si tratta di un compito complesso, per due principali motivi:

- *Perdita di messaggi*: in una rete distribuita, i messaggi possono essere persi a causa di guasti di rete, congestione o altri problemi, e maggiore il numero di nodi e connessioni, maggiore è la probabilità che questo accada.
- *Forte agitazione dei nodi*: in un sistema distribuito, i nodi possono entrare e uscire dalla rete in qualsiasi momento, rendendo difficile mantenere una visione coerente dello stato del sistema.

Una delle tecniche più interessanti per mitigare questi problemi è l'utilizzo di **protocolli epidemici**, *protocols peer to peer* ispirati al modo in cui le infezioni o il “gossip” si diffondono in una popolazione. Per capire il motivo dietro al quale utilizziamo una modalità peer to peer, consideriamo il caso in cui usassimo un modello **point to point**: per utilizzare questa strategia ogni nodo dovrebbe quantomeno essere a conoscenza dell'esistenza di tutti gli altri nodi. Per garantire questo avremmo due possibilità:

- Ogni nodo mantiene una lista di tutti gli altri nodi: questo approccio non scala bene, in quanto ogni volta che un nodo entra o esce dalla rete, tutti gli altri nodi devono essere aggiornati.

- Utilizzare un nodo centrale per mantenere la lista dei nodi: questo approccio introduce un singolo punto di fallimento e può diventare un collo di bottiglia.

Il ruolo dei protocolli epidemici è dunque quello di permettere propagare le informazioni ricevute da un nodo a tutti gli altri, nel modo più efficiente possibile, e con la massima affidabilità possibile. All'interno di un protocollo epidemico, ogni nodo potrebbe trovarsi in uno dei seguenti stati:

- **Infetto**: il nodo ha ricevuto l'informazione e la sta *propagando ad altri nodi*
- **Suscettibile**: il nodo non ha ancora ricevuto l'informazione, ma è in grado di riceverla.
- **Rimosso**: il nodo ha ricevuto l'informazione, ma non la propaga più ad altri nodi.+

Il motivo per cui i nodi possono essere infetti o suscettibili appare abbastanza chiaro: senza nodi suscettibili, l'informazione non si potrebbe propagare, e in modo analogo senza nodi infetti, non ci sarebbe nessuno a propagarla. Per quanto riguarda lo stato di **rimosso** invece, questo viene introdotto per evitare che l'informazione venga propagata all'infinito. Per fare questo i nodi infetti possono decidere di diventare rimossi dopo un certo periodo di tempo, o dopo aver propagato l'informazione ad un certo numero di nodi.

6.3.2. Varianti di Protocolli Epidemici

Esistono diverse varianti di protocolli epidemici, ognuna con le proprie caratteristiche e vantaggi. Di seguito andiamo a presentarne alcune.

Variante Anti-Entropica

Un approccio di tipo **anti-entropico** è tra i più semplici che si possono applicare. Ogni nodo in questo protocollo può essere soltanto **infetto** o **suscettibile**. I nodi infetti *periodicamente* propagano l'informazione ai loro nodi vicini suscettibili. Questo processo continua fino a quando tutti i nodi sono stati infettati.

Il problema maggiore di questo approccio consiste nell'**eccessivo utilizzo di banda** e che ad ogni round, ogni nodo controlla che i suoi vicini siano suscettibili a nuove informazioni.

Approccio Rumor Spreading

La propagazione delle informazioni viene avviata solamente nel momento in cui un nodo riceve **nuova informazione** o può essere attivata in maniera **periodica**. La differenza rispetto a prima è che in questo caso le dinamiche di propagazione sono differenti:

- La quantità di nodi infetti viene fatta decrescere nel tempo mano a mano che il numero di nodi **rimossi** aumenta
- Ogni server può passare da **infetto** a **rimosso** sulla base di alcune euristiche, per esempio con una certa probabilità ad ogni round, o dopo aver inviato l'informazione ad un certo numero di nodi.

6.3.3. Hash Trees

Come già anticipato, è necessario garantire che ogni coppia di nodi possa stabilire quali informazioni sono rispettivamente note ad ognuno. Per questa operazione si potrebbe pensare di applicare una semplice operazione di **differenza insiemistica**. Tuttavia un'operazione di questo tipo potrebbe risultare inefficiente, richiedendo un numero di operazioni pari ad $O(n)$,

dove n è il numero di elementi da confrontare. Per ovviare a questo problema, si può utilizzare una struttura dati chiamata **hash tree** (o **merkle tree**).

Un **hash tree** può essere visto come un **indice gerarchico di hash**. Ogni nodo foglia dell'albero rappresenta un blocco di dati, e ogni nodo interno rappresenta l'hash dei suoi nodi figli. In questo modo, ogni nodo può rappresentare un insieme di dati tramite un singolo hash. Questo permette di confrontare rapidamente grandi insiemi di dati. Figura 43 ne mostra un esempio.

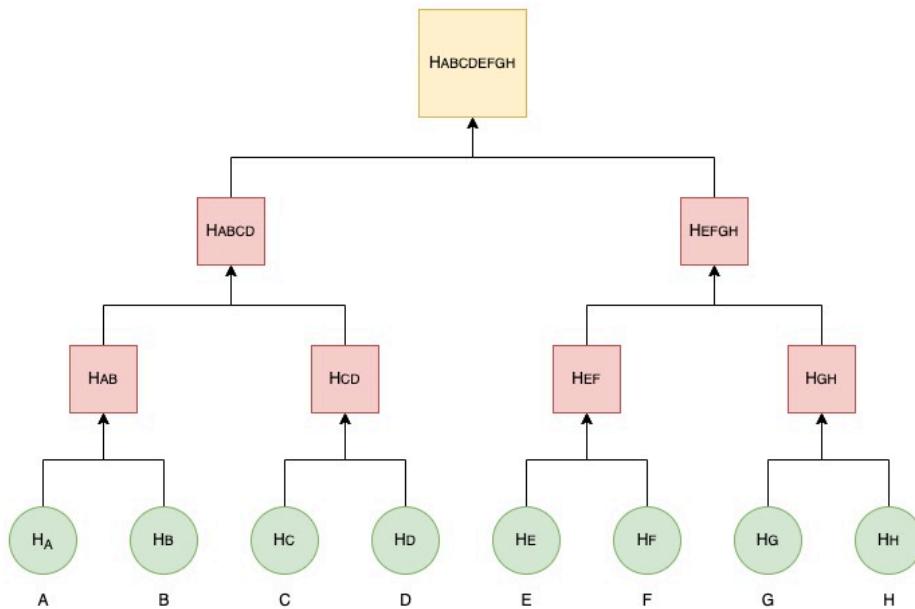


Figura 43: Esempio di Hash Tree con 8 nodi foglia data una lista di messaggi A,B,C,D,E,F,G,H.

Nel caso in cui tutti i messaggi ricevuti da due nodi siano gli stessi, anche gli hash dei nodi radice saranno gli stessi. In caso contrario, i nodi possono scendere lungo l'albero confrontando gli hash dei nodi figli per identificare quali blocchi dati differiscono.

Osservazione

Chiaramente l'utilizzo di una struttura come gli Hash Tree implica che tutti i nodi della rete devono concordare sull'ordinamento dei messaggi che pervengono e su una funzione di hashing univoca.

6.4. Frammentazione («Sharding»)

Una delle tecniche più comuni per la gestione di dati grandi e massivi è quella di ricorrere alla **frammentazione** degli stessi. L'idea è quella di suddividere appunto grandi quantità di dati in **frammenti più piccoli** che possano essere gestiti in maniera più agile. Di seguito presentiamo vantaggi e svantaggi di questa tecnica.

- **Località dei dati:** possiamo delegare ad un nodo locale la gestione e la computazione di operazioni su un frammento di dati comunicando il risultato finale agli altri nodi

- **Riduzione dei costi di comunicazione:** svolgendo le operazioni a livello di singolo frammento in locale, è possibile ridurre la quantità di dati che deve essere trasferita nella rete ✓
- **Performance migliorate:** operazioni su frammenti più piccoli di dati tendono ad essere più veloci rispetto a operazioni su grandi insiemi di dati ✓
- **Indici più efficienti:** gli indici possono essere creati e mantenuti più facilmente su frammenti più piccoli di dati ✓
- Possibilità di applicare **load balancing:** distribuendo i frammenti di dati tra diversi nodi, è possibile bilanciare il carico di lavoro e migliorare le prestazioni complessive del sistema ✓
- **Query più complesse:** le query che coinvolgono più frammenti di dati possono diventare più complesse da gestire e ottimizzare ✗
- **Gestione più complessa:** operazioni di backup e recovery possono diventare più complesse in un sistema frammentato ✗

6.4.1. Allocazione

Ogni volta che tentiamo di memorizzare delle nuove informazioni queste vengono suddivise in **unità di allocazione** e serve decidere quanti e quali nodi dovranno essere impiegati per memorizzare i frammenti. Un'altra tematica importante è quella del **load balancing**: se tutti i frammenti andassero a finire su un singolo nodo, questo diventerebbe un collo di bottiglia per l'intero sistema. Abbiamo diverse possibilità per gestire l'allocazione:

- **Range-based allocation:** supponiamo che i nodi abbiano un identificativo numerico. In questo caso possiamo decidere di allocare i frammenti in base a degli intervalli di identificativi. Per esempio, il nodo 1 potrebbe essere responsabile per i frammenti con ID da 0 a 99, il nodo 2 per quelli da 100 a 199, e così via. Questo approccio è semplice da implementare, ma può portare a squilibri se la distribuzione dei dati non è uniforme.

Esempio: Distribuzione uniforme vs. non-uniforme

sEmemorizziamo date di nascita con solo giorno e mese, la distribuzione sarà pressoché uniforme, mentre anando a memorizzare anche gli anni questo sarà diverso dal momento che alcuni anni sono più rappresentati di altri

- **Cost-based allocation:** il nodo sul quale verranno allocati i frammenti viene scelto in base al costo stimato di accesso ai dati. Questo approccio può essere più complesso da implementare, ma può portare a una migliore distribuzione del carico di lavoro.
- **Hash-based allocation:** utilizza una funzione di hash per mappare i frammenti ai nodi. Si tratta di una tecnica molto simile a quella “range-based” con il vantaggio di distribuire i frammenti più uniformemente tramite le funzioni di hash.

Per quanto la **hashed based allocation** sembri essere una tecnica molto valida, questa presenta due importanti criticità:

- necessità di una conoscenza a priori del numero di nodi nel sistema

- nel momento in cui un nodo subisse dei guasti, molti hashcodes verrebbero invalidati, richiedendo una riallocazione massiva dei frammenti

Consistent Hashing

Una tecnica molto interessante per ovviare ai problemi della **hash-based allocation** è quella del **consistent hashing**. In questo approccio la funzione di hash viene calcolata sia sui **nodi** che sui **frammenti di dati**.

Tutti i valori di hash vengono mappati su un **hash ring**, un intervallo circolare di valori di hash. Ogni frammento di dati viene allocato al nodo il cui valore di hash è più vicino al valore di hash del frammento, procedendo in senso orario lungo l'anello. Figura 44 ne mostra un esempio.

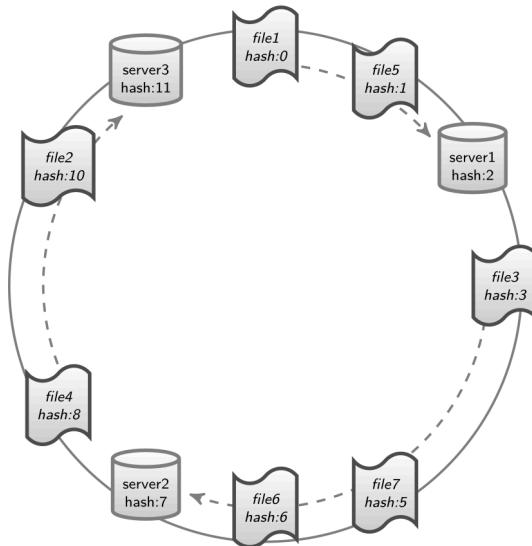


Figura 44: Esempio di Consistent Hashing con 3 nodi e 6 frammenti di dati.

Nel caso di **rimozione di un nodo**, tutti i frammenti del nodo vengono copiati nel nodo successivo nell'anello. Questa situazione è chiaramente illustrata in Figura 45.

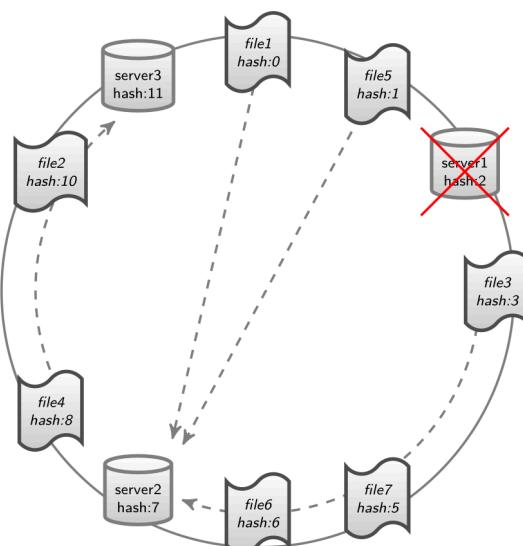


Figura 45: Esempio di Consistent Hashing con rimozione di un nodo.

In maniera abbastanza simile, nel caso in cui si proceda con l'**aggiunta di un nodo**, andremo a considerare il nodo precedente nell'anello e riallocheremo tutti i frammenti che ora ricadono sotto la responsabilità del nuovo nodo. Figura 46 illustra questa situazione.

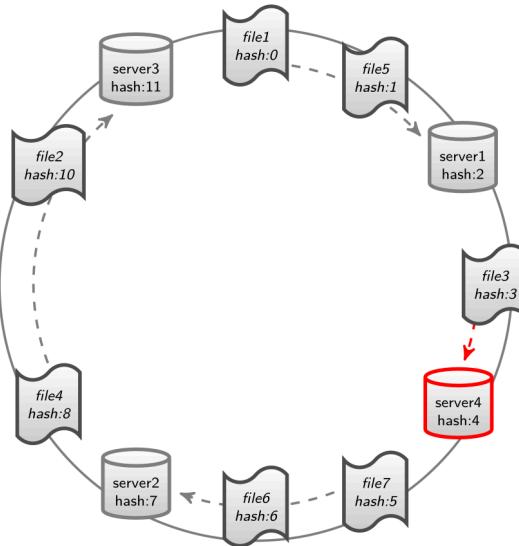


Figura 46: Esempio di Consistent Hashing con aggiunta di un nodo.

Ciò che manca ora è capire come andare a gestire lo stato di questo anello. Supponiamo di avere un client che prova a connettersi in scrittura al nostro sistema distribuito. Per capire in quale nodo scrivere i dati dovremo:

- Calcolare l'hash di un frammento di dati, contattare un nodo casuale in scrittura. A questo punto abbiamo due possibilità:
- Il nodo ha conoscenza dell'intero anello: in questo caso può calcolare direttamente il nodo responsabile per il frammento e inoltrare la richiesta.
- Il nodo può provare ad inoltrare la richiesta al nodo successivo nell'anello, che a sua volta può fare lo stesso fino a quando la richiesta non raggiunge il nodo responsabile.

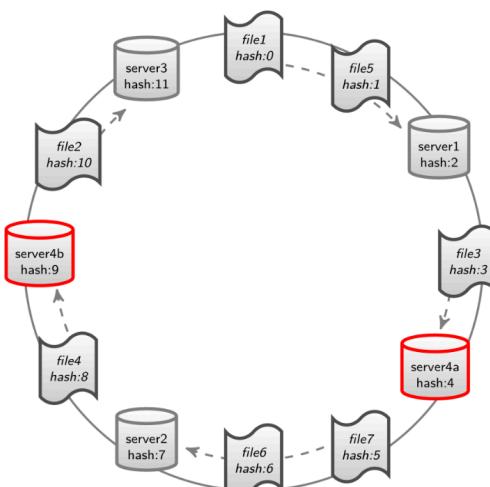


Figura 47: Illustrazione dell'utilizzo di nodi virtuali nel Consistent Hashing.

Evidentemente la seconda opzione è quella che potrebbe richiedere meno sforzo di mantenimento dello stato dell'anello, ma potrebbe richiedere più tempo per raggiungere il nodo responsabile. Per mantenere lo stato dell'anello, l'unica informazione necessaria in ogni nodo è l'identificativo del nodo successivo nell'anello.

Spesso e volentieri, non è verosimile che all'interno di un sistema distribuito tutti i nodi abbiano la stessa capacità computazionale e di storage. In una situazione tale, potremmo non voler distribuire i frammenti in maniera uniforme, andando a prediligere nodi con maggior capacità. Per risolvere questo requisito, possiamo *simulare la presenza di nodi aggiuntivi* gestiti da uno stesso server, chiamati **nodi virtuali**. Utilizzando questo approccio possiamo assegnare più nodi virtuali a server con maggior capacità, e meno nodi virtuali a server con minore capacità. In questo modo, i frammenti di dati verranno distribuiti in maniera proporzionale alla capacità di ogni server. Figura 47 illustra questo concetto.

6.5. Replicazione dei Dati

Oltre a frammentare i dati per rendere le computazioni più efficienti e sostenibili, un'altra tecnica molto comune per la gestione di dati distribuiti è quella della **replicazione**. In questo contesto ci sono due termini che sono importanti da conoscere:

- Le copie dei dati che vengono effettuate sono chiamate **repliche**
- Il numero di nodi su cui le repliche sono memorizzate è chiamato **replication factor**

L'impiego di questa tecnica serve sostanzialmente a due scopi:

- **Affidabilità e disponibilità**: avendo più copie dei dati distribuiti su diversi nodi, il sistema può continuare a funzionare anche in caso di guasti di uno o più nodi.
- **Minor Latenza**: le repliche possono essere utilizzate per effettuare load balancing, e parallelizzazione delle letture, riducendo la latenza di accesso ai dati.

Ovviamente la replicazione dei dati introduce anche delle sfide, tra cui:

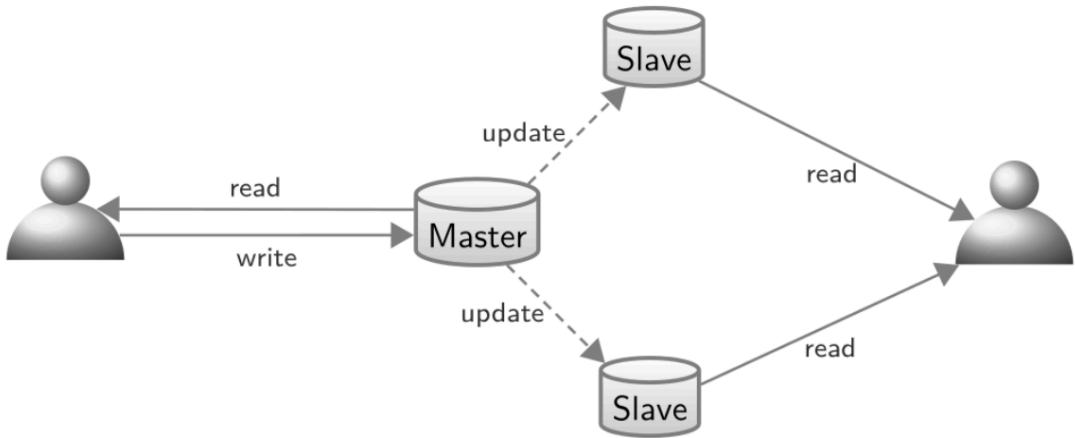
- **Coerenza dei dati**: mantenere tutte le repliche aggiornate può essere complesso
- **Concorrenza**: gestire accessi concorrenti ai dati replicati può portare a conflitti

6.5.1. Replicazione Master-Slave

Si tratta di una delle architetture di replicazione più semplici. In questo modello, un nodo viene designato come **master** mentre gli altri nodi vengono scelti come **slave**. Di seguito andiamo ad elencarne le peculiarità:

- Tutte le operazioni di **scrittura e aggiornamento** sono svolte sul nodo **master**
- Dopo una certa quantità di tempo, il master propaga le modifiche agli **slave**
- Le operazioni di **lettura** possono essere svolte su qualsiasi nodo, sia master che slave

Chiaramente, questo modello presenta un grosso svantaggio: il nodo master rappresenta un **single point of failure**. In caso di guasto del master, il sistema non può più accettare operazioni di scrittura fino a quando un nuovo master non viene eletto. L'immagine presentata di seguito da una rappresentazione schematica del funzionamento di questa architettura.



6.5.2. Replicazione Multi-record

Questo approccio si ispira al modello master-slave introducendo alcune modifiche. Sostanzialmente ogni nodo nel sistema può agire sia come **master** per certe informazioni, sia come **slave** per altre.

Ogni nodo può accettare operazioni in **scrittura** per dati di sua competenza, e propagare le modifiche agli altri nodi. Le operazioni di **lettura** possono essere svolte su qualsiasi nodo. La figura mostrata di seguito illustra questo concetto.

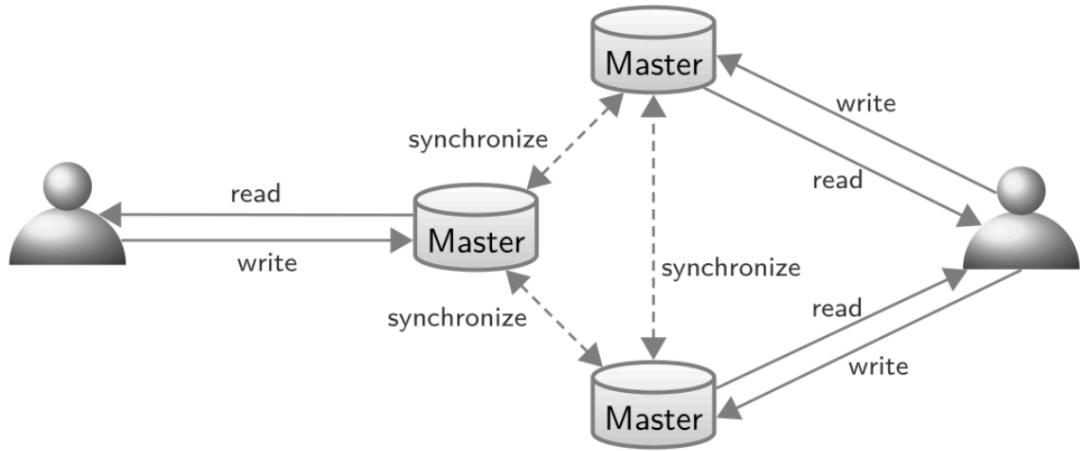
Per andare a sincronizzare le modifiche tra i vari nodi, abbiamo a disposizione due possibili strategie:

- **Replicazione immediata**: ogni volta che un nodo master viene modificato, queste modifiche sono immediatamente propagate agli altri nodi. Questo approccio garantisce una maggiore coerenza, ma può introdurre latenza nelle operazioni di scrittura.
- **Replicazione differita**: le modifiche vengono propagate agli altri nodi dopo un certo intervallo di tempo o quando viene raggiunta una certa soglia di modifiche. Questo approccio può migliorare le prestazioni, ma può portare a situazioni di incoerenza temporanea tra i nodi.



6.5.3. Replicazione Multi-Master

Questo approccio, detto anche «update-anywhere», permette a qualsiasi nodo di accettare operazioni di scrittura e aggiornamento. Si tratta di un approccio che consente **maggior disponibilità e tolleranza ai guasti**. Permette di avere **tempi di risposta migliorati**. Tuttavia, questo modello introduce delle importanti sfide legate alla **coerenza dei dati**: spesso si rende infatti necessario implementare meccanismi di risoluzione dei conflitti per gestire situazioni in cui più nodi tentano di aggiornare gli stessi dati contemporaneamente.



L'immagine presentata sopra illustra il funzionamento di questa architettura. Ogni nodo può accettare operazioni di scrittura e aggiornamento, e le modifiche vengono propagate agli altri nodi. In caso di conflitti, il sistema deve essere in grado di risolverli in modo coerente.

6.5.4. Guasti e Recovery

Avendo discusso la replicazione dei dati, il prossimo passaggio è quello di andare a comprendere come i guasti vengano gestiti in un'architettura distribuita. Ipotizziamo di avere un **replication factor** di 2. Nel caso in cui uno dei due server subisse un guasto, o fosse momentaneamente non raggiungibile, l'altro server dovrebbe essere in grado di continuare a servire le richieste di lettura e scrittura. Nel momento poi in cui il server guasto riprendesse a funzionare, sarà necessaria una **sincronizzazione** con l'altro server per portare il sistema in uno stato coerente. Questa situazione è illustrata in Figura 48.

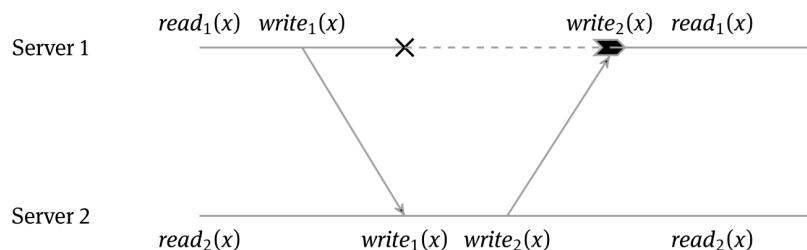


Figura 48: Illustrazione di un guasto di singolo server con replication factor = 2.

Nel caso però in cui, sempre con un **replication factor** di 2, se il secondo server fallisse prima che il primo possa tornare disponibile, tutte le scritture accettate dal secondo server non saranno visibili dal primo. Questo scenario è illustrato in Figura 49.

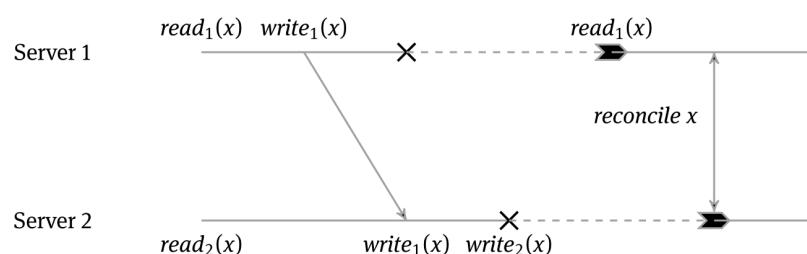


Figura 49: Illustrazione di un guasto di entrambi i server con replication factor = 2.

In questo caso, nel momento in cui entrambi i server torneranno funzionanti, tutte le richieste di scrittura accettate in maniera indipendente dai serer andranno **riconciliate**. Questa situazione ci mette davanti ad un quesito fondamentale, legato a *quale sia il corretto replication factor* da utilizzare. Tendenzialmente una convenzione accettata abbastanza in generale è quella di utilizzare un replication factor pari a 3.

Di seguito andiamo a presentare due tecniche molto comuni per la gestione dei guasti e della successiva riconciliazione dei dati.

Hinted Handoff

Questa tecnica viene utilizzata per gestire i guasti temporanei dei nodi in un sistema distribuito. Nel seguente elenco puntato andiamo a mostrarne il flusso operativo:

- nel momento in cui una replica non è disponibile, le richieste in scrittura vengono **delegate** ad un altro nodo disponibile
- il nodo che riceve le richieste di scrittura riceve un **hint** che indica quale nodo era il destinatario originale della scrittura, in modo tale da poter inoltrare la scrittura non appena il nodo originale torni disponibile

Read Repair

Nell'utilizzo di questa tecnica un *coordinatore* manda una serie di richieste di lettura alle repliche da lui conosciute. Nel momento in cui i nodi rispondono al coordinatore, questo effettua un **majority vote** per capire cosa rispondere al client.

Oltre a fornere risposte il più coerenti possibili al client, il coordinatore si occupa di inviare istruzioni alle repliche che hanno risposto con dati incoerenti in modo da **sincronizzare lo stato** del sistema.

Osservazione

Si noti come, nel caso in cui utilizzassimo un singolo nodo coordinatore, andremmo ad introdurre un **single point of failure**. Per ovviare a questo problema non viene mai scelto un nodo coordinatore fisso, ma questo viene eletto secondo un protocollo di **leader election** ogni volta che un client invia una richiesta.

6.6. Gestione della Concorrenza

Come già menzionato in precedenza, uno dei principali problemi all'interno di un sistema distribuito è quello della gestione della **concorrenza**. Potremmo infatti trovarci nella situazione in cui più componenti tentino di lavorare sugli stessi dati in contemporanea. In questi casi, è fondamentale garantire che tutti i nodi rimangano sincronizzati e l'operazione venga conclusa in maniera coerente.

Commit a 2 Fasi

Il **Two-Phase Commit (2PC)** è un protocollo utilizzato per garantire che le **transazioni distribuite** vengano completate in modo atomico, anche in presenza di guasti. L'idea è che tutti gli agenti devono essere d'accordo sulla finalizzazione di una trasazione. Il protocollo si

divide in una fase di **voting** e in una fase di **decisione**. Anche in questo caso avremo bisogno di un nodo che funga da **coordinatore**. Di seguito andiamo ad elencare alcune altre specificità:

- Il fallimento di un singolo “agente” implica che l’intera transazione venga abortita
- Lo stato che si interpone tra le due fasi è detto “**doubt-phase**”, dato che gli agenti non sanno se la decisione del coordinatore sarà di committare o abortire la transazione
- In caso di guasto del coordinatore, gli agenti rimangono bloccati senza poter committare o abortire

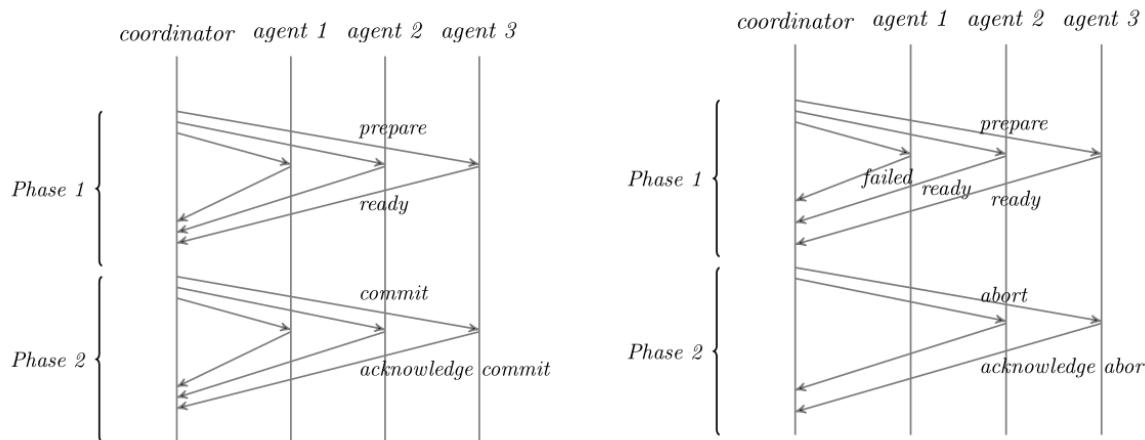


Figura 50: Illustrazione del Two-Phase Commit: a sinistra un commit riuscito, a destra un commit abortito.

6.7. Proprietà dei Database Distribuiti

Per concludere questa panoramica sulle basi di dati distribuite, andiamo a presentare le **proprietà** che queste devono / possono garantire.

Consistenza

Quando un nodo modifica un valore in una replica, il valore dovrebbe essere modificato in tutte le repliche; o se non immediatamente, almeno prima che ci sia la necessità di leggere un certo dato. Si tratta di un processo particolarmente **costoso** nel caso in cui venga utilizzato un **replication factor grande**. Un modo per implementarlo potrebbe consistere nel mantenere bloccate in lettura tutte le repliche fino a quando tutte non siano state aggiornate.

Disponibilità

In un sistema distribuito, è fondamentale che ogni richiesta di lettura o scrittura riceva una risposta anche in presenza di guasti. In particolare vengono tenute in considerazione due *metriche*: **efficienza** in risposta alle query e **bassi tempi di risposta**.

Tolleranza alle Partizioni

Un sistema distribuito deve essere in grado di continuare a funzionare anche in presenza di *partizioni di rete*. Questo significa che anche se alcuni nodi non possono comunicare tra di loro, il sistema deve essere in grado di accettare e processare le richieste.

6.7.1. Congettura di Brewer

Dopo aver definito le proprietà, in questa sezione andiamo a presentare la **congettura di Brewer** che stabilisce una sorta di relazione tra queste proprietà.

Teorema 6.1 (Congettura CAP di Brewer)

Date le proprietà sopra elencate (*consistency, availability, partition tolerance*), in un sistema distribuito è possibile garantire **al massimo due** di queste proprietà contemporaneamente.

♡

In sostanza, in base ai requisiti del sistema che abbiamo la necessità di implementare, andremo a scegliere due delle proprietà che vogliamo andare a garantire. Per esempio, in un sistema in cui è richiesta *alta disponibilità*, è sarà necessario sacrificare la consistenza a favore di una maggiore tolleranza alle partizioni.

Sempre in merito alla consistenza, possiamo distinguere tra due principali modelli:

- **Consistenza forte**: dopo un aggiornamento, ogni lettura successiva restituirà il valore aggiornato e corretto
- **Consistenza debole**: il sistema non garantisce che dopo un aggiornamento le letture seguenti restituiscano il valore aggiornato immediatamente.

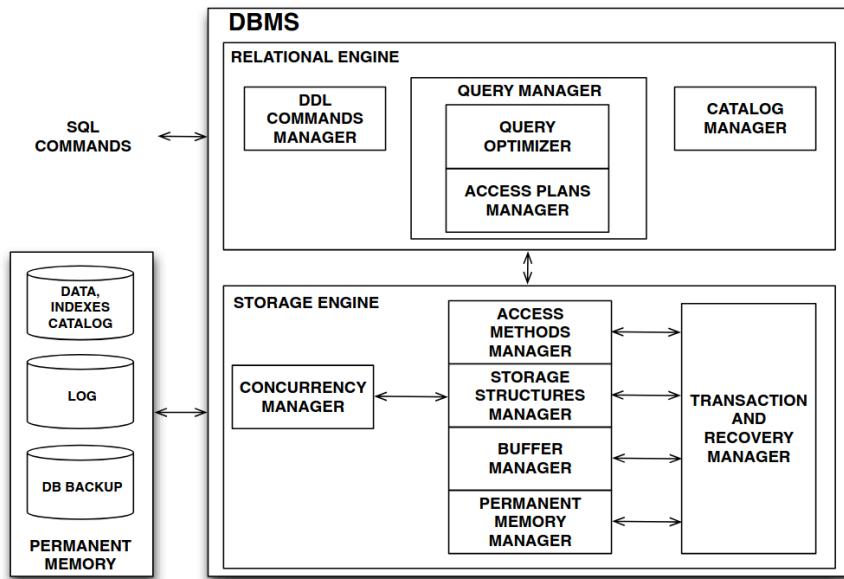
Proprio in merito a questa forma *rilassata* di consistenza, possiamo introdurre due concetti. Il primo è quello di **finestra di inconsistenza**, che rappresenta un intervallo di tempo tra un aggiornamento il momento in cui tutte le repliche non sono ancora state aggiornate. Durante questa finestra, le letture potrebbero restituire valori obsoleti.

Dato il concetto di finestra di inconsistenza, possiamo introdurre la nozione di **eventual consistency** che garantisce che, in *assenza di aggiornamenti*, tutte le repliche convergeranno verso lo stesso valore dopo un certo periodo di tempo, tale periodo è appunto la finestra di inconsistenza.

Uno dei metodi utilizzato per cercare di ottenere consistenza in un sistema distribuito è quello di utilizzare il concetto di **quorum**. In questo approccio, per ogni operazione di lettura o scrittura viene richiesto un numero minimo di repliche per completare l'operazione. Questo concetto potrà essere meglio approfondito all'interno del corso di sistemi distribuiti.

7. Architetture per Basi di Dati

Questo capitolo dà inizio alla seconda parte del corso. Tutto ciò che tratteremo da questo momento in poi supporrà che il punto di partenza sia quello di un **java relational system**. La struttura di base di tale elemento è quella che è già stata presentata in Figura 10, che per completezza viene di seguito riportata.



La scelta di considerare questa architettura viene fatta per semplicità, e per la maggiore familiarità che si ha tipicamente con una base di dati relazionale. Tuttavia, è importante considerare che la maggioranza delle famiglie di database già presentate condivide la gran parte delle componenti con questa architettura.

Come già menzionato l'architettura presentata può essere rappresentata in **due blocchi**

- un blocco che presenta delle componenti specifiche alla famiglia di database che abbiamo scelto di utilizzare, che in questo caso è il **relational engine**
- un blocco che è deputato alla gestione della memorizzazione e il recupero dei dati in memoria principale, a cui ci si riferisce tipicamente con il nome di **storage engine**

Questo schema a blocchi non è sempre presente, esistono casi in cui lo storage engine viene fuso alla componente che normalmente dovrebbe essere più specifica al tipo di dati che vogliamo rappresentare nella nostra base di dati.

In linea di massima però relational (o document database, ...) e storage engine sono tra loro *indipendenti*. Questo maggiore livello di astrazione ci consente di trattare in maniera separata il modo in cui vengono rappresentati a livello logico i dati dal modo in cui questi vengono effettivamente trattati all'interno della memoria secondaria.

Nel corso del capitolo procederemo a presentare le varie componenti seguendo un approccio **bottom-up**, partendo dunque dallo storage engine. Dal momento che non si tratta di componenti fondamentali per il funzionamento di una base di dati, eviteremo di trattare, per ora, il gestore delle transazioni, del recovery e della concorrenza. Andremo dunque a concentrarci sulla colonna centrale dello storage engine.

Vedremo inoltre come questo stack di componenti sia molto simile nel concetto allo stack ISO/OSI, dove ogni componente fornisce servizi per il componente di livello superiore e ne utilizza dal livello inferiore, fino ad arrivare al livello più basso possibile.

7.1. Persistent Memory Manager

Come già menzionato in qualche capitolo precedente il principale ruolo del gestore della memoria persistente è quello di fornire un'**astrazione linearmente indirizzabile** della memoria, nascondendo ai livelli superiori l'effettiva struttura della memoria.

È importante andare a vedere come la memoria secondaria sia strutturata, questa informazione sarà estremamente utile per comprendere anche le sezioni successive. Possiamo dire in generale che la memoria permanente di una DBMS è organizzata a livello logico come un insieme di “database”, ognuno di questi si può vedere come un **insieme di file**, che a loro volta sono organizzati in **pagine** linearmente indirizzabili

Tipicamente vengono utilizzati più **livelli di memoria** in maniera molto simile a quanto accade all'interno di un normale calcolatore. A taglie di memoria più grande, corrisponderà maggior latenza, mentre man mano che si diminuisce la capacità si va a migliorare il tempo di accesso ai dati. Tipicamente i livelli che troviamo in ambienti production scale:

- *glacier storage*: utilizzati per storage di dati a lunghissimo termine
- *data lakes*: utilizzati per memorizzare oggetti usati più frequentemente
- *cloud storage / network drives*: basati su file system distribuiti remotamente
- *drive locali*: dispositivi locali tipicamente accessibili all'interno di una stessa rete
- *gerarchia di memoria del calcolatore*

Ciò che differenzia i vari tipi di memoria è che ad un certo punto della gerarchia esiste una sorta di “barriera” dalla quale in poi tutto quanto ciò che memorizziamo è volatile. Tipicamente questa soglia è costituita dalla **memoria principale**.

7.1.1. Geometria di un Hard Disk

Per quanto possa sembrare poco attuale parlare della geometria di un disco magnetico in questo periodo storico, è importante sottolineare che questo tipo di supporto ha una caratteristica molto conveniente rispetto ad un SSD. Oltre al costo che è indubbiamente minore, ciò che è più importante è il fatto che questo tipo di supporti **non degrada** nel tempo.

È noto infatti che tecnologie come gli SSD sono altamente soggette ad **isteresi**, quel fenomeno per cui lo stato di una porzione della memoria non è più dipendente dai dati memorizzati in un certo momento, ma anche dalla storia di tutti i dati memorizzati in quella porzione di memoria. Questo rende, dopo una certa quantità di tempo, un disco a stato solido inutilizzabile. Sebbene all'interno di un personal computer questo fenomeno possa essere trascurabile, in una base di dati, dove si rende necessario scrivere e leggere dati in maniera intensiva, questo processo potrebbe notevolmente amplificarsi.

All'interno di un supporto magnetico possiamo identificare le seguenti componenti:

- **Tracce**: percorsi circolari su un disco
- Ogni traccia è divisa in **settori** che vanno da 512 a 4KB

- Ogni traccia è composta da un numero variabile di settori, tra i 500 e i 1000
- Un **cilindro** è composto da un insieme di tracce che si trovano nella stessa posizione relativamente alla testina.

Su tutta la superficie del disco abbiamo circa 100k tracce. Per quanto tutte le tracce abbiano una dimensione fisica differente, queste hanno tutte la **stessa capacità di memorizzazione**. Tracce di dimensione minore conterranno informazione in maniera più densa di altre tracce a dimensionalità maggiore.

Ogni traccia è divisa **logicamente** in **blocchi** di dimensione fissa, che rappresentano i blocchi di trasferimento tra i vari livelli di memoria.

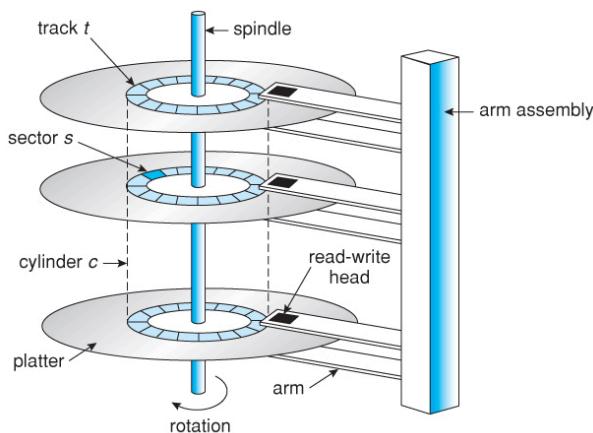


Figura 51: Rappresentazione schematica di un disco magnetico

7.1.2. Lettura da Disco Magnetico

Nel momento in cui andiamo a leggere dei dati da un disco magnetico dobbiamo tenere in considerazione i seguenti aspetti:

- è necessario trasformare l'indirizzo lineare in un **indirizzo fisico**
- una volta trovato il l'indirizzo fisico *muoviamo la testina sul cilindro* di competenza
- una volta trovato il cilindro, andiamo a muovere la testina per farla finire sul **settore** corretto della traccia di interesse

Tutte queste operazioni hanno un costo che possiamo rappresentare tramite la seguente formula:

$$\text{Block Access Time} = t_s + t_r + t_b \quad (4)$$

dove:

- t_s sta per il **seek time**, ovvero il tempo necessario per muovere la testina sul cilindro corretto (3.5/4 ms in media)
- t_r sta per il **rotational latency**, ovvero il tempo necessario per far arrivare il settore corretto sotto la testina (2 ms in media)
- t_b sta per il **transfer time**, ovvero il tempo necessario per trasferire i dati dal disco alla memoria principale (0.1 ms per 4KB circa)

Evidentemente il tempo di trasferimento di un blocco dati è dominato da latenza e seek time. Per questo motivo, invece che andare a leggere un singolo blocco alla volta, è preferibile

leggere più blocchi in sequenza (**pagine**), in modo da minimizzare il numero di seek e rotazioni. Questo fenomeno rende molto importante il concetto di **località spaziale**:

- per trasferire un blocco il tempo necessario è $t_s + t_r + t_b$
- per trasferire n blocchi in sequenza il tempo necessario è $t_s + t_r + n \cdot t_b$

7.2. Buffer Manager

Come già anticipato, l'unità di misura per il trasferimento dei dati all'interno dei supporti di memorizzazione è quella delle **pagine**. Il compito del gestore del **buffer** è proprio quello di spostare pagine *tra la memoria permanente e quella principale*.

Nel fare questo fornisce una **vista logica** della memoria persistente sotto forma di pagine che possono essere utilizzate in memoria principale. I suoi obiettivi sono i seguenti:

- **limitare** il più possibile **lettture ripetute** di una stessa pagina
- **minimizzare** il numero di **accessi in scrittura**

La porzione di memoria gestita dal buffer manager viene chiamata **buffer pool**. Tipicamente questi buffer possono essere molto grandi e risiedono in memoria principale. Il meccanismo che governa il funzionamento del buffer pool è estremamente simile a quello in uso nei sistemi operativi per la gestione della memoria virtuale o di una cache.

7.2.1. Bozza di Interfaccia del Buffer Manager

Di seguito viene presentato un insieme di operazioni che un buffer manager dovrebbe essere in grado di fornire come interfaccia sulla quale costruire il resto del sistema.

GB_getAndPinPage(pageID) --> Page

Si tratta di un'operazione fondamentale per l'effettiva lettura di una pagina: recupera la pagina con identificativo `pageID` dalla memoria persistente spostando la pagina dalla memoria permanente al buffer. L'operazione di **pinning** consente nel rappresentare l'informazione che la pagina è in uso, non si tratta di un flag, quanto più di un contatore che tiene traccia di quante entità stanno utilizzando la pagina.

GB_setDirty(pageID)

Questa operazione è fondamentale nel momento in cui andiamo a modificare dati all'interno di una pagina gestita nel buffer pool. Segnala che la pagina con identificativo `pageID` è stata modificata in memoria principale e deve essere scritta nuovamente su memoria permanente prima di essere rimossa dal buffer.

GB_unpinPage(pageID)

Quando abbiamo terminato di utilizzare una pagina, possiamo fare in modo che questa, a discrezione del del buffer, possa essere rimossa dalla porzione di memoria di sua competenza. Segnala che la pagina con identificativo `pageID` non è più in uso per una certa risorsa diminuendone il contatore di pinning. Se il contatore di pinning arriva a zero, la pagina può essere rimossa dal buffer pool.

GB_flushPage(pageID)

Questa operazione serve nel momento in cui vogliamo forzare la scrittura di una pagina su memoria permanente. Scrive la pagina con identificativo `pageID` su memoria permanente se questa è stata segnalata come **dirty**.

GB_getNewPage(Field,Type) --> Page

Allocà una nuova pagina all'interno della memoria permanente e la porta in memoria principale. La pagina viene inizializzata in base al tipo di dati che deve contenere.

7.2.2. Buffer Pool

Come già anticipato, il principale componente di competenza del gestore del buffer è il **buffer pool**. Figura 52 presenta una rappresentazione schematica di un buffer pool.

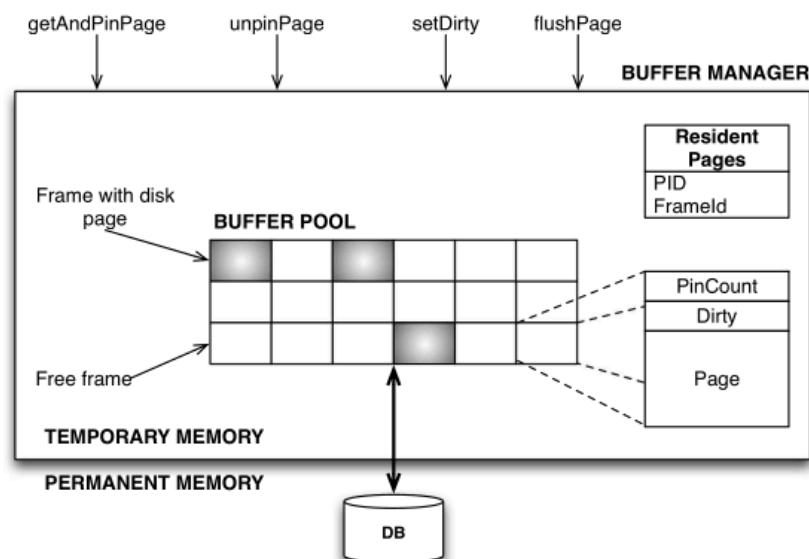


Figura 52: Rappresentazione schematica del buffer pool

Possiamo notare come le funzioni di gestione del buffer siano rese disponibili all'esterno del buffer manager tramite un'interfaccia. Possiamo notare come il buffer pool sia composto da un insieme di **frame** di dimensione fissa, tipicamente della stessa dimensione di una pagina. Ogni frame può contenere una pagina che è stata portata in memoria principale dalla memoria permanente. Ogni frame contiene inoltre delle informazioni di **metadati** che servono al buffer manager per tenere traccia dello stato della pagina contenuta al suo interno:

- un **contatore di pinning** che tiene traccia di quante entità stanno utilizzando la pagina
- un flag **dirty** che segnala se la pagina è stata modificata in memoria principale
- un campo **pageID** che identifica in maniera univoca la pagina contenuta all'interno del frame

Scrittura di Pagine su Memoria Permanente

Oltre che tramite l'API di flushing, è possibile che una pagina venga scritta in memoria permanente nel momento in cui, avendo il contatore di pinning a zero, il buffer manager decide di rimuoverla dal pool per fare spazio ad una nuova richiesta. In questo caso, se la pagina è

segnalata come dirty, questa viene scritta in memoria permanente prima di essere rimossa dal buffer pool.

Out of Memory Error

Nel caso in cui il buffer pool sia pieno e venga richiesta una nuova pagina, il buffer manager deve decidere quale pagina rimuovere per fare spazio alla nuova richiesta. Tipicamente ciò che viene fatto è cercare una pagina con contatore di pinning a zero, e tra queste sceglierne una. Nel momento in cui non fosse presente nessuna pagina con contatore di pinning a zero, il buffer manager non sarebbe in grado di soddisfare la richiesta e verrebbe segnalato un errore del tipo **out of memory**.

7.2.3. Tabella delle Pagine Residenti

Una componente fondamentale per il funzionamento del buffer manager è la **tabella delle pagine residenti** (resident page table). Si tratta di una struttura dati che consente di tenere traccia di quali pagine sono attualmente presenti all'interno del buffer pool.

Mentre sul disco le pagine sono identificate da un indirizzo fisico, il **pageID**, all'interno del buffer pool le pagine potrebbero essere assegnate in maniera *dinamica* a diversi frame. La tabella delle pagine residenti consente di tenere traccia di questa associazione tra **pageID** e frame del buffer pool. Figura 53 presenta una rappresentazione schematica della tabella delle pagine residenti.

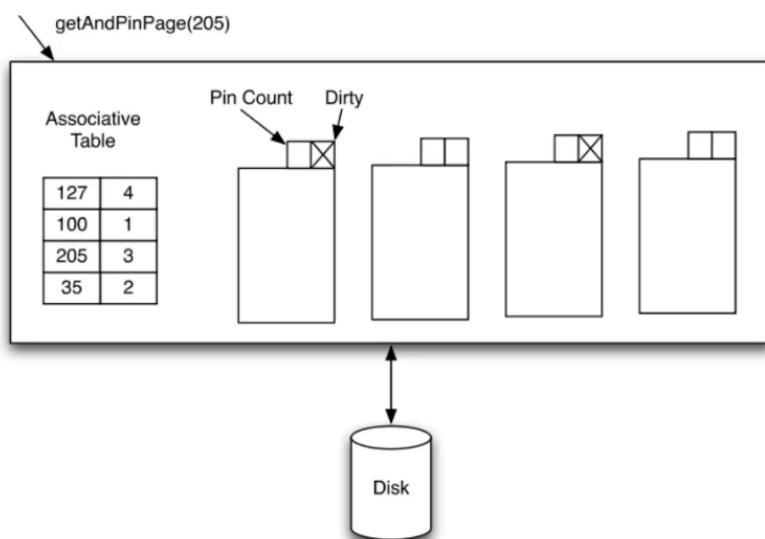


Figura 53: Rappresentazione schematica della tabella delle pagine residenti

Gestione di una richiesta di una pagina

Di seguito andiamo a mostrare il flusso operativo che avviene all'interno del buffer manager nel momento in cui si rende necessario gestire la richiesta di una pagina:

- Se la pagina è già presente nel buffer pool, andiamo semplicemente ad incrementare il pin count ritornandone il riferimento al richiedente
- Se la pagina non è presente procediamo come di seguito:

- Scegliamo una frame libero, o una frame dove il pin count sia zero tramite l'utilizzo di una **politica di sostituzione**. Se tale pin count è zero, e la pagina è dirty, procediamo a scriverla su memoria permanente
- Carichiamo la pagina richiesta dalla memoria permanente alla frame scelta, impostando il pin count a 1 e il flag dirty a false
- Aggiorniamo la tabella delle pagine residenti ritornando al chiamante il riferimento alla pagina appena caricata

Politica di Sostituzione

Nel momento in cui si rende necessario scegliere una pagina da rimuovere dal buffer pool, il buffer manager deve adottare una **politica di sostituzione** per scegliere quale pagina rimuovere.

La possibilità più semplice è quella di utilizzare un approccio **LRU**. Questo approccio è particolarmente adatto alla situazione in cui dobbiamo effettuare un `JOIN`, tuttavia non è detto che sia sempre la scelta ottimale. In alcuni casi **MRU** potrebbe essere una scelta migliore.

Nel caso in cui avessimo degli indici, allora potremmo aver bisogno di applicare politiche di sostituzione più sofisticate, che meglio si adattano al tipo di accesso che stiamo effettuando sui dati. In questo caso avremo però bisogno di utilizzare un **buffer pool separato** da quello normale per gestire le pagine degli indici, in modo da poter applicare politiche di sostituzione differenti.

7.2.4. Struttura di una Pagina di Memoria

Abbiamo già visto come l'unità di trasferimento dei dati tra memoria principale e secondaria sia la **pagina**. Fino a questo momento abbiamo soltanto visto come una pagina sia caratterizzata da un identificativo univoco.

Da un punto di vista **fisico** (all'interno di un file, che ricordiamo essere un insieme di pagine) una pagina è una struttura dati di **dimensione fissa**. Dal punto di vista **logico** (all'interno di un buffer) una pagina è una struttura che contiene le seguenti informazioni:

- **informazioni di servizio**: metadati necessari per la gestione della pagina all'interno del buffer pool
- **record**: i dati veri e propri memorizzati all'interno della pagina

Ora che abbiamo intuito come lo scopo di una pagina è quello di contenere diversi record, è necessario capire come effettuare accesso ai vari record. Abbiamo già citato in capitoli precedenti come ogni record sia fatto da **campi** (fields). Ognuno di questi campi può essere memorizzato in modo diverso:

- **dimensione fissa**: per esempio potremmo allocare un certo numero di byte per memorizzare certi valori. In questo caso l'accesso ai campi è molto semplice, in quanto basterà conoscere l'offset del campo all'interno del record per poterlo recuperare. Tuttavia questo approccio, nel caso di dati con dimensione variabile, potrebbe portare a sprechi di spazio, in quanto potremmo allocare più spazio del necessario
- **separatore speciale**: potremmo scegliere di utilizzare un carattere special per separare i valori all'interno di un record

- utilizzo di un **prefisso** che indica la **lunghezza del campo**

Tipicamente è presente una corrispondenza univoca tra record e pagine: ogni record è contenuto in una sola pagina. Tuttavia, ci sono dei casi, per esempio con dati in formato `BLOB`, in cui un singolo record potrebbe essere più grande della dimensione di una pagina. Per gestire questo caso avremo bisogno di una *sovrastruttura* che andremo a introdurre più avanti nel corso. L'idea è quella di memorizzare in record in un contenitore sufficientemente grande e memorizzare un riferimento a questo contenitore.

Esempio: Memorizzazione di record con campi a dimensione fissa

Si consideri la seguente tabella rappresentante un record con diversi campi ognuno con diverse dimensioni ma memorizzati con delle dimensionalità fisse.

ATTRIBUTO	POSIZIONE	TIPO DEL VALORE	VALORE
Name	1	<code>char(10)</code>	Rossi
StudentCode	2	<code>char(6)</code>	456
City	3	<code>char(2)</code>	MI
BirthYear	4	<code>int(2)</code>	68

In questo caso il numero totale di caratteri utilizzato effettivamente sarà 20. Di seguito mostriamo come sia possibile memorizzarlo secondo i vari approcci presentati sopra:

- *Dimensione fissa*: `Rossi....456...MI68`
- *Separatore speciale*: `Rossi|456|MI|68|`
- *Indice* che indica l'inizio di ogni campo: `(1,6,9,11)Rossi456MI68`

Per fare **riferimento** ad un **record** utilizziamo un identificativo univoco chiamato **record identifier**, che è composto da due campi:

- un **pageID** che identifica la pagina in cui il record è memorizzato
- un **offset** che indica la posizione del record all'interno della pagina

L'utilizzo di questi record identifier è particolarmente utile nel momento in cui andiamo ad utilizzare degli **indici** per velocizzare l'accesso ai dati.

Quello appena presentato, chiamato **riferimento diretto**, non è però il modo migliore per fare riferimento a record. Immaginiamo infatti di star indicizzando un insieme di record che si possono trovare su più pagine, e l'indice punta al riferimento diretto di ogni record. Se andiamo ad eliminare dei record in una pagina potremmo a un certo punto aver bisogno di *deframmentare* la pagina per recuperare spazio. In questo caso, i riferimenti diretti all'interno dell'indice andrebbero a puntare a posizioni errate. Ricalcolare l'indice sarebbe estremamente costoso.

Per questo motivo si preferisce mantenere un livello di **indirezione** tra l'indice e il record vero e proprio. In questo caso l'indice punterà ad un elemento dello **slot array** che sarà una struttura presente a livello di pagina. Sarà poi compito dello slot array tenere traccia del riferimento diretto (che in termini pratici, consiste solamente nell'*offset*) di ogni record. In questo modo, nel momento in cui andiamo a deframmentare la pagina, sarà sufficiente aggiornare lo slot array senza dover toccare l'indice. Figura 54 mostra schematicamente la differenza tra l'approccio indiretto e indiretto.

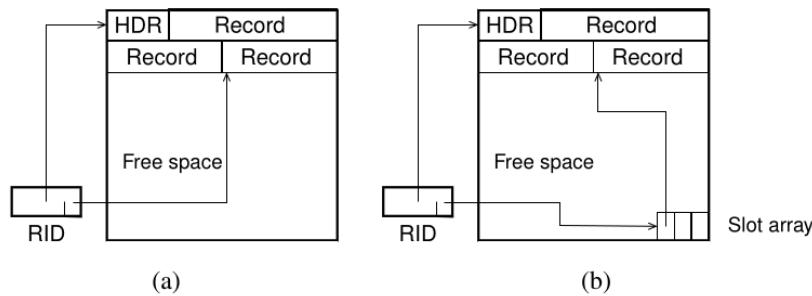


Figura 54: a) Rappresentazione di utilizzo di riferimenti diretti per i record all'interno di una pagina. b) Rappresentazione di utilizzo di uno slot array per i record all'interno di una pagina.

Osservazione

Lo slot array viene posizionato tipicamente alla fine della pagina, in modo da poter crescere verso l'alto. Se andassimo infatti a posizionare questo elemento all'inizio della pagina, ogni elemento aggiunto potrebbe farlo crescere di dimensionalità, comportando una nuova modifica di tutti gli offset.

7.3. Strutture per File Management

Come già noto dalla sezione del permanent memory manager, sappiamo che un **file** si può vedere dal punto di vista logico come un **insieme di pagine**.

Fino a questo momento ci siamo sempre concentrati sulla gestione dei record all'interno di una pagina. Abbiamo imparato come leggere dei record e come andarli a memorizzare all'interno di una pagina, dando per assunto di conoscere la pagina in cui i record sono memorizzati. Nella prossima sezione andremo a concentrarci su come venga gestito lo spazio all'interno di un file, che ricordiamo essere un insieme di pagine:

- come viene **scelto** dove memorizzare un nuovo record
- come **modificare** il contenuto di un **file** per riciclare memoria
- come **compattare** lo spazio all'interno di un **file**

In generale il tema di maggiore importanza è capire come **scegliere** una pagina sufficientemente capiente da poter memorizzare un nuovo record; sempre che questa pagina esista, in caso contrario sarà necessario allocare una nuova pagina all'interno del file.

In questa sezione andremo a vedere diversi approcci per la gestione dell'allocazione dello spazio all'interno di un file: *seriale*, *sequenziale*, *associativo* (per attributi unici o non unici).

Modello di Costo

Prima di andare a vedere i vari modelli di allocazione dello spazio all'interno di un file, è importante definire un **modello di costo** che ci consenta di valutare l'efficacia dei vari approcci. Andremo a tenere in considerazione i seguenti aspetti:

- **spazio utilizzato**: quantità di spazio effettivamente utilizzato per memorizzare i record, specialmente nel caso in cui venga utilizzata memoria ausiliaria
- **performance**: numero di operazioni read/write necessario per eseguire determinate operazioni come inserimento, aggiornamento, cancellazione e ricerca

All'interno di questo modello di costo, considereremo un file R con N_{rec} record di lunghezza L_r e un numero di pagine N_{page} di dimensione D_{page} .

Osservazione

Si noti come tutti i parametri definiti nell'ultimo paragrafo siano **necessari** per poter valutare l'efficacia del nostro modello di allocazione.

In base alle informazioni fornite dal nostro modello di costo, avremo la possibilità di comprendere quale sia il modello di allocazione più adatto alle nostre esigenze. Tipicamente l'**unità di misura** tramite cui esprimiamo il costo è il numero di **accessi a pagina** necessari per eseguire una certa operazione (sia in lettura, sia in scrittura).

7.3.1. Modello Seriale e Sequenziale

È stato scelto di raggruppare questi due approcci per l'organizzazione dei file in quanto condividono molte caratteristiche. La loro peculiarità consiste nel fatto che entrambi gli approcci **non utilizzano strutture dati ausiliarie**, la memoria viene utilizzata solamente per tenere traccia delle pagine che compongono il file.

La differenza sostanziale tra i due approcci consiste nel fatto che il modello **seriale** (*heap file*) non suppone alcun ordinamento dei valori all'interno del file, mentre il modello **sequenziale** suppone che i record siano memorizzati in ordine in base al valore di uno o più attributi.

Organizzazione Seriale

Il modello *heap file* è una tecnica molto semplice ed efficiente in termini della memoria utilizzata. Il problema è che nel caso di file molto grandi potrebbe risultare in performance scadenti. È ottimale nel caso invece in cui i file gestiti siano di piccole dimensioni. Si tratta dell'organizzazione per file che viene utilizzata di default nella maggior parte dei DBMS. Di seguito andiamo a vedere come vengono gestite le varie operazioni sui file:

- **ricerca** di un singolo valore per un attributo: è necessario effettuare una scansione sequenziale fino a quando non viene trovato il valore cercato, o fino a quando la fine della lista non viene raggiunta
- **ricerca** di un **intervallo** di valori: similmente al caso precedente è richiesta una scansione completa. La differenza rispetto a prima è che abbiamo un "stop criteria" differente, dal momento che dobbiamo continuare a leggere fino a quando non superiamo il valore massimo dell'intervallo

- l'**inserimento** di un nuovo record avviene semplicemente aggiungendo il record alla fine dell'ultimo file
- la **cancellazione** e l'**aggiornamento** di un record vengono effettuate tramite un'operazione di ricerca del record e di *riscrittura*.

Costi del Modello Seriale

Di seguito andiamo ad illustrare i vari costi associati all'utilizzo di una architettura di questo genere. In generale, in termini di **spazio** avremo bisogno di una quantità espressa dall'equazione di seguito:

$$N_{\text{page}}(R) = N_{\text{rec}}(R) \cdot \frac{L_r}{D_{\text{page}}} \quad (5)$$

Osservazione

Il valore di L_r è tipicamente esatto, nel caso in cui abbiamo a che fare con valori provenienti da un dominio “statico”, nel caso in cui avessimo valori a lunghezza variabile, il suo sarà una media delle varie dimensioni memorizzate fino ad un certo punto.

Possiamo notare come in Equazione 5 il costo in termini spazio sia unicamente dipendente dall'informazione memorizzata all'interno di un file. Andiamo ora ad osservare il costo in termini di numero accessi alle pagine per le varie operazioni:

- Per quanto riguarda **cancellazione** e **aggiornamento** avremo bisogno di un numero di accessi alle pagine pari a quello necessario per una ricerca più un accesso in scrittura per riscrivere il record: $C_d = C_u = C_s + 1$

Il motivo per cui il costo di una cancellazione di valore è uguale a quello di un aggiornamento è dovuto al fatto che, per evitare di dover spostare tutti i record cancellandone uno, si preferisce sovrascrivere il record o impostare un flag di cancellazione. In questo modo il costo risulta essere identico.

- per l'**inserimento** di un record avremo già conoscenza riguardo a quale sia l'ultima pagina del file, e potremo quindi effettuare una lettura della pagina di interesse e un conseguente inserimento: $C_i = 2$
- per la **ricerca di un singolo valore** per un attributo: $C_s \approx \lceil N_{\text{page}} \frac{R}{2} \rceil$ mediamente nel caso in cui il valore sia presente, in caso contrario avremo $C_s = N_{\text{page}}(R)$
- per la **ricerca di un intervallo** di valori: $C_r = N_{\text{page}}(R)$, non essendo infatti i valori ordinati, si rende necessario scorrere l'intero file per essere sicuri di aver trovato tutti i valori nell'intervallo.

Organizzazione Sequenziale

L'idea alla base dell'organizzazione sequenziale è quella di mantenere i record ordinati rispetto ad una chiave K e memorizzarli in memoria continua. Questo principio consente di abbassare notevolmente i costi di ricerca, sia per un singolo valore, sia per un intervallo di valori.

Lo svantaggio principale di questo approccio è legato al **mantenimento dell'ordinamento**. Questo comporta infatti che ogni inserimento e aggiornamento comporti una riorganizzazione del file, che potrebbe essere estremamente costosa. Nello specifico, le operazioni di aggiornamento sono ancora più complicate, dal momento che possono o non possono cambiare l'organizzazione del file a seconda del valore della chiave K .

Per tutte queste ragioni, l'utilizzo di questo modello di organizzazione, sebbene interessante dal punto di vista teorico, non viene praticamente mai utilizzato nei DBMS moderni. L'unico caso in cui risulti utile è quello in cui il file venga utilizzato in sola lettura, come per esempio nel caso di **data warehousing**.

Ricerca di un singolo valore

Nel momento in cui dobbiamo andare a cercare una determinata chiave k all'interno di un file sequenziale, possiamo sfruttare l'ordinamento di questo. A livello teorico possiamo utilizzare tutti i possibili algoritmi già noti per la ricerca in liste ordinate, come per esempio la **ricerca binaria**.

Osservazione

Si rende necessario tenere in considerazione che i dati vengono trasferiti **una pagina alla volta** (o a blocchi di pagine). Questo porta ci porta ad adattare i nostri algoritmi di ricerca.

Pensiamo ad esempio alla ricerca binaria in cui andiamo normalmente a scegliere un “*pivot*” e sulla base del valore di questo andiamo a decidere se cercare nella metà sinistra o destra della lista di valori. In questo caso non abbiamo più soltanto un “*pivot*”, ma ci servirà utilizzare “**pagine pivot**”:

- Scegliamo una “*pagina pivot*” (tipicamente al centro del file) P_i , con $1 \leq i \leq N_{\text{page}}(R)$
- Prendiamo m_i, M_i come rispettivamente il valore minimo e massimo dell'attributo cercato nella pagina P_i
- Se $m_i \leq k \leq M_i$ allora abbiamo trovato la pagina di nostro interesse e possiamo cercare k al suo interno, in caso contrario se $k < m_i$ allora andiamo a cercare nella metà sinistra del file, altrimenti andiamo a cercare nella metà destra del file.
- Se l'intervallo di pagine da cercare è vuoto, allora il valore non è presente nel file

Osservazione

Si noti come il costo di ricerca di un valore all'interno di una pagina una volta che questa è stata trovata si può considerare **trascurabile**, dal momento che il costo totale (in termini di tempo e latenza) sarà dominato dal numero di accessi alle pagine.

7.3.2. Algoritmi di Ricerca e Costi

Di seguito andiamo a presentare varie possibilità per implementare la ricerca di un singolo valore all'interno di un file sequenziale andandone a valutare i costi associati.

Ricerca Binaria

Come noto da precedenti corsi di algoritmi, la ricerca binaria, che va a scegliere il “**pivot**” al **centro dell'intervallo di ricerca** in cui cercare produce un costo medio dato da:

$$C_s = \log_2(N_{\text{page}}(R))$$

Sebbene in un normale algoritmo di ricerca questo costo sia accettabile se non addirittura buono, in un contesto come il nostro dobbiamo fare di meglio. Infatti nel caso in cui avessimo 400k pagine, il costo medio sarebbe di circa 19 accessi a pagina. Questo costo è ancora troppo elevato per essere accettabile in un DBMS.

Ricerca Interpolata

Un miglioramento rispetto alla ricerca binaria può essere ottenuto tramite l'utilizzo della **ricerca interpolata**. L'idea è quella di scegliere il **pivot** in maniera più intelligente, andando a stimare la posizione del valore cercato all'interno del valore di ricerca.

L'idea di base è che le chiavi siano distribuite in maniera all'incirca **uniforme** nell'intervallo di ricerca. In questo modo possiamo stimare la *probabilità che una chiave sia minore di un certo k* tramite la seguente formula:

$$p_k = \frac{k - k_{\min}}{k_{\max} - k_{\min}}$$

Dato il valore di p_k possiamo stimare la posizione della pagina pivot tramite la seguente formula:

$$P_i = \left\lceil P_1 + p_k \cdot \left(P_{N_{\text{page}}(R)} - P_1 \right) \right\rceil$$

In sostanza p_k , supponendo che le chiavi siano distribuite uniformemente, ci consente di scalare il valore di k all'interno dell'intervallo di ricerca. Utilizzando questa tecnica i costi saranno i seguenti:

- $C_s = O(\log_2 \log_2 N_{\text{page}}(R))$ nel caso “medio”
- $C_s = O(N_{\text{page}}(R))$ nel caso peggiore, tuttavia questo caso è molto raro

Utilizzando ricerca interpolata nel caso di 400k pagine avremo un costo medio di circa 4 accessi a pagina, che è decisamente migliore rispetto alla ricerca binaria.

Altri costi per la ricerca interpolata

Ipotizzando di andare ad utilizzare la ricerca interpolata, andiamo a vedere quali sono gli altri costi associati alle altre operazioni sui file:

- Nel caso di **ricerca di un intervallo** ($k_1 \leq k \leq k_2$), ipotizzando di avere chiavi uniformemente distribuite nell'intervallo $[k_{\min}, k_{\max}]$ avremo quanto segue:

$$f_s = \frac{k_2 - k_1}{k_{\max} - k_{\min}}$$

che corrisponde alla frazione totale delle pagine coperte dall'intervallo di ricerca rispetto al totale, questo elemento prende il nome di **fattore di selettività**. Dato questo fattore di selettività, il costo totale si può esprimere come:

$$\lceil \log_2 N_{\text{page}}(R) \rceil + \lceil f_s \cdot N_{\text{page}}(R) \rceil - 1$$

dove il primo membro della somma corrisponde al **costo per la ricerca della prima pagina** dell'intervallo, mentre il secondo membro corrisponde al **costo per la lettura di tutte le pagine successive** nell'intervallo. Chiaramente andiamo a sottrarre 1 in quanto la **prima pagina è già stata letta**.

- Nel caso di **inserimento**, dobbiamo innanzitutto individuare la pagina corretta in cui inserire il nuovo record. Questo comporta la scansione di tutte le pagine precedenti rispetto a quella di destinazione. Nel caso peggiore, quando tutte le pagine sono completamente occupate, è necessario spostare i record verso le pagine successive. Tuttavia, siccome il costo che ci interessa misurare è quello in termini di **accessi a pagina**, è sufficiente considerare lo spostamento di **un solo record per pagina** verso la pagina successiva, per ciascuna delle pagine che seguono quella di inserimento. Il costo totale dell'operazione di inserimento è quindi dato da:

$$C_i = C_s + N_{\text{page}}(R) + 1$$

dove, il primo termine rappresenta il costo per la **ricerca della pagina di inserimento**, il secondo termine rappresenta il costo per la **scansione delle pagine successive per lo spostamento dei record**, e l'ultimo termine rappresenta il costo per la **scrittura del nuovo record nella pagina di destinazione**.

- Nel caso di **cancellazione**, andiamo a considerare soltanto il caso di cancellazione **logica**, possiamo dire che questo ha costo come nel caso seriale: $C_d = C_s + 1$.

Confronto tra Modello Seriale e Sequenziale

Di seguito andiamo a presentare una tabella riassuntiva che confronta i due modelli di organizzazione per file appena presentati in termini dei costi a loro associati.

Type	Memory	Single value search	Interval search	Insertion	Deletion
Serial	$N_{\text{pag}}(R)$	$\lceil N_{\text{pag}}(R)/2 \rceil$	$N_{\text{pag}}(R)$	2	$C_s + I$
Sequential	$N_{\text{pag}}(R)$	$\lceil \log_2 N_{\text{pag}}(R) \rceil$	$C_s + I + \lceil f_s \cdot N_{\text{pag}}(R) \rceil$	$C_s + I + N_{\text{pag}}(R)$	$C_s + I$

Figura 55: Confronto tra modello seriale e sequenziale

In generale non abbiamo un vincitore assoluto tra i due approcci, ognuno ha i suoi pro e contro:

- il modello **seriale** è ideale nel caso in cui abbiamo bisogno di effettuare tanti **inserimenti**, ma è pessimo nel caso in cui dobbiamo effettuare ricerche molto piccole
- il modello **sequenziale** è ideale nel caso in cui abbiamo bisogno di effettuare tante **ricerche**, ma è pessimo nel caso in cui dobbiamo effettuare tanti inserimenti

7.4. Problemi di Ordinamento

Come già noto, l'ordinamento di dati è uno dei problemi più classici ed importanti nell'ambito dell'informatica. Anche nelle basi di dati si tratta di un'operazione molto comune: spesso infatti capita che sia necessario ordinare i dati in base a certi attributi per poter rispondere a query specifiche.

Altri casi molto comuni in cui è necessario ordinare i dati sono i seguenti:

- unione di dati provenienti da più tabelle
- eliminazione di duplicati
- raggruppamenti rispetto a determinate chiavi
- operazioni insiemistiche (unioni, intersezioni, differenze)

Seppur siano già state viste numerose tecniche di ordinamento in corsi precedenti, all'interno di un DBMS l'algoritmo di ordinamento maggiormente utilizzato utilizzato è **external merge-sort**. Questo algoritmo funziona in due fasi:

- **ordinamento**: crea piccoli ordinamenti detti **run**
- **merge**: fonde più run ordinati in un unico file ordinato

Di seguito andiamo a vedere cosa succede ad ogni passo dell'algoritmo.

Fase di ordinamento

All'interno del **buffer** vengono ordinate B (dimensione del buffer) pagine per volta tramite un algoritmo di **sort interno** e scrive il risultato su disco come un **run** ordinato. Questo processo viene ripetuto fino a quando tutte le pagine del file sono state lette e scritte su disco come run ordinate. Al termine del sorting otteniamo n run dove:

$$n = \left\lceil \frac{N_{\text{page}}(R)}{B} \right\rceil$$

Fase di merge

In questa avviene l'ordinamento vero e proprio. Sappiamo che abbiamo un buffer di B pagine a disposizione, che è uno spazio limitato rispetto a quello effettivamente necessario. Per superare questo limite andremo a considerare $Z = B - 1$ run. Per ognuna di queste run andremo a tenere all'interno del buffer una pagina, riservando l'ultima pagina per scrivere il risultato.

In questo modo, ad ogni passo andremo a scegliere il record più piccolo tra quelli nelle Z pagine, scrivendolo nella pagina di output. Man mano che la pagina di output si riempie questa viene scritta su disco e andrà a dar forma al nuovo file ordinato, nel frattempo le Z pagine vengono riempite progressivamente.

Questo processo verrà ripetuto per $\frac{Z}{n}$ fasi di merge, tutti i risultati prodotti saranno **intermedi** e verranno fusi nuovamente fino a quando non rimarrà un unico file ordinato.

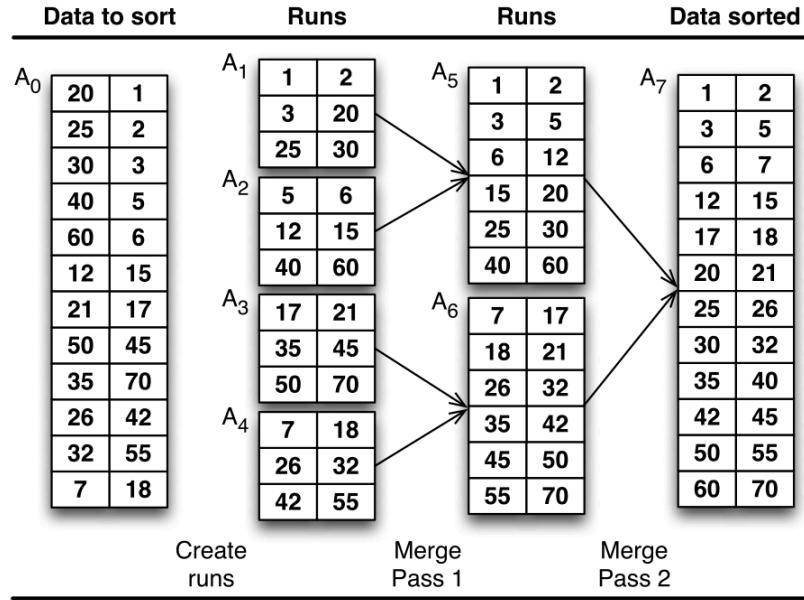


Figura 56: Rappresentazione schematica dell'algoritmo di external merge-sort con dimensione del buffer di 3 pagine, e pagine di dimensione 2

Figura 56 mostra schematicamente il funzionamento di external merge sort. Si noti come dal momento che il buffer può contenere 3 pagine di dimensione 2, allora $Z = 2$. Non a caso il risultato della fase di sort consiste in 4 run ordinate di 3 pagine ciascuna.

Costo di External Merge-Sort

Andiamo ora a vedere quali sono i costi associati all'algoritmo di external merge-sort. In generale questo costo è estremamente dipendente dal numero di volte che i dati vengono spostati tra il buffer e la memoria permanente.

Per quanto riguarda l'algoritmo in sé, sappiamo che questo può essere visto come diviso in un singolo passo di ordinamento e un certo numero k di passi di merge. Indipendentemente dal tipo di operazione effettuata, ogni passo comporta la lettura e la scrittura di tutte le pagine del file. Definiamo quindi il costo di singolo passo $C_{\text{step}} = 2 \cdot N_{\text{page}}(R)$.

Il costo totale dell'algoritmo sarà dunque dato da:

$$C_{\text{tot}} = (1 + k) \cdot C_{\text{step}} = (1 + k) \cdot 2 \cdot N_{\text{page}}(R)$$

A questo punto non resta che andare a investigare il numero di passi di merge k . Sappiamo che il numero di run iniziali S è strettamente legato alla dimensione del buffer B : $S = \lceil N_{\text{page}} \frac{R}{B} \rceil$. In ogni passo di merge, possiamo fondere $Z = B - 1$ run, riducendo dunque il numero di run di un fattore Z : $k = \lceil \log_Z S \rceil = \lceil \log_{B-1} \lceil N_{\text{page}} \frac{R}{B} \rceil \rceil$. Possiamo dunque riscrivere il costo totale come:

$$2 \cdot N_{\text{page}}(R) \cdot (1 + \lceil \log_Z S \rceil) \quad (6)$$

Nello specifico, nel caso in cui avessimo $N_{\text{page}}(R) < B^2$ allora avremmo bisogno di una sola fusione; questo dal momento che $S = N_{\text{page}} \frac{R}{B} < B$. In questo caso il costo totale si ridurrebbe a $C_{\text{tot}} = 4 \cdot N_{\text{page}}(R)$.

Di seguito vediamo alcuni benchmark che mostrano il costo di external merge-sort in funzione del numero di pagine avendo fissati $B = 3$, $Z = 2$ e la differenza rispetto ad avere $B = 257$ e conseguentemente un valore di $Z = 256$.

$N_{\text{pag}}(R)$	B = 3, Z = 2			B = 257, Z = 256		
	Merge Passes	Cost	Time (m)	Merge Passes	Cost	Time (m)
100	6	1400	0.23	0	200	0.03
1000	9	20 000	3.33	1	4000	0.67
10 000	12	260 000	43.33	1	40 000	6.67
100 000	16	3 400 000	566.67	2	600 000	100.00
1 000 000	19	40 000 000	6666.67	2	6 000 000	1000.00

Figura 57: Benchmark di external merge-sort in funzione del numero di pagine per due diverse dimensioni del buffer: a sinistra $B=3$, a destra $B=257$

Indice delle figure

Figura 1	Interazioni del DBMS	5
Figura 2	Diagramma Entità-Relazione di una libreria	9
Figura 3	Due diversi piani di query per la stessa richiesta in algebra relazionale.	11
Figura 4	Esempio di applicazione dell'algoritmo MapReduce per contare le occorrenze di ogni parola all'interno di un input	15
Figura 5	Applicazione della funzionalità di combine al metodo MapReduce	15
Figura 6	Esempio di de-normalizzazione: a sinistra una struttura normalizzata basata su due entità distinte, a destra una rappresentazione de-normalizzata della stessa relazione	18
Figura 7	Esempio di Salvataggio atomico	18
Figura 8	Rappresentazione grafica della pipeline di aggregazione specificata nel codice sopra	26
Figura 9	Esempio di utilizzo dell'operatore <code>\$lookup</code> per effettuare un left-outer join ..	32
Figura 10	Organizzazione interna delle componenti di un R-DBMS	34
Figura 11	Esempio di Run-Length Encoding	38
Figura 12	Esempio di Bit-vector encoding	38
Figura 13	Esempio di dictionary-encoding	39
Figura 14	Esempio di dictionary-encoding per sequenze	39
Figura 15	Esempio di Frame-of-Reference encoding con valori oltre l'offset massimo	40
Figura 16	Esempio di encoding differenziale con valori oltre l'offset massimo	40
Figura 17	Terminologia degli Extensible Record Stores	43
Figura 18	Flusso delle operazioni di scrittura in un Extensible Record Store	45
Figura 19	Flusso delle operazioni di lettura in un Extensible Record Store	46
Figura 20	Formato di un file in un Extensible Record Store	47
Figura 21	Write-Ahead Logging in un Extensible Record Store	47
Figura 22	Compattazione dei file in un Extensible Record Store	49
Figura 23	Bloom Filter all'interno dei file in un Extensible Record Store	49
Figura 24	Due esempi di utilizzo di un Bloom Filter durante la lettura in un Extensible Record Store, <code>query1</code> mostra il caso in cui la chiave cercata è presente, <code>query2</code> mostra il caso in cui la chiave cercata non è presente	51
Figura 25	Esempio di grafo diretto (a sinistra) e indiretto (a destra).	52
Figura 26	Esempio di multigrafo con più archi tra alcuni nodi. e_3, e_4 sono archi multipli tra i nodi v_2 e v_3	52
Figura 27	Rappresentazione delle relazione 'conoscenti di conoscenti' tramite un grafo ..	53
Figura 28	Esempi di basi di dati a grafo in un social network (a sinistra) e in un sistema geospaziale (a destra).	55
Figura 29	Esempio di Property Graph che rappresenta una rete sociale.	55
Figura 30	Esempio di Property Graph con tipi e label che rappresenta una rete sociale.	56
Figura 31	Esempio di multi-edge non ammesso dal momento che ha la stessa label di un altro arco già presente per collegare gli stessi nodi.	56
Figura 32	Esempio di multigrafo non diretto con cicli (a sinistra) e la sua matrice di adiacenza (a destra).	58

Figura 33 Esempio di multigrafo diretto con cicli (a sinistra) e la sua matrice di incidenza (a destra)	59
Figura 34 Esempio di multigrafo diretto con cicli (a sinistra) e la sua matrice di incidenza (a destra)	59
Figura 35 Esempio di rappresentazione di un grafo indiretto (a sinistra) e diretto (a destra) tramite liste di adiacenza.	60
Figura 36 Esempio di rappresentazione di un grafo indiretto (a sinistra) e diretto (a destra) tramite liste di incidenza.	61
Figura 37 Esempio di pattern matching in Cypher per trovare nodi e relazioni specifiche all'interno di un grafo.	65
Figura 38 Grafo che rappresenta una rete sociale con relazioni di amicizia e ristoranti preferiti.	66
Figura 39 Esempio di property graph model che rappresenta persone e film con relazioni .	71
Figura 40 Illustrazione di un guasto su un nodo singolo in un sistema distribuito.	74
Figura 41 Illustrazione di un guasto di rete in un sistema distribuito.	74
Figura 42 Illustrazione di un partizionamento di rete in un sistema distribuito.	75
Figura 43 Esempio di Hash Tree con 8 nodi foglia data una lista di messaggi A,B,C,D,E,F,G,H.	77
Figura 44 Esempio di Consistent Hashing con 3 nodi e 6 frammenti di dati.	79
Figura 45 Esempio di Consistent Hashing con rimozione di un nodo.	79
Figura 46 Esempio di Consistent Hashing con aggiunta di un nodo.	80
Figura 47 Illustrazione dell'utilizzo di nodi virtuali nel Consistent Hashing.	80
Figura 48 Illustrazione di un guasto di signolo server con replication factor = 2.	83
Figura 49 Illustrazione di un guasto di entrambi i server con replication factor = 2.	83
Figura 50 Illustrazione del Two-Phase Commit: a sinistra un commit riuscito, a destra un commit abortito.	85
Figura 51 Rappresentazione schematica di un disco magnetico	89
Figura 52 Rappresentazione schematica del buffer pool	91
Figura 53 Rappresentazione schematica della tabella delle pagine residenti	92
Figura 54 a) Rappresentazione di utilizzo di riferimenti diretti per i record all'interno di una pagina. b) Rappresentazione di utilizzo di uno slot array per i record all'interno di una pagina.	95
Figura 55 Confronto tra modello seriale e sequenziale	100
Figura 56 Rappresentazione schematica dell'algoritmo di external merge-sort con dimensione del buffer di 3 pagine, e pagine di dimensione 2	102
Figura 57 Benchmark di external merge-sort in funzione del numero di pagine per due diverse dimensioni del buffer: a sinistra B=3, a destra B=257	103