Basi di dati

UniVR - Dipartimento di Informatica

Fabio Irimie

2° Semestre 2025/2026

Indice

1 Tecnologia dei DBMS **4**
1.1 Architettura transazionale di un DBMS . . . 4
1.2 Transazione . . . 5
1.2.1 Sintassi . . . 5
1.2.2 Proprietà delle transazioni . . . 5
1.3 Architettura di un DBMS . . . 6
1.3.1 Vista ad alto livello . . . 7
1.3.2 Modulo di gestione delle transazioni . . . 7

2 Linguaggio di interrogazione SQL **8**
2.1 PostgreSQL . . . 9
2.2 Fondamenti del DDL . . . 9
2.2.1 Identificatori . . . 9
2.2.2 Domini elementari . . . 9
2.2.3 Domini definiti dall'utente . . . 11
2.2.4 Vincoli intrarelazionali . . . 12
2.2.5 Vincoli interrelazionali . . . 13
2.3 CREATE TABLE . . . 14
2.4 Modifiche degli schemi . . . 14
2.4.1 DROP DOMAIN . . . 14
2.4.2 DROP TABLE . . . 14
2.4.3 ALTER . . . 15
2.5 Cancellazione dati in una tabella . . . 15
2.6 Inserimento dati in una tabella . . . 16
2.6.1 Aggiornamento dati in una tabella . . . 16
2.7 SELECT . . . 16
2.8 FROM . . . 18
2.9 WHERE . . . 19
2.9.1 LIKE . . . 21
2.9.2 SIMILAR TO . . . 22
2.9.3 BETWEEN . . . 23
2.9.4 IN . . . 23
2.9.5 IS NULL . . . 23
2.10 Gestione dei duplicati . . . 24
2.11 JOIN . . . 25
2.11.1 INNER JOIN . . . 27
2.11.2 LEFT OUTER JOIN . . . 27
2.11.3 RIGHT OUTER JOIN . . . 28
2.11.4 FULL OUTER JOIN . . . 28
2.12 Uso di variabili (alias) . . . 28
2.13 Ordinamento del risultato . . . 29
2.14 Operatori insiemistici . . . 30
2.15 Operatori aggregati . . . 31
2.15.1 COUNT . . . 31
2.15.2 SUM . . . 31
2.15.3 MAX/MIN . . . 32
2.15.4 AVG . . . 32
2.16 Interrogazioni con raggruppamento . . . 32

2.16.1 HAVING . . . 32
2.17 Interrogazioni nidificate . . . 32
2.17.1 WHERE . . . 33
2.17.2 Binding e EXISTS . . . 33
2.17.3 Interrogazioni nidificate nella clausola SELECT . . . 34
2.17.4 Interrogazioni nidificate nella clausola FROM . . . 34
2.17.5 Esercizi . . . 35
2.18 Viste . . . 38

3 Strutture fisiche e strutture di accesso ai dati 39
3.1 Gestore dei buffer . . . 39
3.1.1 Buffer . . . 39
3.1.2 Politica di gestione della memoria . . . 40
3.1.3 Gestione delle pagine . . . 40
3.1.4 Primitive per la gestione del buffer . . . 41
3.2 Gestore dell'affidabilità . . . 42
3.2.1 File di log . . . 42
3.2.2 Regole di scrittura sul file di log . . . 43
3.2.3 Guasti . . . 44
3.3 Gestore dei metodi di accesso . . . 46
3.3.1 Metodi di accesso . . . 47
3.3.2 Organizzazione di una pagina . . . 47
3.3.3 Operazioni sulle pagine . . . 49
3.3.4 Strutture primarie per l'organizzazione di file . . . 50
3.3.5 Struttura sequenziale . . . 50
3.3.6 Indici . . . 52
3.3.7 Struttura ad albero . . . 55
3.3.8 Struttura ad accesso calcolato (hash) . . . 61
3.3.9 Collisioni . . . 63

4 Transazioni concorrenti 64
4.1 Anomalie . . . 65
4.1.1 Perdita di aggiornamento . . . 65
4.1.2 Lettura inconsistente . . . 66
4.1.3 Lettura sporca . . . 66
4.1.4 Aggiornamento fantasma . . . 67
4.1.5 Inserimento fantasma . . . 67
4.2 Schedule . . . 68
4.2.1 Schedule seriale . . . 68
4.3 Tecniche di gestione della concorrenza . . . 68
4.3.1 View-equivalenza . . . 68
4.3.2 Conflict-equivalenza . . . 74
4.3.3 Locking a due fasi stretto . . . 79
4.3.4 Blocco critico (deadlock) . . . 81
4.3.5 Gestione della concorrenza in SQL . . . 83

5 Ottimizzazione **84**
5.1 Compilazione della query 84
5.2 Ottimizzazione dipendente dai metodi di accesso 84
5.2.1 Scansione sequenziale 85
5.2.2 Ordinamento 85
5.2.3 Accesso diretto tramite indice 87
5.2.4 Join 87
5.3 Scelta del piano di esecuzione 92
5.4 Riepilogo dei costi 93
5.4.1 Esercizi 94

6 MongoDB **100**
6.1 Replicazione 100
6.1.1 Algoritmo di elezione Raft 101
6.1.2 Oplog 101
6.1.3 Replicazione delle scritture 101
6.1.4 Replicazione delle letture 102
6.2 Architettura dei replica set 103
6.3 Sharding 103
6.3.1 Architettura dello sharding 104
6.3.2 Balancer 104
6.4 Interrogazioni in presenza di shared data 104
6.4.1 Proprietà ACID 105

7 Esercizi **105**
7.1 Query 105
7.1.1 Esercizio 1 106
7.1.2 Esercizio 2 106
7.1.3 Esercizio 3 107

1 Tecnologia dei DBMS

Un DBMS (Database Management System) è un software in grado di gestire collezioni di dati che siano **grandi, condivise e persistenti** assicurando **affidabilità e privatezza**.

- Grandi: i dati devono essere organizzati in modo efficiente per poter essere gestiti in grandi quantità
- Condivise: i dati devono essere accessibili da più utenti contemporaneamente
- Persistenti: i dati devono essere memorizzati in modo permanente, anche dopo la chiusura del programma
- Affidabilità: i dati devono essere protetti da errori e guasti del sistema
- Privatezza: i dati devono essere protetti da accessi non autorizzati

Un DBMS deve essere anche **efficiente** ed **efficace**:

- Efficiente: il DBMS deve essere in grado di gestire grandi quantità di dati in modo rapido e con un utilizzo minimo delle risorse
- Efficace: il DBMS deve essere in grado di soddisfare le esigenze degli utenti in modo completo e accurato

Il DBMS si appoggia sul filesystem estendendo le sue funzionalità.

1.1 Architettura transazionale di un DBMS

Un DBMS è composto da tre livelli:

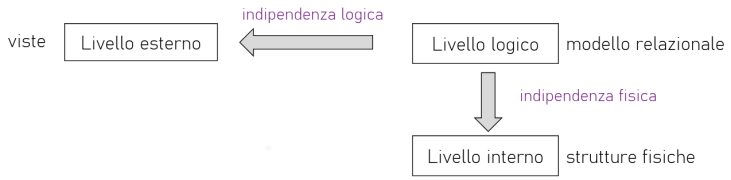


Figura 1: Architettura a 3 livelli di un DBMS

Un DBMS basato sul **modello relazionale** è nella maggior parte dei casi anche un **sistema transazionale**, cioè fornisce un meccanismo per la definizione ed esecuzione di transazioni:

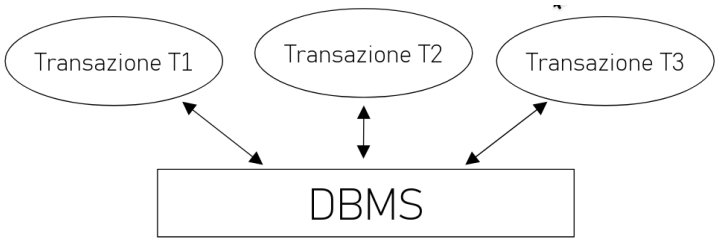


Figura 2: Architettura transazionale di un DBMS

1.2 Transazione

Definizione 1.1. È **un'unità** di lavoro svolta da un programma applicativo (che interagisce con una base di dati) per la quale si vogliono garantire proprietà di correttezza, robustezza e isolamento.

La principale caratteristica di una transazione è che deve essere **atomica**, cioè deve essere eseguita completamente o non essere eseguita affatto. **Non sono ammesse esecuzioni parziali**.

1.2.1 Sintassi

La sintassi per definire una transazione è la seguente:

```
<transazione> -> begin transaction
                 <programma>
                 end transaction

<programma> -> < <istruzione> | commit work | rollback work >
               { < <istruzione> | commit work | rollback work > }
```

- La transazione va a buon fine all'esecuzione di un `commit work`.
- La transazione non ha alcun effetto se viene eseguito un `rollback work`.

Una transazione è ben formata se:

- Inizia con un `begin transaction`.
- Termina con un `end transaction`.
- La sua esecuzione comporta il raggiungimento di un `commit work` o di un `rollback work` e dopo il `commit/rollback` non si eseguono altri accessi alla base di dati.

Esempio 1.1. Un esempio di transazione ben formata è il seguente:

```
begin transaction;
  update CONTO set saldo = saldo - 1200
    where filiale = '005' and numero = 15;
  update CONTO set saldo = saldo + 1200
    where filiale = '005' and numero = 205;
  commit work;
end transaction;
```

1.2.2 Proprietà delle transazioni

Una transazione ha 4 proprietà fondamentali, note come ACID:

- **Atomicità** (**A**tomicity): Una transazione è una unità di esecuzione **indivisibile**. O viene eseguita completamente o non viene eseguita affatto.

  Implicazioni:

  - Se una transazione viene interrotta prima del commit, il lavoro fin qui eseguito dalla transazione deve essere disfatto ripristinando la situazione in cui si trovava la base di dati prima dell'inizio della transazione.
  - Se una transazione viene interrotta dopo l'esecuzione del commit (commit eseguito con successo), il sistema deve assicurare che la transazione abbia effetto sulla base di dati.

- **Consistenza** (**C**onsistency): L'esecuzione di una transazione non deve violare i vincoli di integrità.

  Implicazioni:

  - Verifica immediata:
    * Fatta nel corso della transazione
    * Viene abortita solo l'ultima operazione e il sistema restituisce all'applicazione una segnalazione d'errore
    * L'applicazione può quindi reagire alla violazione.
  - Verifica differita:
    * Al commit se un vincolo di integrità viene violato la transazione viene abortita senza possibilità da parte dell'applicazione di reagire alla violazione.

- **Isolamento** (**I**solation): L'esecuzione di una transazione deve essere indipendente dalla contemporanea esecuzione di altre transazioni.

  Implicazioni:

  - Il rollback di una transazione non deve creare rollback a catena di altre transazioni che si trovano in esecuzione contemporaneamente.
  - Il sistema deve regolare l'esecuzione concorrente con meccanismi di controllo dell'accesso alle risorse.

- **Persistenza** (**D**urability): L'effetto di una transazione che ha eseguito il commit non deve andare perso.

  Implicazioni:

  - Il sistema deve essere in grado, in caso di guasto, di garantire gli effetti delle transazioni che al momento del guasto avevano già eseguito un commit.

Un DBMS che gestisce transazioni dovrebbe garantire per ogni transazione che esegue tutte queste proprietà.

1.3 Architettura di un DBMS

L'architettura di un DBMS è composta da **moduli** che si occupano di gestire le diverse funzionalità del sistema.

1.3.1 Vista ad alto livello

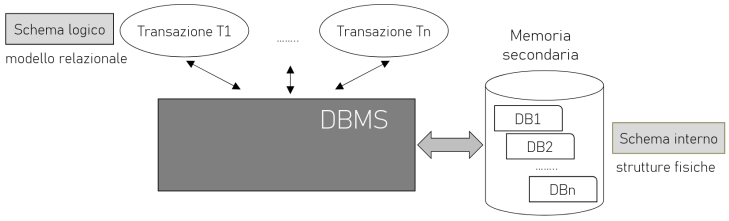


Figura 3: Vista ad alto livello dell'architettura di un DBMS

1.3.2 Modulo di gestione delle transazioni

Questo modulo si occupa di trasformare una transazione in un insieme di operazioni sulla base di dati, garantendo le proprietà ACID.

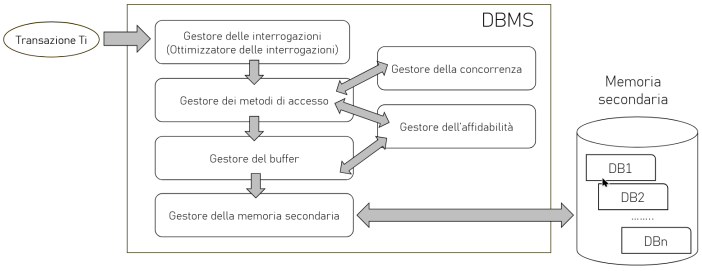


Figura 4: Modulo di gestione delle transazioni di un DBMS

I singoli moduli che compongono questo modulo sono:

- **Gestore delle interrogazioni**: Riceve in ingresso una query SQL

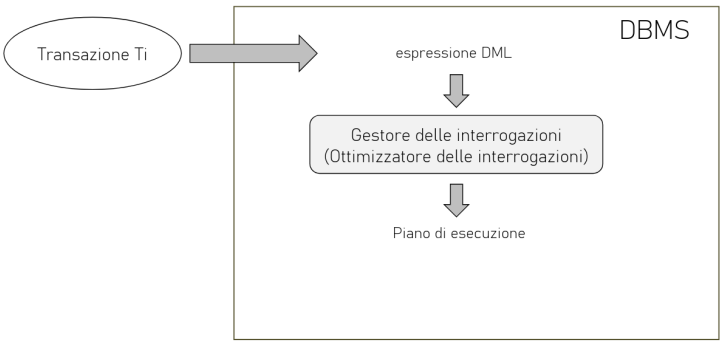


Figura 5: Gestore delle interrogazioni di un DBMS

- **Gestore dei metodi di accesso**: Determina quali sono le pagine della memoria secondaria di cui si ha bisogno, ma capisce anche se è possibile accederci.

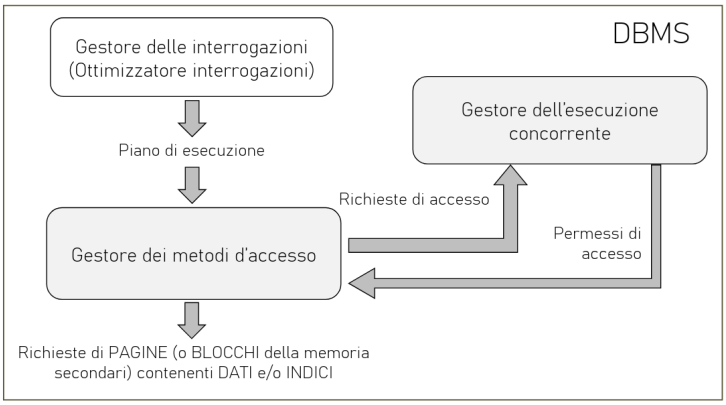


Figura 6: Gestore dei metodi di accesso di un DBMS

- **Gestore dell'affidabilità**: Carica in memoria principale le pagine della memoria secondaria di cui si ha bisogno, e le mantiene in memoria finché non sono più necessarie.

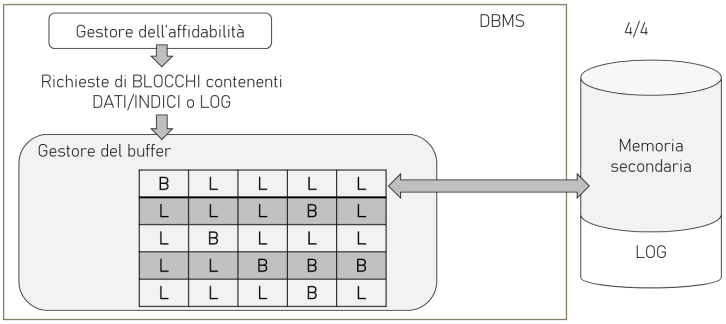


Figura 7: Gestore del buffer di un DBMS

- **Gestore del buffer**
- **Gestore della memoria secondaria**
- **Gestore della concorrenza**

I moduli che contribuiscono a garantire le proprietà ACID sono:

- **Gestore dei metodi di accesso**: garantisce la consistenza
- **Gestore dell'esecuzione concorrente**: garantisce l'isolamento e l'atomicità
- **Gestore dell'affidabilità**: garantisce l'atomicità e la persistenza

2 Linguaggio di interrogazione SQL

SQL (Structured Query Language) è il linguaggio di riferimento per le basi di dati relazionali. SQL è un linguaggio dichiarativo composto da 3 tipi di istruzioni:

- **Data Definition Language** (DDL): linguaggio per la definizione della struttura della base di dati (es. `CREATE`, `ALTER`, `DROP`)
- **Data Manipulation Language** (DML): linguaggio per manipolare i dati come inserimento, aggiornamento e cancellazione (es. `INSERT`, `UPDATE`, `DELETE`)
- **Data Query Language** (DQL): linguaggio per interrogare i dati (es. `SELECT`)

Esistono varie versioni di SQL, in questo corso ci baseremo su SQL-92 (SQL-2), che è una versione standard del linguaggio SQL.

La parte di SQL dedicata alle interrogazioni fa parte dei DQL (Data Query Language). SQL esprime le interrogazioni in modo dichiarativo (specifica l'obiettivo e non la procedura per ottenerlo, come l'algebra relazionale). Una parte del DBMS, detta query optimizer, traduce di fatto l'interrogazione SQL nel linguaggio procedurale interno del sistema, che è nascosto all'utente. SQL permette quindi di descrivere le interrogazioni in modo astratto ed ad alto livello.

2.1 PostgreSQL

È un Relational Database Management System (RDBMS) con alcune funzionalità orientate agli oggetti e verrà utilizzato all'interno del corso. Il modello di interazione con le basi di dati è **Client-Server**.

2.2 Fondamenti del DDL

2.2.1 Identificatori

Un identificatore è una sequenza di caratteri che identifica un elemento del database come ad esempio una tabella, un attributo, un vincolo, ecc. Gli identificatori in SQL devono rispettare le seguenti regole:

- Devono iniziare con una lettera (UTF-8) o un underscore (_)
- Possono contenere lettere, numeri (0-9) e underscore (_)
- Non possono essere parole chiave riservate di SQL

Gli identificatori sono **case-insensitive**, quindi `idPersona` e `idpersona` sono considerati lo stesso identificatore.

2.2.2 Domini elementari

I domini elementari sono i tipi predefiniti di SQL:

- **Tipo carattere**: Rappresenta stringhe di lunghezza variabile secondo la seguente sintassi:

  ```
  CHARACTER [VARYING][(lunghezza)]
  ```

  - `CHARACTER`: carattere singolo
  - `CHARACTER(20)`: stringa di lunghezza fissa di 20 caratteri. Se la stringa è più corta di 20 caratteri, viene riempita con spazi a destra.

- `CHARACTER VARYING(20)`: stringa di lunghezza variabile con un massimo di 20 caratteri. Se la stringa è più corta di 20 caratteri, viene memorizzata senza spazi aggiuntivi.
- `TEXT` (estensione PostgreSQL): stringa di lunghezza variabile senza un limite.

Le stringhe sono delimitate da apici singoli (`'`).

- **Tipo booleano**: Rappresenta valori booleani (vero o falso) più lo stato nullo secondo la seguente sintassi:

```
BOOLEAN
```

La rappresentazione dei valori booleani può essere:

- `TRUE` o `FALSE`
- `t` o `f`
- `1` o `0`
- `yes` o `no`
- `NULL`

- **Tipo numerico esatto**: Rappresenta numeri interi o con parte decimale fissa.
  - Valori interi:
    * `SMALLINT`: 16bit con segno
    * `INTEGER`: 32bit con segno
  - Valori decimali:
    * `NUMERIC | DECIMAL [(precisione [, scala])]`
      · `precisione`: numero totale di cifre (intere + decimali)
      · `scala`: numero di cifre decimali
      · `|`: indica che è possibile utilizzare sia `NUMERIC` che `DECIMAL`

- **Tipo numerico approssimato**: Rappresenta numeri in virgola mobile **approssimati**:
  - `REAL`: 6 cifre decimali
  - `DOUBLE PRECISION`: 15 cifre decimali

- **Tipo temporale**: Rappresenta istanti di tempo:
  - `DATE`

    Rappresenta una data (fra apici) nel formato ISO: `YYYY-MM-DD` (anno-mese-giorno)
  - `TIME [(precisione)][WITH TIME ZONE]`

    Rappresenta un'orario del tipo `hh:mm:ss` (ore-minuti-secondi).
    * `precisione`: numero di cifre decimali per le frazioni di secondo

    * WITH TIME ZONE: indica che l'orario include anche il fuso orario nei seguenti possibili formati:
      · [+-]hh:mm: rappresenta la differenza rispetto all'ora di Greenwich
      · Nome del fuso orario (es. UTC, CET, EST, ecc.)

  - `TIMESTAMP [(precisione)][WITH TIME ZONE]`

    Equivale a DATE + TIME.

- **Tipo intervallo temporale**: Rappresenta un intervallo di tempo secondo la seguente sintassi:

  ```
  INTERVAL PrimaUnitaDiTempo [TO UltimaUnitaDiTempo]
  ```

  dove PrimaUnitaDiTempo e UltimaUnitaDiTempo definiscono le unità di misura che devono essere utilizzate, dalla più precisa alla meno precisa.

  - INTERVAL YEAR
  - INTERVAL MONTH
  - INTERVAL DAY
  - INTERVAL YEAR TO MONTH
  - INTERVAL DAY TO HOUR
  - ecc...

  Ad esempio INTERVAL "1 DAY"

2.2.3 Domini definiti dall'utente

Per creare un dominio si usa il seguente comando:

```
CREATE DOMAIN nome AS tipoBase [valoreDefault][vincolo]
```

Definisce un dominio utilizzabile in definizioni di relazioni, anche con vincoli e valori default. Il dominio può essere definito a partire da un dominio elementare oppure da un altro dominio definito dall'utente

Esempio 2.1. Alcuni esempi di domini definiti dall'utente sono i seguenti:

```
CREATE DOMAIN Voto                        -- Nome dominio
  AS SMALLINT                             -- Tipo base
  DEFAULT NULL                            -- Valore default
  CHECK ( value >=18 AND value <= 30 ) -- Vincolo
```

```
CREATE DOMAIN giorniSettimana AS CHARACTER (3)
  CHECK (VALUE IN (
    "LUN",
    "MAR",
    "MER",
    "GIO",
    "VEN",
    "SAB",
    "DOM"
  ));
```

2.2.4 Vincoli intrarelazionali

I vincoli intrarelazionali sono vincoli che si applicano a livello di relazione, cioè a una singola tabella. I principali tipi di vincoli intrarelazionali sono:

- `NOT NULL`: Il valore nullo non è ammesso per un attributo

  ```
  nome VARCHAR(20) NOT NULL
  ```

- `DEFAULT`: Specifica un valore di default per un attributo quando il comando di inserimento non specifica nessun valore per l'attributo

  ```
  cognome VARCHAR(20) NOT NULL DEFAULT "VUOTO"
  ```

- `UNIQUE`: Impone che i valori di un attributo (o insieme di attributi) siano una superchiave, cioè righe diverse della tabella non possono avere gli stessi valori

  - Il seguente vincolo impone che non ci siano due righe che abbiano sia il nome che il cognome uguale

    ```
    Nome VARCHAR(20),
    Cognome VARCHAR(20),
    UNIQUE(Nome,Cognome)
    ```

  - Il seguente vincolo impone che non ci siano due righe che abbiano lo stesso nome o lo stesso cognome

    ```
    Nome VARCHAR(20) UNIQUE,
    Cognome VARCHAR(20) UNIQUE
    ```

- `PRIMARY KEY`: Identifica l'attributo che rappresenta la chiave primaria della relazione. Impone i seguenti vincoli:

  - Si usa una sola volta per tabella
  - Implica il vincolo `NOT NULL`

  Si può definire in due modi:

  - Nella definizione dell'attributo se la chiave primaria è composta da un solo attributo:

    ```
    matricola CHAR(6) PRIMARY KEY;
    ```

  - Nel vincolo della tabella se la chiave primaria è composta da più attributi:

    ```
    nome VARCHAR(20),
    cognome VARCHAR(20),
    PRIMARY KEY(nome, cognome)
    ```

- `CHECK`: Specifica un vincolo generico che devono soddisfare le tuple della tabella. Il vincolo è soddisfatto se la sua espressione è **vera** o **nulla**.

```sql
CREATE TABLE Impiegato (
  matricola CHAR(6) PRIMARY KEY,
  nome VARCHAR(20) NOT NULL,
  cognome VARCHAR(20) NOT NULL,
  qualifica VARCHAR(20),
  stipendio NUMERIC(8,2) DEFAULT 500.00 NOT NULL
  CHECK(stipendio >= 0.0), --check di attributo
  UNIQUE(cognome, nome),
  CHECK(nome <> cognome) --check di tabella
);
```

2.2.5 Vincoli interrelazionali

Un vincolo interrelazionale (o di integrità referenziale) crea un legame tra i valori di un attributo (o insieme di attributi) A della tabella corrente, detta **interna o slave**, e i valori di un attributo (o di un insieme di attributi) B di un'altra tabella detta **esterna o master**.

Il vincolo impone che:

- In ogni tupla della tabella **interna**, il valore di A (se è diverso dal valore nullo), deve essere presente tra i valori di B nella tabella esterna.
- L'attributo B della tabella **esterna** deve essere soggetto ad un vincolo `UNIQUE` o `PRIMARY KEY`. Può anche non essere la chiave primaria, ma deve essere identificante per le tuple della tabella esterna.

Per definire un vincolo di integrità referenziale si usa la seguente sintassi:

- `REFERENCES`: Se nel vincolo è coinvolto solo un attributo, si specifica la tabella esterna e il relativo attributo coinvolto nel vincolo.

```sql
CREATE TABLE Impiegato ( -- Tabella interna
  Matricola CHARACTER(6) PRIMARY KEY,
  Nome CHARACTER VARYING(20) NOT NULL,
  Cognome CHARACTER VARYING(20) NOT NULL,
  Dipart CHARACTER VARYING(15)
    REFERENCES Dipartimento(NomeDip), -- Tabella esterna
  Ufficio NUMERIC(3),
  Stipendio NUMERIC(9) DEFAULT 0,
  UNIQUE (Cognome, Nome)
);
```

- `FOREIGN KEY`: Se nel vincolo è coinvolto un insieme di attributi, si specifica l'insieme di attributi della tabella interna coinvolti nel vincolo. Poi con il vincolo `REFERENCES` si specifica la tabella esterna e l'insieme di attributi della tabella esterna coinvolti nel vincolo.

```sql
CREATE TABLE Impiegato ( -- Tabella interna
  Matricola CHARACTER(6) PRIMARY KEY,
  Nome CHARACTER VARYING(20) NOT NULL,
  Cognome CHARACTER VARYING(20) NOT NULL,
  Dipart CHARACTER VARYING(15)
    REFERENCES Dipartimento(NomeDip),
  Ufficio NUMERIC(3),
  Stipendio NUMERIC(9) DEFAULT 0,
  FOREIGN KEY (Nome, Cognome)
    REFERENCES Anagrafica (Nome, Cognome) -- Tabella esterna
);
```

2.3 CREATE TABLE

Definisce uno schema di relazione e ne crea un'istanza vuota specificando attributi, domini e vincoli secondo la seguente sintassi:

```
CREATE TABLE tabella (
     attributo dominio [ valoreDefault ] { vincolo }
  {, attributo dominio [ valoreDefault ] { vincolo } }
  {, vincoloTabella }
);
```

dove:

- `[]`: indicano parti opzionali
- `[ valoreDefault ]`: indica un eventuale valore default
- `{}`: indica un elemento che può essere ripetuto zero o più volte
- `{ vincolo }`: indica eventuali (zero o più) vincoli di integrità per ogni attributo

Esempio 2.2. Un esempio di creazione di una tabella è il seguente:

```
CREATE TABLE Impiegato (
  Matricola CHARCTER(6) PRIMARY KEY,
  Nome      CHARCTER    VARYING(20) NOT NULL,
  Cognome   CHARCTER    VARYING(20) NOT NULL,
  Dipart    CHARCTER    VARYING(15),
  Stipendio NUMERIC(9)  DEFAULT 0,

  -- Vincoli di integrita'
  FOREIGN KEY(Dipart)
    REFERENCES Dipartimento(NomeDip),
  UNIQUE (Cognome, Nome)
)
```

2.4 Modifiche degli schemi

2.4.1 DROP DOMAIN

Permette di rimuovere i domini definiti dall'utente secondo la seguente sintassi:

```
DROP DOMAIN nomeDominio
```

2.4.2 DROP TABLE

Permette di rimuovere le tabelle secondo la seguente sintassi:

```
DROP TABLE nomeTabella
```

2.4.3 ALTER

Effettua modifiche a domini, tabelle, ecc...

- **Aggiunta di un nuovo attributo** (`ADD COLUMN`):
  - Sintassi:

    ```
    ALTER TABLE tabella ADD COLUMN attributo tipo;
    ```

  - Esempio:

    ```
    ALTER TABLE impiegato ADD COLUMN stipendio NUMERIC(8,2);
    ```

- **Rimozione di un attributo** (`DROP COLUMN`):
  - Sintassi:

    ```
    ALTER TABLE tabella DROP COLUMN attributo;
    ```

  - Esempio:

    ```
    ALTER TABLE impiegato DROP COLUMN stipendio;
    ```

- **Modifica valore di default di un attributo** (`ALTER COLUMN`):
  - Sintassi:

    ```
    ALTER TABLE tabella ALTER COLUMN attributo { SET DEFAULT
        valore | DROP DEFAULT };
    ```

  - Esempio:

    ```
    ALTER TABLE impiegato ALTER COLUMN stipendio SET DEFAULT
        1000.00;
    ```

- **Ridenominazione di un attributo** (`RENAME COLUMN`):
  - Sintassi:

    ```
    ALTER TABLE tabella RENAME COLUMN vecchioAttributo TO
        nuovoAttributo;
    ```

  - Esempio:

    ```
    ALTER TABLE impiegato RENAME COLUMN stipendio TO salario;
    ```

2.5 Cancellazione dati in una tabella

Le tuple di una tabella vengono cancellate con il comando `DELETE` secondo la seguente sintassi:

```
DELETE FROM tabella [WHERE condizione];
```

- `condizione`: è un'espressione booleana che seleziona quali tuple cancellare
- Se `WHERE` non è presente **tutte le tuple saranno cancellate**

Ad esempio:

```
DELETE FROM impiegato WHERE matricola = "A001";
```

2.6 Inserimento dati in una tabella

Le tuple di una tabella vengono inserite con il comando `INSERT` secondo la seguente sintassi:

```
INSERT INTO tabella [(attributo1, attributo2, ...)]
VALUES (valore1, valore2, ...);
```

oppure

```
INSERT INTO tabella [(attributo1, attributo2, ...)]
SELECT (...);
```

- Se la lista degli attributi non è presente, i valori devono essere specificati nell'ordine in cui sono stati definiti nella tabella.
- Se un attributo non è specificato, viene utilizzato il valore di default se è definito, altrimenti viene utilizzato il valore nullo.

Ad esempio:

```
INSERT INTO impiegato (matricola, nome, cognome, dipartimento)
VALUES ("A001", "Mario", "Rossi", "Informatica");
```

2.6.1 Aggiornamento dati in una tabella

Le tuple di una tabella vengono aggiornate con il comando `UPDATE` secondo la seguente sintassi:

```
UPDATE tabella
SET attributo1 = valore1, attributo2 = valore2, ...
[WHERE condizione];
```

- `condizione`: è un'espressione booleana che seleziona quali tuple aggiornare
- Se `WHERE` non è presente **tutte le tuple saranno aggiornate**

Ad esempio:

```
UPDATE impiegato
SET stipendio = stipendio * 1.1
WHERE dipartimento = "Informatica";
```

2.7 SELECT

La sintassi dell'istruzione SELECT è la seguente:

```
SELECT [ DISTINCT ]
[ * | expression [[ AS] output_name ] [, ...] ]
[ FROM from_item [, ...] ]
[ WHERE condition ]
[ GROUP BY grouping_element [, ...] ]
[ HAVING condition [, ...] ]
[ { UNION | INTERSECT | EXCEPT } [ DISTINCT ] other_select ]
[ ORDER BY expression [ ASC | DESC | USING operator ]]
```

dove:

- `expression`: è un espressione che determina un attributo

- `from_item`: è un'espressione che determina una sorgente per gli attributi
- `condition`: è un'espressione booleana per selezionare i valori degli attributi
- `grouping_element`: è un'espressione per poter eseguire operazioni su più valori di un attributo

La struttura dell'istruzione è la seguente:

```
SELECT ListaAttributi
FROM ListaTabelle
[WHERE Condizione]
```

`SELECT`, `FROM` e `WHERE` sono chiamate **clausole**. Il `SELECT` estrae dal prodotto cartesiano delle tabelle elencate nella clausola `FROM`, le righe che soddisfano la condizione specificata nella clausola `WHERE`. Il risultato dell'interrogazione è una tabella con una riga per ogni riga prodotta dalla clausola `FROM` ed eventualmente filtrata dalla clausola `WHERE` (se esiste), le cui colonne dipendono dalla lista degli attributi nella clausola `SELECT`.

Esempio 2.3. Prendendo ad esempio la seguente base di dati:

Impiegato

Nome	Cognome	Dipart	Ufficio	Stipendio	Città
Mario	Rossi	Amministrazione	10	45	Milano
Carlo	Bianchi	Produzione	20	36	Torino
Giovanni	Verdi	Amministrazione	20	40	Roma
Franco	Neri	Distribuzione	16	45	Napoli
Carlo	Rossi	Direzione	14	80	Milano
Lorenzo	Gialli	Direzione	7	73	Genova
Paola	Rosati	Amministrazione	75	40	Venezia
Marco	Franco	Produzione	20	46	Roma

Figura 8: Esempio di base di dati

Un esempio di istruzione SELECT è il seguente:
Estrarre lo stipendio degli impiegati di cognome 'Rossi'

```
SELECT Stipendio
FROM Impiegato
WHERE Cognome = 'Rossi';
```

Il risultato di questa interrogazione è la seguente tabella:

Stipendio
45
80

Figura 9: Risultato dell'istruzione SELECT

Alcune possibili espressioni sono:

- *: seleziona tutti gli attributi:

```
SELECT *
FROM Impiegato;
```

- as: permette di rinominare un attributo:

```
SELECT Stipendio AS Salario
FROM Impiegato;
```

2.8 FROM

Clausola FROM: specifica le tabelle da cui estrarre i dati:

```
FROM from_item [, ...]
```

dove `from_item` può essere **una sola** delle seguenti clausole:

1. **table_name [[AS] alias [(column_alias [, ...])]**: Se sono presenti due o più tabelle, si esegue il prodotto cartesiano tra tutte le tabelle. Se ci sono attributi con lo stesso nome su tabelle diverse, nella SELECT questi attributi sono identificati anteponendo il nome della tabella e un '.' (es. `nomeTabella.nomeAttributo`).

2. **(other_select) [AS] nomeRisultato [(column_alias [,...])]**: è una SELECT innestata. Il risultato della SELECT interna è lo schema con nome `nomeRisultato` su cui fare la SELECT corrente.

3. **from_item [NATURAL] join_type from_item [ON condition]**: è la clausola JOIN. Metodo più sofisticato di 1) che permette di selezionare un sottoinsieme del prodotto cartesiano di due o più tabelle. `join_type` specifica il tipo di join.

Esempio 2.4. Considerando la seguente base di dati:

Dipartimento

Nome	Indirizzo	Città
Amministrazione	Via Tito Livio, 27	Milano
Produzione	P.le Lavater, 3	Torino
Distribuzione	Via Segre, 9	Roma
Direzione	Via Tito Livio, 27	Milano
Ricerca	Via Venosa, 6	Milano

Impiegato

Nome	Cognome	Dipart	Ufficio	Stipendio	Città
Mario	Rossi	Amministrazione	10	45	Milano
Carlo	Bianchi	Produzione	20	36	Torino
Giovanni	Verdi	Amministrazione	20	40	Roma
Franco	Neri	Distribuzione	16	45	Napoli
Carlo	Rossi	Direzione	14	80	Milano
Lorenzo	Gialli	Direzione	7	73	Genova
Paola	Rosati	Amministrazione	75	40	Venezia
Marco	Franco	Produzione	20	46	Roma

Figura 10: Esempio di base di dati per la clausola JOIN

Estrarre i nomi ed il cognome degli impiegati e le città in cui lavorano:

```
SELECT Impiegato.Nome, Impiegato.Cognome,
       Dipartimento.Citta'
FROM Impiegato, Dipartimento
WHERE Impiegato.Dipart = Dipartimento.Nome
```

Il risultato di questa interrogazione è la seguente tabella:

Impiegato.Nome	Impiegato.Cognome	Dipartimento.Città
Mario	Rossi	Milano
Carlo	Bianchi	Torino
Giovanni	Verdi	Milano
Franco	Neri	Roma
Carlo	Rossi	Milano
Lorenzo	Gialli	Milano
Paola	Rosati	Milano
Marco	Franco	Torino

Figura 11: Risultato dell'istruzione SELECT con clausola FROM

2.9 WHERE

Clausola WHERE: specifica la condizione per selezionare le righe da estrarre:

```
WHERE condition
```

La clausola `WHERE` ammette come argomento un'espressione booleana costruita combinando predicati semplici con gli operatori AND, OR e NOT

- **AND**: seleziona le righe in cui tutti i predicati sono veri
- **OR**: seleziona le righe in cui almeno un predicato è vero
- **NOT**: seleziona le righe in cui il predicato è falso

Ciascun predicato semplice usa gli operatori =, <> (<> $\Longleftrightarrow \neq$), <, >, <= e >= per confrontare un'espressione costruita a partire dal valore di un attributo con un valore costante, oppure un'altra espressione.

Esempio 2.5. Considerando la seguente base di dati:

Dipartimento

Nome	Indirizzo	Città
Amministrazione	Via Tito Livio, 27	Milano
Produzione	P.le Lavater, 3	Torino
Distribuzione	Via Segre, 9	Roma
Direzione	Via Tito Livio, 27	Milano
Ricerca	Via Venosa, 6	Milano

Impiegato

Nome	Cognome	Dipart	Ufficio	Stipendio	Città
Mario	Rossi	Amministrazione	10	45	Milano
Carlo	Bianchi	Produzione	20	36	Torino
Giovanni	Verdi	Amministrazione	20	40	Roma
Franco	Neri	Distribuzione	16	45	Napoli
Carlo	Rossi	Direzione	14	80	Milano
Lorenzo	Gialli	Direzione	7	73	Genova
Paola	Rosati	Amministrazione	75	40	Venezia
Marco	Franco	Produzione	20	46	Roma

Figura 12: Esempio di base di dati per la clausola JOIN

Estrarre i nomi ed il cognome degli impiegati che lavorano nell'ufficio 20 del dipartimento Amministrazione.

```
SELECT Nome, Cognome
FROM Impiegato
WHERE Uffico = 20 AND
      Dipart = 'Amministrazione'
```

Il risultato di questa interrogazione è la seguente tabella:

Nome	Cognome
Giovanni	Verdi

Figura 13: Risultato dell'istruzione SELECT con clausola WHERE

Esempio 2.6. Considerando la base di dati precedente:

Estrarre i nomi ed il cognome degli impiegati che lavorano nel dipartimento Amministrazione o nel dipartimento Produzione.

```
SELECT Nome, Cognome
FROM Impiegato
WHERE Dipart = 'Amministrazione'
      OR Dipart = 'Produzione'
```

Il risultato di questa interrogazione è la seguente tabella:

Nome	Cognome
Mario	Rossi
Carlo	Bianchi
Giovanni	Verdi
Paola	Rosati
Marco	Franco

Figura 14: Risultato dell'istruzione SELECT con clausola WHERE e operatore OR

Esempio 2.7. Considerando la base di dati precedente:

Estrarre i nomi degli impiegati di cognome Rossi che lavorano dipartimento Amministrazione o nel dipartimento Produzione.

```
SELECT Nome
FROM Impiegato
WHERE Cognome = 'Rossi' AND
      (Dipart = 'Amministrazione' OR
      Dipart = 'Produzione')
```

Il risultato di questa interrogazione è la seguente tabella:

Nome
Mario

Figura 15: Risultato dell'istruzione SELECT con clausola WHERE e operatori AND e OR

2.9.1 LIKE

Per il confronto tra stringhe, SQL mette a disposizione anche l'operatore LIKE che consente verificare se una stringa rispetta un certo "pattern" (oppure ILIKE per case insentive). I due caratteri speciali per il confronto sono:

- _: confronto con un carattere arbitrario
- %: confronto con una stringa di lunghezza arbitraria (anche 0) di caratteri arbitrari

Esempio 2.8. Un esempio di pattern è il seguente:

```
_o%i
```

che equivale a una stringa che ha una o in seconda posizione, seguita da una sequenza qualsiasi di caratteri (anche vuota) e che termina con una i.

Esempio 2.9. Considerando la base di dati precedente:

Estrarre gli impiegati che hanno un cognome cha ha una "o" in seconda posizione e finisce con una "I".

```
SELECT *
FROM Impiegato
WHERE Cognome LIKE '_o%i';
```

Il risultato di questa interrogazione è la seguente tabella:

Nome	Cognome	Dipart	Ufficio	Stipendio	Città
Mario	Rossi	Amministrazione	10	45	Milano
Carlo	Rossi	Direzione	14	80	Milano
Paola	Rosati	Amministrazione	75	40	Venezia

Figura 16: Risultato dell'istruzione SELECT con clausola WHERE e operatore LIKE

2.9.2 SIMILAR TO

L'operatore SIMILAR TO è un LIKE più espressivo che accetta, come pattern, un sottoinsieme delle espressioni regolari POSIX (versione SQL).

La sintassi delle espressioni regolari è:

- _: 1 carattere qualsiasi
- %: 0 o più caratteri qualsiasi
- *: ripetizione del precedente match 0 o più volte
- +: ripetizione del precedente match UNA o più volte
- {n,m}: ripetizione del precedente match almeno n e non più di m volte
- [...]: ... è un elenco di caratteri ammissibili

Esempio 2.10. Considerando la seguente base di dati:

matricola	cognome	nome	indirizzo	città	media
IN0001	Rossi	Marco	via nobile	Verona	27.50
IN0002	Paoli	Paolo	via dritta	Padova	28.6666
IN0003	Bianchi	Dante	via storta	VERONA	18.345
IN0004	Rossi	Matteo	via nobile	Verona	24.567
IN0005	Verdi	Luca	via nuova	Va	28.6666
IN0006	Nocasa	Caio			

Figura 17: Esempio di base di dati per la clausola SIMILAR TO

Estrarre gli studenti con cognome che inizia con 'A' o 'B', o 'D', o 'N' e finisce con 'a':

```
SELECT cognome, nome, citta' FROM Studente
```

```
WHERE cognome SIMILAR TO '[ABDN]{1}%a';
```

Il risultato di questa interrogazione è evidenziato in giallo.

2.9.3 BETWEEN

L'operatore BETWEEN consente di verificare se un valore è compreso tra due valori estremi (inclusi). La sintassi è la seguente:

```
expression BETWEEN value1 AND value2
```

Esempio 2.11. Considerando la seguente base di dati:

matricola	cognome	nome	indirizzo	città	media
IN0001	Rossi	Marco	via nobile	Verona	27.50
IN0002	Paoli	Paolo	via dritta	Padova	28.6666
IN0003	Bianchi	Dante	via storta	VERONA	18.345
IN0004	Rossi	Matteo	via nobile	Verona	24.567
IN0005	Verdi	Luca	via nuova	Va	28.6666
IN0006	Nocasa	Caio			

Figura 18: Esempio di base di dati per la clausola BETWEEN

Estrarre tutti gli studenti che hanno matricola tra 'IN0002' e 'IN0004'.

```
SELECT cognome, nome, matricola FROM Studente
WHERE matricola BETWEEN 'IN0002' AND 'IN0004';
```

Il risultato di questa interrogazione è il seguente:

cognome	nome	matricola
Rossi	Marco	IN0001
Paoli	Paolo	IN0002
Bianchi	Dante	IN0003
Rossi	Matteo	IN0004

Figura 19: Risultato dell'istruzione SELECT con clausola WHERE e operatore BETWEEN

2.9.4 IN

Nella clausola WHERE può apparire l'operatore [NOT] IN per testare l'appartenenza di un valore ad un insieme. La sintassi è la seguente:

```
WHERE attributo [NOT] IN ( valore [, ...] );
```

2.9.5 IS NULL

Per verificare se un attributo ha un valore nullo o meno si utilizza il predicato IS NULL che ritorna true se l'attributo ha valore nullo. Oppure IS NOT NULL.

```
WHERE attributo IS [ NOT ] NULL;
```

SQL-2 implementa la logica a 3 valori, cioè un predicato semplice restituisce il valore NULL quando uno qualsiasi dei termini del predicato ha valore nullo. Il predicato IS NULL è una eccezione, restituendo sempre il valore vero oppure false, mai il valore NULL.

2.10 Gestione dei duplicati

Siccome la rimozione delle righe duplicate è un'operazione costosa, SQL le mantiene di default. Per togliere i duplicati si può utilizzare la parola chiave DISTINCT dopo SELECT:

```
SELECT DISTINCT Stipendio
FROM Impiegato
WHERE Cognome = 'Rossi';
```

Il costrutto ALL è invece usato per mantenere i duplicati, ma è superfluo.

Esempio 2.12. Considerando la seguente base di dati:

Persona

CodFiscale	Nome	Cognome	Città
RSSMRA55B21T234J	Mario	Rossi	Verona
BNCCLR69T30H745Z	Carlo	Bianchi	Roma
RSSGNN41A31B344C	Giovanni	Rossi	Verona
RSSPRT75C12F205V	Pietro	Rossi	Milano

Figura 20: Esempio di base di dati per la clausola DISTINCT

Estrarre le città delle persone il cui cognome è Rossi

```
SELECT Citta
  FROM Persona
  WHERE Cognome = 'Rossi'
```

L'output di questa interrogazione è la seguente tabella:

Città
Verona
Verona
Milano

Per togliere i duplicati si può utilizzare la parola chiave DISTINCT dopo SELECT:

```
SELECT DISTINCT Citta
  FROM Persona
  WHERE Cognome = 'Rossi'
```

L'output di questa interrogazione è la seguente tabella:

Città
Verona
Milano

2.11 JOIN

Le interrogazioni con il join possono essere definite in due modi:

1. Senza l'uso di nuova sintassi, e specifica il legame tra le tabelle coinvolte a livello della clausola WHERE.
2. Con una sintassi riservata a livello di clausola FROM

La clausola FROM ammette come argomento:

```
table_name [[AS] name]] [, ...]
```

Si è visto che se sono presenti due o più nomi di tabelle, si esegue il prodotto cartesiano tra tutte le tabelle e lo schema del risultato può contenere tutti gli attributi del prodotto cartesiano. Il prodotto cartesiano di due o più tabelle è un CROSS JOIN.

A partire da SQL-2, esistono altri tipi di JOIN (join_type):

- **INNER JOIN**: restituisce solo le righe che soddisfano la condizione di join
- **LEFT [OUTER] JOIN**: restituisce tutte le righe della tabella di sinistra e le righe della tabella di destra che soddisfano la condizione di join, e NULL per le righe della tabella di destra che non soddisfano la condizione di join
- **RIGHT [OUTER] JOIN**: restituisce tutte le righe della tabella di destra e le righe della tabella di sinistra che soddisfano la condizione di join, e NULL per le righe della tabella di sinistra che non soddisfano la condizione di join
- **FULL [OUTER] JOIN**: restituisce tutte le righe di entrambe le tabelle, e NULL per le righe che non soddisfano la condizione di join

La sintassi generale è:

```
table_name [ NATURAL ] join_type table_name [ON join_condition [,
    ...]].
```

dove `join_condition` è un'espressione booleana che seleziona le tuple del join da aggiungere al risultato. Le tuple selezionate possono essere poi filtrate con la condizione della clausola WHERE

- Formulazione con JOIN:

```
SELECT I. cognome , R. nomeRep
FROM Impiegato AS I
    CROSS JOIN Reparto AS R;
```

- Formulazione con ';':

```
SELECT I. cognome , R. nomeRep
FROM Impiegato AS I, Reparto AS R;
```

cognome	nome
Rossi	Acquisti
Verdi	Acquisti
Rossi	Vendite
Verdi	Vendite

Figura 21: Esempio di base di dati per la clausola JOIN

Esempio 2.13 (Prodotto Cartesiano).

Esempio 2.14. Considerando la seguente base di dati:

Impiegato

Nome	Cognome	Dipart	Ufficio	Stipendio	Città
Mario	Rossi	Amministrazione	10	45	Milano
Carlo	Bianchi	Produzione	20	36	Torino
Giovanni	Verdi	Amministrazione	20	40	Roma
Franco	Neri	Distribuzione	16	45	Napoli
Carlo	Rossi	Direzione	14	80	Milano
Lorenzo	Gialli	Direzione	7	73	Genova
Paola	Rosati	Amministrazione	75	40	Venezia
Marco	Franco	Produzione	20	46	Roma

Dipartimento

Nome	Indirizzo	Città
Amministrazione	Via Tito Livio, 27	Milano
Produzione	P.le Lavater, 3	Torino
Distribuzione	Via Segre, 9	Roma
Direzione	Via Tito Livio, 27	Milano
Ricerca	Via Venosa, 6	Milano

Figura 22: Esempio di base di dati per la clausola JOIN

```
SELECT i.Nome, i.Cognome, d.Citta
  FROM Impiegato as i, Dipartimento as d -- Prodotto cartesiano
  WHERE i.Dipart = d.Nome
```

Il JOIN si può spostare nella clausola FROM, a livello delle tabelle (questa sintassi sarà utilizzata per gli esercizi del corso). La sintassi è la seguente:

```
SELECT ListaAttributi
  FROM Tabella1 JOIN Tabella2 ON ( CondizioneDiJoin )
  [WHERE AltraCondizione]
```

Esempio 2.15. Ad esempio:

```
SELECT i.Nome, d.Citta
FROM Impiegato AS i JOIN Dipartimento as d ON ( i.Dipart = d.
    Nome )
```

Esempio 2.16. Considerando la seguente base di dati:

Persona

CodFiscale	Nome	Cognome	Città
RSSMRA55B21T234J	Mario	Rossi	Verona
BNCCLR69T30H745Z	Carlo	Bianchi	Roma
RSSGNN41A31B344C	Giovanni	Rossi	Verona
RSSPRT75C12F205V	Pietro	Rossi	Milano

Figura 23: Esempio di base di dati per la clausola DISTINCT

Selezionare i padri di persone che guadagnano più di 20

```
SELECT DISTINCT Padre
  FROM Persone JOIN Paternita ON ( Figlio = Nome )
  WHERE Reddito > '20'
```

2.11.1 INNER JOIN

La sintassi è la seguente:

```
table1 [ INNER ] JOIN table2 ON join_condition [, ...]
```

Rappresenta il tradizionale Θ join dell'algebra relazionale. Combina ciascuna riga r_1 di table1 con ciascuna riga di table2 che soddisfa la condizione della clausola ON.

```
SELECT I. cognome , R. nomeRep, R.sede
FROM Impiegato AS I
    INNER JOIN Reparto AS R
    ON I.nomerep = R.nomerep;
```

cognome	nomerep	sede
Rossi	Acquisti	Verona
Verdi	Vendite	Verona

Figura 24: Esempio di base di dati per la clausola INNER JOIN

Esempio 2.17.

2.11.2 LEFT OUTER JOIN

La sintassi è la seguente:

```
table1 LEFT [ OUTER ] JOIN table2 ON join_condition [, ...].
```

Si esegue un INNER JOIN. Poi, per ciascuna riga r_1 di table1 che non soddisfa la condizione con qualsiasi riga di table2, si aggiunge una riga al risultato con i valori di r_1 e assegnando NULL agli altri attributi.

```
INSERT INTO Reparto (nomerep, sede, telefono) VALUES
( 'Finanza', 'Padova', '02 8028888');

SELECT R. nomeRep, I.cognome
FROM Reparto AS R
    LEFT OUTER JOIN Impiegato AS I
    ON I.nomerep = R.nomerep;
```

nomerep	cognome
Acquisti	Rossi
Vendite	Verdi
Finanza	

Figura 25: Esempio di base di dati per la clausola LEFT OUTER JOIN

Esempio 2.18.

ATTENZIONE: Il LEFT [OUTER] JOIN non è simmetrico, cioè con le stesse tabelle si possono ottenere risultati diversi a seconda dell'ordine in cui si scrivono le tabelle.

2.11.3 RIGHT OUTER JOIN

La sintassi è la seguente:

```
table1 RIGHT [ OUTER ] JOIN table2 ON join_condition [, ...].
```

Si esegue un INNER JOIN. Poi, per ciascuna riga r_2 di table2 che non soddisfa la condizione con qualsiasi riga di table1, si aggiunge una riga al risultato con i valori di r_2 e assegnando NULL agli altri attributi.

```
SELECT I. cognome , R. nomeRep
FROM Impiegato I RIGHT OUTER JOIN RepartoR
ON I. nomerep = R. nomerep ;
```

nomerep	cognome
Acquisti	Rossi
Vendite	Verdi
Finanza	

Figura 26: Esempio di base di dati per la clausola RIGHT OUTER JOIN

Esempio 2.19.

2.11.4 FULL OUTER JOIN

La sintassi è la seguente:

```
table1 FULL [ OUTER ] JOIN table2 ON join_condition [, ...].
```

È equivalente a:

```
INNER JOIN + LEFT OUTER JOIN + RIGHT OUTER JOIN
```

Non è equivalente a CROSS JOIN!

2.12 Uso di variabili (alias)

SQL permette di associare un nome alternativo (alias) alle tabelle che compaiono nella clausola FROM e viene usato per far riferimento alla tabella nel contesto dell'interrogazione.

Esempio 2.20. Considerando la seguente base di dati:

Impiegato

Nome	Cognome	Dipart	Ufficio	Stipendio	Città
Mario	Rossi	Amministrazione	10	45	Milano
Carlo	Bianchi	Produzione	20	36	Torino
Giovanni	Verdi	Amministrazione	20	40	Roma
Franco	Neri	Distribuzione	16	45	Napoli
Carlo	Rossi	Direzione	14	80	Milano
Lorenzo	Gialli	Direzione	7	73	Genova
Paola	Rosati	Amministrazione	75	40	Venezia
Marco	Franco	Produzione	20	46	Roma

Figura 27: Esempio di base di dati per l'uso di alias

Estrarre tutti gli impiegati che hanno lo stesso cognome ma nome diverso, del dipartimento Produzione.

```
SELECT i1.Cognome, I1.Nome
  FROM Impiegato i1, Impiegato i2
  WHERE i1.Cognome = i2.Cognome AND
  i1.Nome <> i2.Cognome AND
  i2.Dipart = 'Produzione'
```

Questa interrogazione confronta ciascuna riga di Impiegato con tutte le righe di Impiegato associate al dipartimento Produzione. Si può immaginare che al momento della definizione dell'alias vengano create due copie della tabella Impiegato, ciascuna associata ad una delle due variabili i1 e i2. Le due copie contengono tutte le righe di Impiegato Sulla copia i2 vengono selezionate solo le righe del dipartimento 'Produzione'.

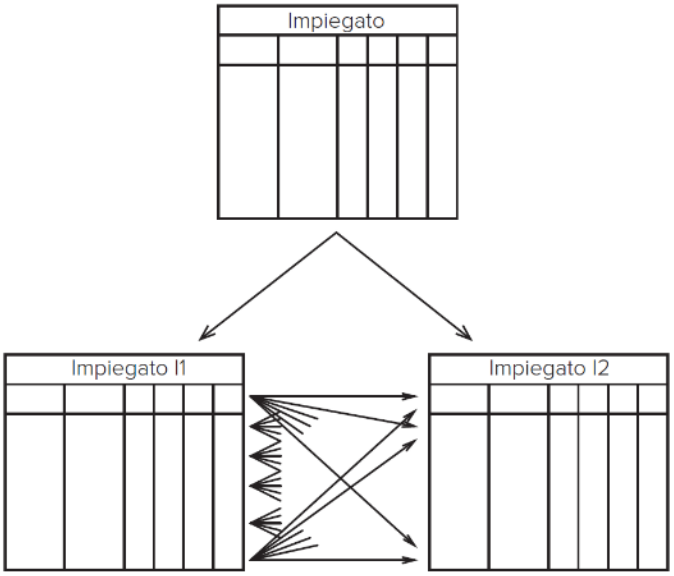


Figura 28: Esempio di base di dati per l'uso di alias e JOIN

2.13 Ordinamento del risultato

Una relazione è un insieme di tuple non ordinato, ma spesso c'è bisogno di costruire un ordine sulle righe. Per questo SQL mette a disposizione la clausola ORDER BY che permette di ordinare le righe del risultato in base ai valori di uno o più attributi. La sintassi è la seguente:

```
ORDER BY attributo [{ ASC | DESC }] [, ... ];
```

Le righe vengono ordinate per il primo attributo nell'elenco, a parità di valore del primo attributo si usa il secondo, e così via in sequenza. Di default l'ordinamento è ASC.

Esempio 2.21. Considerando la seguente base di dati:

Nome	Età	Reddito
Andrea	27	21
Aldo	25	15
Filippo	26	30

Nome	Età	Reddito
Aldo	25	15
Filippo	26	30
Andrea	27	21

Figura 29: Esempio di base di dati per la clausola ORDER BY

Estrarre il nome, l'età ed il reddito delle persone con meno di trenta anni in ordine alfabetico.

```
SELECT Nome, Eta, Reddito
  FROM Persone
  WHERE Eta < '30'
  ORDER BY Eta
```

Esempio 2.22. Considerando la seguente base di dati:

Automobile

Targa	Marca	Modello	NroPatente
KB 574 WW	Fiat	Punto	VR 2030020Y
GA 652 FF	Fiat	Panda	VR 2030020Y
BJ 747 XX	Lancia	Ypsilon	PZ 1012436B
ZB 421 JJ	Fiat	Uno	MI 2020030U

Targa	Marca	Modello	NroPatente
BJ 747 XX	Lancia	Ypsilon	PZ 1012436B
GA 652 FF	Fiat	Panda	VR 2030020Y
KB 574 WW	Fiat	Punto	VR 2030020Y
ZB 421 JJ	Fiat	Uno	MI 2020030U

Figura 30: Esempio di base di dati per la clausola ORDER BY

Estrarre il contenuto della tabella Automobile ordinato in base alla marca (in modo discendente) e al modello.

```
SELECT *
  FROM Automobile
  ORDER BY Marca DESC, Modello
```

2.14 Operatori insiemistici

Gli operatori insiemistici disponibili sono:

```
SELECT
  { UNION | INTERSECT| EXCEPT
  [ALL] SELECT }
```

Gli operatori insiemistici, al contrario del resto del linguaggio, assumono come default di eseguire un'eliminazione dei duplicati, a meno che si usi ALL. Sono sufficienti domini compatibili ed ugual numero di attributi. La corrispondenza degli attributi si basa sulla posizione, e se i nomi degli attributi sono diversi di solito il sistema usa il nome del primo operando.

2.15 Operatori aggregati

Gli operatori aggregati permettono di eseguire operazioni su più righe per ottenere un unico valore. Gli operatori aggregati disponibili sono:

- **COUNT**
- **SUM**
- **AVG**
- **MIN**
- **MAX**

Tutte le condizioni specificate vengono valutate su una tupla alla volta e gli operatori permettono di valutare proprietà che dipendono da insiemi di tuple.

Prima viene eseguita l'interrogazione considerando solo le clausole FROM e WHERE, quindi l'operatore di aggregazione viene applicato alla tabella contenente il risultato.

2.15.1 COUNT

Conta il numero di righe di una tabella o il numero di valori di uno o più attributi. La sintassi è la seguente:

```
COUNT ( <* | [DISTNCT | ALL] ListaAttributi> )
```

dove:

- `*`: conta il numero di righe della tabella
- `DISTINCT ListaAttributi`: conta il numero di valori diversi degli attributi ListaAttributi
- `ALL ListaAttributi`: conta il numero di righe che hanno valori diversi da NULL negli attributi in ListaAttributi

2.15.2 SUM

La sintassi è la seguente:

```
SUM ( [DISTINCT | ALL] AttrExpr )
```

Restituisce la somma dei valori posseduti dall'espressione. Ammette come argomento solo espressioni che rappresentano valori numerici o intervalli di tempo.

- DISTINCT elimina i duplicati
- ALL trascura solo i valori nulli

2.15.3 MAX/MIN

La sintassi è la seguente:

```
<MAX | MIN> ( [DISTINCT | ALL] AttrExpr )
```

Restituisce il massimo/minimo dei valori posseduti dall'espressione. Ammettono come argomento solo espressioni su cui è definito un ordinamento (quindi anche stringhe)

2.15.4 AVG

La sintassi è la seguente:

```
AVG ( [DISTINCT | ALL] AttrExpr )
```

Restituisce la media dei valori posseduti dall'espressione.

- DISTINCT elimina i duplicati
- ALL trascura solo i valori nulli

2.16 Interrogazioni con raggruppamento

Gli operatori di aggregazione possono essere applicati separatamente a sottoinsiemi di righe. La clausola GROUP BY permette di specificare come dividere le tabelle in sottoinsiemi e permette di specificare un insieme di attributi secondo cui raggruppare le righe che possiedono gli stessi valori per questo insieme di attributi.

In una interrogazione che fa uso della clausola GROUP BY può comparire come argomento della SELECT solo un sottoinsieme degli attributi usati nella clausola GROUP BY oppure operatori di aggregazione. Ogni sottoinsieme di righe generato dal GROUP BY deve corrispondere ad un'unica riga nella tabella risultato.

2.16.1 HAVING

È possibile applicare delle selezioni ai gruppi generati tramite la clausola GROUP BY tramite la clausola HAVING. Questa clausola permette di specificare delle condizioni che devono essere soddisfatte dai gruppi generati dopo l'esecuzione del raggruppamento tramite GROUP BY.

Osservazione:

- La clausola WHERE applica delle selezioni su SINGOLE RIGHE, prima dell'eventuale raggruppamento tramite GROUP BY.
- La clausola HAVING applica delle selezioni su GRUPPI, generati a seguito dell'esecuzione del raggruppamento tramite GROUP BY.

2.17 Interrogazioni nidificate

Un'interrogazione nidificata è un'interrogazione che contiene al suo interno altre interrogazioni. La nidificazione può avvenire sia nella clausola SELECT, che nella clausola FROM, che nella clausola WHERE. Il caso tipico è la nidificazione nella clausola WHERE, dove un predicato contiene un confronto tra un valore (espressione su una singola riga) ed il risultato di un'interrogazione SQL.

2.17.1 WHERE

Se in un predicato si confronta un attributo o una espressione che produce un valore singolo con il risultato di un'interrogazione, sorge il problema di disomogeneità dei termini del confronto. I normali operatori di confronto (=, <>, <, >, $\leqslant$, $\geqslant$) possono essere estesi con le parole chiave ANY e ALL:

- **ANY**: la riga soddisfa la condizione se risulta vero il confronto (con l'operatore specificato) tra il valore dell'attributo (o espressione) e ALMENO uno degli elementi restituiti dall'interrogazione.
- **ALL**: la riga soddisfa la condizione se risulta vero il confronto (con l'operatore specificato) tra il valore dell'attributo (o espression) e TUTTI gli elementi restituiti dall'interrogazione.

Per il controllo di appartenenza o esclusione rispetto ad un insieme si possono usare anche gli operatori IN (= ANY) e NOT IN (<> ALL). L'interrogazione può essere eseguita una sola volta ed il risultato ottenuto può essere salvato in una tabella temporanea usata per eseguire ciascun confronto con le righe dalla tabella esterna, però l'interrogazione nidificata fa riferimento al contesto dell'interrogazione che la racchiude attraverso l'uso di una variabile (**passaggio di binding**), quindi l'interrogazione nidificata dovrà essere **valutata separatamente per ciascuna riga** prodotta dalla valutazione della query esterna.

2.17.2 Binding e EXISTS

L'operatore EXISTS riceve come parametro un'interrogazione nidificata e restituisce il valore vero solo se l'interrogazione fornisce un risultato non vuoto. Si può utilizzare soltanto quando c'è un passaggio di binding tra la query esterna e quella interna, cioè quando la query interna fa riferimento ad una variabile che assume valori dalle righe prodotte dalla query esterna.

Esempio 2.23. Estrarre le persone che hanno degli omonimi (ovvero persone con lo stesso nome e cognome, ma codice fiscale diverso)

```
SELECT *
  FROM Persona P
  WHERE EXISTS (
    SELECT *
    FROM Persona P1
    WHERE P1.Nome = P.Nome AND
    P1.Cognome = P.Cognome AND
    P1.CodFiscale <> P.CodFiscale
  );
```

Esempio 2.24. Estrarre le persone che non hanno degli omonimi (ovvero persone con lo stesso nome e cognome, ma codice fiscale diverso)

```
SELECT *
  FROM Persona P
  WHERE NOT EXISTS (
    SELECT *
```

```
    FROM Persona P1
    WHERE P1.Nome = P.Nome AND
    P1.Cognome = P.Cognome AND
    P1.CodFiscale <> P.CodFiscale
  );
```

oppure

```
SELECT *
  FROM Persona P
  WHERE NOT IN (
    SELECT Nome, Cognome
    FROM Persona P1
    WHERE P1.CodFiscale <> P.CodFiscale
  );
```

2.17.3 Interrogazioni nidificate nella clausola SELECT

Le interrogazioni nidificate possono essere specificate anche come elemento della clausola SELECT. È necessario che l'interrogazione nidificata restituisca una sola tupla (riferimenti tramite variabile alla interrogazione esterna). Spesso le interrogazioni nidificate nel SELECT sostituiscono l'uso del JOIN.

Esempio 2.25. Estrarre per ogni cantante il numero di canzoni di cui è autore.

```
SELECT DISTINCT Nome,
  NumCanzoni =
    (
      SELECT COUNT(*)
      FROM Autore A
      WHERE A.Nome = C.Nome
    )
  FROM Cantante C
```

equivale a

```
  SELECT DISTINCT C.Nome,
    COUNT(*) as NumCanzoni
    FROM Cantante C JOIN Autore A
    ON C.Nome = A.Nome
    GROUP BY C.Nome
```

2.17.4 Interrogazioni nidificate nella clausola FROM

Le interrogazioni nidificate possono essere utilizzate nella clausola FROM come sorgente dati su cui eseguire l'interrogazione. È possibile creare una catena di interrogazioni, dove ciascuna interrogazione utilizza il risultato della precedente come una tabella di base.

Esempio 2.26. Estrarre per ogni cantante che è anche autore di canzoni il numero di canzoni di cui è rispettivamente interprete e autore.

```
SELECT DISTINCT C.Nome, C.NumCantate, A.NumScritte
  FROM
```

```sql
  ( SELECT Nome, Count(*) as NumCantate FROM Cantante GROUP BY Nome) C
JOIN
  ( SELECT Nome, Count(*) as NumScritte FROM Autore GROUP BY Nome) A
ON C.Nome = A.Nome
```

Prima vengono eseguite le interrogazioni nella clausola FROM e i risultati di queste vengono usati come normali tabelle.

2.17.5 Esercizi

Esercizio 2.1. Considerando la seguente base di dati:

- Turista(<u>username</u>, cognome, nome, data_nascita, nazionalita)
- Attrazione(<u>codice</u>, nome, descrizione, latitudine, longitudine, indirizzo, citta, prov)
- Recensione(<u>attrazione, turista, timestamp</u>, titolo, testo, voto)

Con i seguenti vincoli di integrità:

- Recensione.attrazione → Attrazione
- Recensione.turista → Turista

Scrivere le seguenti interrogazioni SQL:

1. Trovare tutte le attrazioni della città di Verona il cui nome inizia con la lettera f maiuscola o minuscola

```sql
SELECT *
  FROM Attrazione
  WHERE nome ILIKE 'f%';
```

oppure

```sql
SELECT *
  FROM Attrazione
  WHERE nome LIKE 'f%' OR nome LIKE 'F%';
```

oppure

```sql
SELECT *
  FROM Attrazione
  WHERE LOWER(nome) LIKE 'f%'; -- Converte nome in minuscolo;
```

2. Trovare tutti i turisti che nel 2025 **non** hanno mai rilasciato recensioni con un voto maggiore di 2 per attrazioni fuori dalla provincia di Verona.

```sql
SELECT turista
  FROM Turista T
  EXCEPT -- Le due query devono avere schemi compatibili
  (
    SELECT turista
```

```
      FROM Recensione as R
      JOIN Attrazione A
        ON (A.codice = R.attrazione)
      WHERE EXTRACT(YEAR FROM R.timestamp) = '2025'
        AND R.voto > 2
        AND A.provincia <> 'Verona'
  );
```

Questa query è difficile da cambiare, ad esempio aggiungendo attributi perchè bisognerebbe cambiare sia la query esterna che quella interna. Un'alternativa è la seguente:

```
SELECT T.username, T.nome, T.cognome
  FROM Turista as T
  WHERE T.username
  NOT IN
  (
    SELECT turista
      FROM Recensione as R
      JOIN Attrazione A
        ON (A.codice = R.attrazione)
      WHERE EXTRACT(YEAR FROM R.timestamp) = '2025'
        AND R.voto > 2
        AND A.provincia <> 'Verona'
  );
```

Questa query non richiede che gli schemi siano compatibili, e permette di aggiungere facilmente altri attributi alla query esterna.

3. Trovare tutte le attrazioni che hanno ricevuto nel 2025 un numero di recensioni superiore a quelle ricevute nel 2024.

```
-- Attrazioni con numero recensioni nel 2025
SELECT R1.attrazione, COUNT(*)
  FROM Recensione as R1
  WHERE EXTRACT(YEAR FROM R1.timestamp) = 2025
  GROUP BY R1.attrazione;
  HAVING COUNT(*) > (
    -- Numero recensioni nel 2024
    SELECT COUNT(*)
      FROM Recensione as R2
      WHERE R1.attrazione = R2.attrazione
        AND EXTRACT(YEAR FROM R2.timestamp) = 2024;
  );
```

Nella query annidata si può omettere il GROUP BY perchè grazie al data binding si sta già confrontando le recensioni della stessa attrazione.

4. Trovare la media dei voti massimi rilasciati dagli utenti.

```
SELECT AVG(R1.voto_max)
  FROM (
    SELECT MAX(R2.voto) as voto_max
      FROM Recensione as R2
      GROUP BY R2.turista;
  ) as R1;
```

Non si può scrivere AVG(MAX()) perchè l'operatore aggregato esterno non riceve una lista di valori, ma un singolo attributo che rappresenta il risultato dell'operatore aggregato interno.

Esercizio 2.2. Considerando la seguente base di dati:

- Auto(targa, num_telaio, modello, colore, km, cilindrata, posti, proprietario)
- Parcheggio(numero, comune, indirizzo, coperto)
- Prenotazione(codice, parcheggio, posto, auto, data, ora_inizio, ora_fine, tipo, pagato)

Con i seguenti vincoli di integrità:

- Prenotazione.parcheggio → Parcheggio
- Prenotazione.auto → Auto

Scrivere le seguenti interrogazioni SQL:

1. Trovare tutti i parcheggi coperti di Verona

```
SELECT *
  FROM Parcheggio
  WHERE
    comune = 'Verona';
    AND coperto
```

2. Trovare tutti i parcheggi in cui **non** sono mai state fatte prenotazioni online nel 2024. Per ciascun parcheggio riportare il nome e il numero di prenotazioni complessive.

```
-- Tutti i parcheggi
SELECT P.nome, COUNT(*) as num_prenotazionu
  FROM Parcheggio as P
    JOIN Prenotazione as Pr
    ON (P.codice = Pr.parcheggio)
  WHERE P.codice
    NOT IN (
      -- Parcheggi con almeno una prenotazione online nel
    2024
      SELECT P2.parcheggio
      FROM  Prenotazione as Pr2
      WHERE Pr2.tipo = 'online'
        AND EXTRACT(YEAR FROM data) = '2024'
    )
  GROUP BY P.nome;
```

3. Trovare tutti i parcheggi in cui vengono effettuate più prenotazioni online rispetto a quelle fisihce.

```sql
SELECT Pr.parcheggio
  FROM Prenotazione as Pr
  WHERE Pr.tipo = 'online'
  GROUP BY Pr.parcheggio
    HAVING COUNT(*) > (
      SELECT COUNT(*)
        FROM Prenotazione as Pr2
        WHERE Pr2.tipo = 'fisica'
          AND Pr2.parcheggio = Pr.parcheggio -- Data binding
    );
```

Siccome c'è un data binding tra la query esterna e quella interna, la query interna viene valutata separatamente per ciascun parcheggio restituito dalla query esterna.

2.18 Viste

Le viste permettono di salvare una query che genera una tabella temporanea. La sintassi è la seguente:

```sql
CREATE VIEW nome_vista AS ( query );
```

Si da un nome alla query (nome_vista) che si può usare come se fosse il nome di una tabella.

Esempio 2.27. Considerando la base di dati dell'esercizio 2.2, trovare quanti parcheggi del comune di 'Verona' hanno avuto un numero di prenotazioni nel 2024 superiore al numero medio di prenotazioni complessive nel 2024 dei parcheggi di 'Padova'.

I passi da seguire sono:

1. Calcolare il numero di prenotazioni del 2024 per parcheggio, per ogni parcheggio di 'Verona'
2. Calcolare il numero di prenotazioni del 2024 per parcheggio, per ogni parcheggio di 'Padova'
3. Calcolare la media su 2

```sql
-- Per ogni parcheggio si ottiene:
-- * codice
-- * comune
-- * num prenotazioni
CREATE VIEW NumPrenotazioni as (
  SELECT Pr.codice as parcheggio, P.comune, COUNT(*) as num_pr
  FROM Parcheggio as P
    JOIN Prenotazione as Pr
    ON (P.codice = Pr.parcheggio)
  WHERE EXTRACT(YEAR FROM Pr.data) = '2024'
  GROUP BY Pr.codice, P.comune
);

-- Calcola il numero medio di prenotazioni dei parcheggi
-- di Padova nel 2024
SELECT NP1.parcheggio
```

```sql
FROM NumPrenotazioni as NP1
WHERE NP1.num_pr > (
    SELECT AVG(NP2.num_pr) -- Ritorna 1 valore
    FROM NumPrenotazioni as NP2
    WHERE NP2.comune = 'Padova'
  )
  AND NP1.comune = 'Verona';
  -- Per ogni parcheggio di 'Verona' guardo quali hanno il numero
  -- di prenotazioni > della media di quelli di 'Padova'
```

3 Strutture fisiche e strutture di accesso ai dati

Le basi di dati risiedono in **memoria secondaria**, in quanto sono:

- **Grandi**: non possono essere contenute completamente in memoria centrale
- **Persistenti**: hanno un tempo di vita che non è limitato all'esecuzione dei programmi che le usano

Le caratteristiche della memoria secondaria sono:

- Non è direttamente utilizzabile dai programmi
- I dati sono organizzati **a blocchi**. La grandezza dei blocchi dipende dal sistema operativo.
- Le uniche operazioni possibili sono:
  - Lettura di un intero blocco
  - Scrittura di un intero blocco
- Il costo di tali operazioni è ordini di grandezza maggiore del costo per accedere ai dati in memoria centrale

3.1 Gestore dei buffer

L'interazione tra memoria secondaria e memoria centrale avviene tramite il **gestore del buffer** e consiste nel caricare uno o più **blocchi dalla memoria secondaria** in **pagine** dedicate della **memoria centrale** detta **buffer**. La dimensione delle pagine in memoria centrale è solitamente un multiplo della dimensione dei blocchi in memoria secondaria, ma semplifichiamo assumendo che siano uguali:

$$1\ \mathtt{pagina} \approx 1\ \mathtt{blocco}$$

3.1.1 Buffer

Il buffer è una zona di memoria **condivisa dalle applicazioni** ed è importante gestirlo in modo ottimizzato per massimizzare le prestazioni. Quando un dato viene utilizzato più volte in tempi ravvicinati, il buffer evita di accedere alla memoria secondaria. Il gestore del buffer si occupa di caricare dati dalla memoria secondaria alla memoria centrale e viceversa ottimizzando gli accessi.

3.1.2 Politica di gestione della memoria

Le politiche della gestione della memoria sono:

- **Lettura di un blocco**:
  - Se il blocco **è già presente in una pagina del buffer**, allora non si esegue una lettura su memoria secondaria, ma si restituisce un puntatore alla pagina del buffer.
  - Se il blocco **non è già presente in una pagina del buffer**, si cerca una pagina libera e si carica il blocco nella pagina, restituendo il puntatore alla pagina stessa.
- **Scrittura di un blocco**: In caso di richiesta di scrittura di un blocco precedentemente caricato in una pagina del buffer, il gestore del buffer può decidere di **differire la scrittura** su memoria secondaria in un secondo momento.

Queste politiche si basano sul **principio di località**, cioè i dati referenziati di recente hanno maggiore probabilità di essere referenziati nuovamente nel futuro.

Inoltre una nota **legge empirica** afferma che il 20% dei dati viene referenziato l'80% delle volte.

3.1.3 Gestione delle pagine

Le pagine del buffer possono essere rappresentate tramite una tabella in cui

- Per ogni **pagina del buffer** si memorizza il blocco B contenuto indicando il file e il numero di blocco (o offset). L indica che la pagina è libera
- Per ogni pagina del buffer si memorizza un insieme di **variabili di stato** tra cui si trovano sicuramente:
  - Un **Contatore I** (B_I): indica il **numero di transazioni** che stanno usando quella pagina
  - Un **Bit di stato J** (B_J): indica se la pagina è stata modificata (1) o no (0)

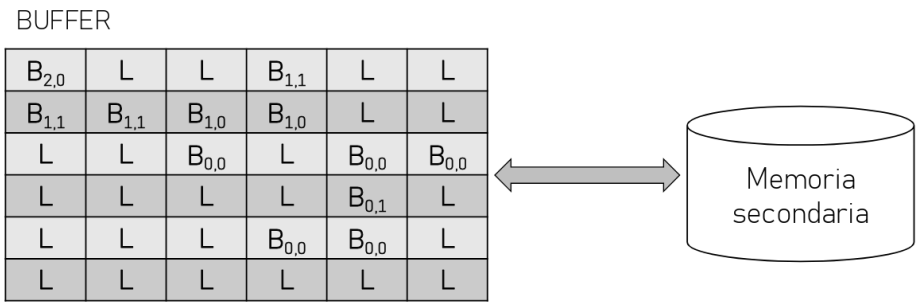


Figura 31: Gestione delle pagine del buffer

3.1.4 Primitive per la gestione del buffer

Le primitive per la gestione del buffer sono:

- `fix`: viene usata dalle transazioni per richiedere l'accesso ad un blocco e restituisce un puntatore alla pagina contenente il blocco richiesto. I passi di questa primitiva sono:

  1. Il blocco richiesto viene cercato nelle pagine del buffer, in caso sia presente, si restituisce un puntatore alla pagina contenente il blocco
  2. Altrimenti, viene scelta una pagina libera P (pagina L oppure con contatore I = zero). Si sceglie la pagina in base a criteri diversi:
     - **LRU** (Least Recently Used): La pagina usata meno di recente
     - **FIFO** (First In First Out): La pagina caricata da più tempo
     - Se il bit di stato J di P è a 1 (cioè la pagina è stata modificata), P viene salvata in memoria secondaria (primitiva flush).
     - Se J = 0 non si fa nulla

     Si carica il blocco in P aggiornando file e numero di blocco corrispondenti.
  3. Se non esistono pagine libere, il gestore del buffer può adottare due politiche:
     - **Steal**: "ruba" una pagina ad un'altra transazione (primitiva flush).
     - **No steal**: sospende la transazione inserendola in una coda di attesa.

     Se una pagina si libera (contatore I = zero) il gestore si comporterà come al punto precedente.
  4. Quando una transazione accede ad una pagina per la prima volta il contatore si incrementa.

- `unfix`: viene usata dalle transazioni per indicare che la transazione ha terminato di usare il blocco
  - L'effetto è il decremento del contatore I che indica l'uso della pagina.

- `setDirty`: viene usata dalle transazioni per indicare al gestore del buffer che il blocco della pagina è stato modificato
  - L'effetto è la modifica del bit di stato J a 1

- `flush`: viene usata dal gestore del buffer per salvare blocchi sulla memoria secondaria in modo **asincrono**.
  - L'effetto è la liberazione delle pagine "dirty", quindi il bit di stato J viene impostato a 0

- `force`: viene usata per salvare in memoria secondaria in modo **sincrono** il blocco contenuto nella pagina (primitiva usata dal modulo Gestore dell'Affidabilità).
  - L'effetto è il salvataggio in memoria secondaria del blocco e il bit di stato J posto a zero

3.2 Gestore dell'affidabilità

Siccome il gestore del buffer differisce le operazioni di scrittura, questo potrebbe essere pericoloso, quindi il gestore dell'affidabilità si occupa di garantire:

- **Atomicità**: che le transazioni non vengano lasciate incomplete con alcune operazioni eseguite ed altre no
- **Persistenza**: che gli effetti di ciascuna transazione conclusa con un commit siano mantenuti in modo permanente a prescindere dai guasti o dalle scritture differite

3.2.1 File di log

Il funzionamento si basa sull'utilizzo del **log**, cioè un archivio persistente su cui vengono registrate tutte le operazioni eseguite dal DBMS e per il corretto funzionamento il gestore deve disporre di un dispositivo di memoria **stabile**, cioè resistente ai guasti.

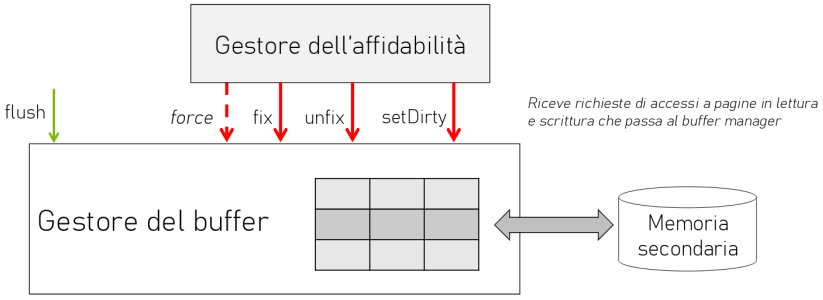


Figura 32: Gestore dell'affidabilità nell'architettura di un DBMS

Il file di log viene memorizzato in memoria stabile e al suo interno vengono memorizzati:

- **Record di transazione**: cioè informazioni sulla transazione:
  - $B(T)$: Begin di una transazione T
  - $C(T)$: Commit di una transazione T
  - $A(T)$: Abort di una transazione T
  - Insert, delete, update eseguite da una transazione T su un oggetto O, ad esempio una riga in una tabella
    - $I(T, O, AS)$: Insert di un oggetto O da parte di una transazione T con il suo **after state** AS (stato dell'oggetto dopo l'operazione)
    - $D(T, O, BS)$: Delete di un oggetto O da parte di una transazione T con il suo **before state** BS (stato dell'oggetto prima dell'operazione)
    - $U(T, O, BS, AS)$: Update di un oggetto O da parte di una transazione T con il suo **before state** BS e il suo **after state** AS

L'obiettivo è quello di consentire in caso di ripristino le seguenti operazioni:

- **UNDO**: per disfare un'azione su un oggetto O è sufficiente ricopiare in O il valore BS; l'insert/delete viene disfatto cancellando/inserendo O;
- **REDO**: per rifare un'azione su un oggetto O è sufficiente ricopiare in O il valore AS; l'insert/delete viene rifatto inserendo/cancellando O

Gli inserimenti controllano sempre l'esistenza dell'oggetto O prima di eseguire l'operazione, quindi non vengono inseriti i duplicati

Per queste operazioni vale la **proprietà di idempotenza**, cioè:

$$\text{UNDO}(\text{UNDO}(A)) = \text{UNDO}(A)$$
$$\text{REDO}(\text{REDO}(A)) = \text{REDO}(A)$$

- **Record di sistema**: cioè informazioni sulle operazioni eseguite dal DBMS:
  - **Operazione di DUMP** della base di dati: record di `DUMP`, copia completa e consistente della base di dati
  - **Operazione di CheckPoint**: record $\text{CK}(T_1, \ldots, T_n)$ indica che all'esecuzione del CheckPoint le transazioni attive erano $T_1, \ldots, T_n$
    * Sono elencate le transazioni che non hanno ancora espresso la loro scelta relativamente all'esito finale.
    * Operazione svolta periodicamente dal gestore dell'affidabilità per garantire che gli effetti delle transazioni che hanno eseguito il commit siano registrati in modo permanente.
    * Prevede i seguenti passi:
      1. Sospensione delle operazioni di scrittura, commit e abort delle transazioni
      2. Esecuzione della primitiva `force` sulle pagine "dirty" di transazioni che hanno eseguito il commit
      3. Scrittura sincrona (primitiva `force`) sul file di LOG del record di CheckPoint con gli identificatori delle transazioni attive
      4. Ripresa delle operazioni di scrittura, commit e abort delle transazioni

3.2.2 Regole di scrittura sul file di log

Il gestore dell'affidabilità deve rispettare le seguenti regole di scrittura sul file di log:

- **Regola WAL** (Write-Ahead Logging): i record di log devono essere scritti sul LOG prima dell'esecuzione delle corrispondenti operazioni sulla base di dati (garantisce la possibilità di fare sempre UNDO)
- **Regola di Commit-Precedenza**: i record di log devono essere scritti sul LOG prima dell'esecuzione del COMMIT della transazione (garantisce la possibilità di fare sempre REDO)

Tali regole consentono inoltre di salvare i blocchi delle pagine "dirty" in modo totalmente asincrono rispetto al commit delle transazioni.

3.2.3 Guasti

Una transazione T sceglie in modo atomico l'esito di COMMIT nel momento in cui scrive nel file di log in modo sincrono (primitiva `force`) il suo record di COMMIT $C(T)$. Se avviene:

- Un guasto prima della scrittura del record di COMMIT, allora si effettua un UNDO
- Un guasto dopo la scrittura del record di COMMIT, allora si effettua un REDO

Esistono due tipologie di guasto:

- **Guasto di sistema**: è la perdita del contenuto della memoria centrale. Si risolve con la **ripresa a caldo** seguendo i seguenti passi:
  1. Si accede al file di LOG e si ripercorre all'indietro il LOG fino al più recente record di CheckPoint
  2. Si inizializzano:
     - L'insieme UNDO con tutte le transazioni da disfare

       $$\mathrm{UNDO} = \text{tutte le transazioni presenti nel record di CheckPoint}$$

     - L'insieme REDO con tutte le transazioni da rifare

       $$\mathrm{REDO} = \varnothing$$

  3. Si ripercorre in avanti il LOG e:
     - Per ogni record $B(T)$ incontrato si aggiunge T all'insieme UNDO
     - Per ogni record $C(T)$ incontrato si sposta T dall'insieme UNDO all'insieme REDO
  4. Si ripercorre all'indietro il LOG disfacendo le operazioni eseguite dalle transazioni UNDO risalendo fino alla prima azione della transazione più vecchia.
  5. Si ripercorre in avanti il LOG per rifare le operazioni eseguite dalle transazioni REDO
- **Guasto di dispositivo**: è la perdita di parte o tutto il contenuto della base di dati in memoria secondaria. Si risolve con la ripresa a freddo seguendo i seguenti passi:
  1. Si accede al DUMP della base di dati e si ricopia selettivamente la parte deteriorata della base di dati
  2. Si accede al LOG risalendo al record di DUMP
  3. Si ripercorre in avanti il LOG rieseguendo tutte le operazioni relative alla perte deteriorata comprese le azioni di commit e abort
  4. Si applica una ripresa a caldo

Esercizio 3.1. Data la seguente situazione del file di LOG:

$B(T1), B(T2), U(T2, O1, B1, A1), I(T1, O2, A2), B(T3), C(T1), B(T4),$
$U(T3, O2, B3, A3), U(T4, O3, B4, A4), CK(T2, T3, T4), C(T4), B(T5),$
$U(T3, O3, B5, A5), U(T5, O4, B6, A6), D(T3, O5, B7), A(T3), C(T5),$
$I(T2, O6, A8)$ guasto

Che passi svolge la ripresa a caldo?

1. Risale il LOG fino all'ultimo record di CheckPoint $CK(T2, T3, T4)$
2. Inizializza UNDO e REDO:
   - $UNDO = \{T_2, T_3, T_4\}$. Viene inizializzato con le transazioni del CheckPoint
   - $REDO = \varnothing$
3. Ripercorre in avanti il LOG dal CheckPoint fino al guasto e si aggiungono o si spostano le transazioni tra UNDO e REDO:
   - $UNDO = \{T_2, T_3, \cancel{T_4}, \cancel{T_5}\}$
   - $REDO = \{T_4, T_5\}$

   Quindi la situazione dopo questo passo è:
   - $UNDO = \{T_2, T_3\}$
   - $REDO = \{T_4, T_5\}$
4. Si ripercorre all'indietro il LOG disfacendo le operazioni eseguite dalle transazioni UNDO risalendo fino alla prima azione della transazione più vecchia.

   $DELETE(O6)$ (disfare insert)
   $O5 := B7$ (disfare delete)
   $O3 := B5$ (disfare update)
   $O2 := B3$ (disfare update)
   $O1 := B1$ (disfare update)
5. Si ripercorre in avanti il LOG per rifare le operazioni eseguite dalle transazioni REDO:

   $O3 := A4$ (rifare update)
   $O4 := A6$ (rifare update)

Esercizio 3.2. Data la seguente situazione del file di LOG:

$B(T1), B(T2), B(T3), I(T1, O1, A1), D(T2, O2, B2), B(T4),$
$U(T4, O3, B3, A3), U(T1, O4, B4, A4), C(T2), CK(T1, T3, T4), B(T5), B(T6),$
$U(T5, O5, B5, A5), A(T3), CK(T1, T4, T5, T6), B(T7), C(T4),$
$U(T7, O6, B6, A6), U(T6, O3, B7, A7), B(T8), A(T7),$ guasto

Che passi svolge la ripresa a caldo?

1. Si risale il LOG fino all'ultimo record di CheckPoint $CK(T1, T4, T5, T6)$
2. Inizializza UNDO e REDO:
   - $UNDO = \{T1, T4, T5, T6\}$
   - $REDO = \varnothing$
3. Ripercorre in avanti il LOG dal CheckPoint fino al guasto e si aggiungono o si spostano le transazioni tra UNDO e REDO:
   - $UNDO = \{T1, \cancel{T4}, T5, T6, T7, T8\}$
   - $REDO = \{T4\}$

   Quindi la situazione dopo questo passo è:
   - $UNDO = \{T1, T5, T6, T7, T8\}$
   - $REDO = \{T4\}$
4. Si ripercorre all'indietro il LOG disfacendo le operazioni eseguite dalle transazioni UNDO risalendo fino alla prima azione della transazione più vecchia:

$$\begin{aligned} &O3 := B7 \quad \text{(disfare update)} \\ &O6 := B6 \quad \text{(disfare update)} \\ &O5 := B5 \quad \text{(disfare update)} \\ &O4 := B4 \quad \text{(disfare update)} \\ &DELETE(O1) \quad \text{(disfare insert)} \end{aligned}$$

5. Si ripercorre in avanti il LOG per rifare le operazioni eseguite dalle transazioni REDO:

$$O3 := A3 \quad \text{(rifare update)}$$

Si nota che ci sono due operazioni che riguardano l'oggetto O3 e l'ultima operazione è quella di T4 che ha eseguito il commit, quindi è quella che viene rifatta. Quella di T6 non viene applicata.

3.3 Gestore dei metodi di accesso

Il gestore dei metodi di accesso è il modulo che esegue il piano di esecuzione prodotto dall'ottimizzatore e produce sequenze di richieste di accesso ai blocchi della base

di dati presenti in memoria secondaria. Le richieste vengono inviate al gestore del buffer che si occupa di caricare i blocchi necessari in pagine di memoria centrale.

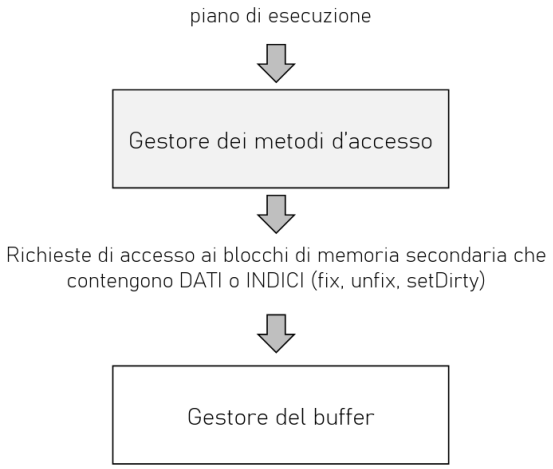


Figura 33: Gestore dei metodi di accesso nell'architettura di un DBMS

3.3.1 Metodi di accesso

I metodi di accesso sono moduli **software** che implementano gli algoritmi di **accesso e manipolazione dei dati** organizzati in specifiche strutture fisiche.

Esempio 3.1. Alcuni esempi di metodi di accesso sono:

- Scansione sequenziale
- Accesso via indice
- Ordinamento
- Varie implementazioni del join

Ogni metodo di accesso ai dati conosce:

- **L'organizzazione delle tuple** (o dei record di indice) nei blocchi DATI (o INDICE) salvati in memoria secondaria. Come una tabella (o indice) viene organizzata in pagine di memoria secondaria.
- **L'organizzazione fisica interna delle pagine** sia quando contengono DATI, sia quando contengono strutture fisiche di accesso (o INDICI).

3.3.2 Organizzazione di una pagina

In una pagina sono presenti due tipi di informazioni:

- **Informazioni utili**: dati veri e propri, cioè le tuple di una tabella
- **Informazioni di controllo**: consentono di accedere alle informazioni utili, ad esempio il dizionario, bit di parità, altre informazioni del file system, ecc.

La struttura di una pagina è la seguente:

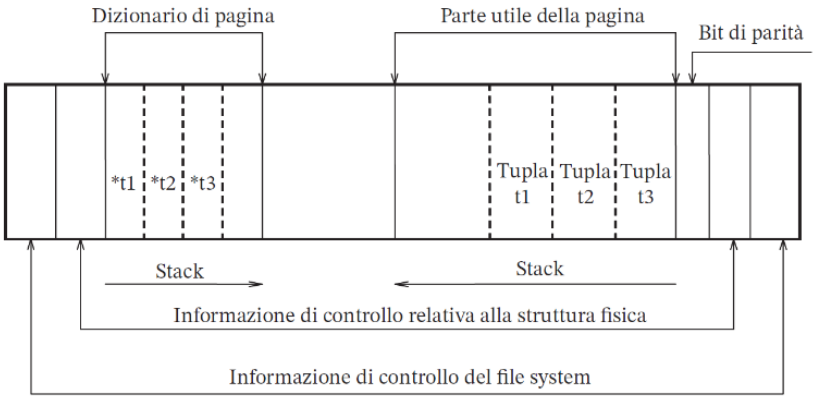


Figura 34: Organizzazione di una pagina

e i diversi campi sono:

- **Block header e block trailer**: Ogni pagina, siccome coincide con un blocco di memoria di massa, ha una parte iniziale ed una parte finale che contengono informazioni di controllo utilizzate dal file system

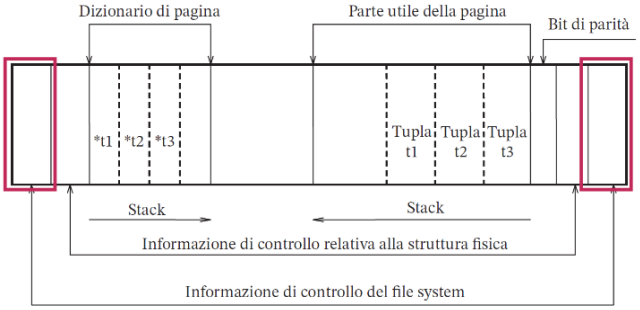


Figura 35: Block header e block trailer

- **Page header e page trailer**: Ogni pagina, siccome contiene dati gestiti dal DBMS, ha una parte iniziale ed una finale contenenti l'informazione di controllo relativa alla specifica struttura fisica. Ad esempio: identificatore dell'oggetto (tabella, indice, dizionario dei dati, ecc.) contenuto nella pagina, puntatori a pagine successive o precedenti nella struttura dati, numero di dati utili elementari (tuple) contenuti nella pagina e quantità di memoria libera (contigua o non contigua) disponibile nella pagina.

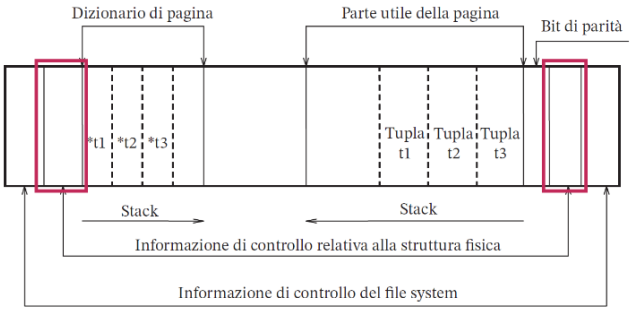


Figura 36: Page header e page trailer

- – **Dizionario di pagina**: Contiene puntatori a ciascun dato utile elementare contenuto della pagina. Se le tuple sono a:
  - ∗ **Lunghezza fissa**, allora il dizionario di pagina non è necessario perchè si deve soltanto memorizzare la dimensione delle tuple e l'offset dal punto iniziale
  - ∗ **Lunghezza variabile** (valori nulli o stringhe), allora il dizionario memorizza l'offset di ogni tupla presente nel blocco e di ogni attributo di ogni tupla

  Molti gestori non consentono la separazione di una tupla su più pagine.
  – **Parte utile**: Contiene i dati veri e propri. Il dizionario di pagina e la parte utile crescono come stack contrapposti, lasciando libera la parte centrale in uno spazio contiguo.
  – **Bit di parità**: Verifica che l'informazione contenuta nella pagina sia valida

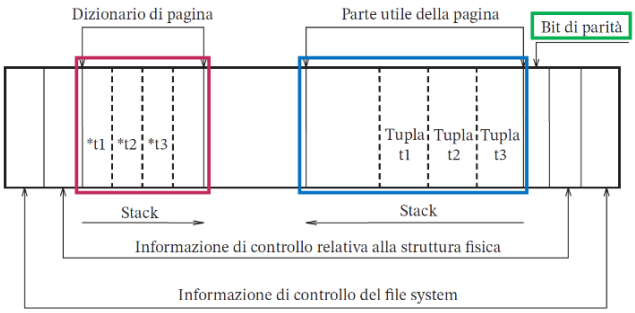


Figura 37: Dizionario di pagina, parte utile e bit di parità

3.3.3 Operazioni sulle pagine

Le operazioni che si possono eseguire sulle pagine sono:

- **Inserimento ed aggiornamento di una tupla**:
  – Se esiste spazio contiguo sufficiente: inserimento semplice.

  - Se non esiste spazio contiguo ma esiste spazio sufficiente: riorganizzare lo spazio ed eseguire un inserimento semplice (operazione in memoria centrale). Operazione aggiuntiva: **riorganizzazione** della pagina.
  - Se non esiste spazio sufficiente: interazione col file system per allocare nuovi blocchi per il file.

- **Cancellazione**: è sempre possibile anche senza riorganizzare il contenuto nella pagina.
- **Accesso ad una tupla** particolare tramite il valore di chiave oppure l'offset nel dizionario.
  - Accesso sequenziale a più tuple.
- **Accesso ad un attributo di una tupla**: identificato in base all'offset e alla lunghezza del campo dopo aver identificato la tupla tramite la chiave o il suo offset.

3.3.4 Strutture primarie per l'organizzazione di file

La struttura primaria di un file stabilisce il criterio secondo il quale sono disposte le tuple all'interno del file presente in memoria secondaria. Le strutture primarie più comuni sono:

- **Sequenziali**: disposizione consecutiva delle tuple che si basa su un criterio specifico, come l'ordine di inserimento
- **Ad accesso calcolato** (hash): collocano le tuple in posizioni determinate sulla base dell'esecuzione di un algoritmo di hashing
- **Ad albero**: utilizzate soprattutto come strutture secondarie (indici)

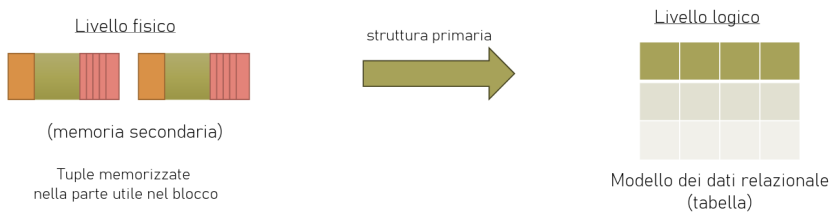


Figura 38: Strutture primarie per l'organizzazione di file

3.3.5 Struttura sequenziale

Un file è costituito da blocchi **logicamente** consecutivi e le tuple vengono inserite nei blocchi rispettando una sequenza:

- **Struttura sequenziale disordinata** (o seriale): le tuple sono disposte in base al loro ordine di inserimento, senza alcun criterio specifico. Questa è la struttura più semplice e più diffusa.

  L'efficienza di ogni operazione è la seguente:

- **Inserimento**: operazione molto efficiente, è sufficiente mantenere un riferimento all'ultimo blocco e richiede un solo accesso in memoria secondaria. Si avrà un costo maggiore in caso di verifica dei vincoli di integrità.
- **Ricerca**: operazione non efficiente, richiede ogni volta una scansione sequenziale che consideri tutti i record, con un costo lineare nel numero di blocchi. La ricerca si può fare usando strutture secondarie (indici).
- **Eliminazione**: facile una volta individuate le tuple coinvolte. Si marcano le tuple come "cancellate" senza riorganizzazione locale.
- **Modifica**: facile una volta individuate le tuple coinvolte. Si procede in loco se possibile, cioè quando non c'è stata nessuna modifica alla dimensione della tupla, oppure si procede con una cancellazione e un inserimento in fondo al file.

Eliminazioni e modifiche possono portare ad un uso non ottimale dello spazio siccome si devono effettuare riorganizzazioni periodiche.

- **Struttura sequenziale ad array**: le tuple sono disposte come in un array, e la loro posizione dipende dal valore assunto in ciascuna tupla da un campo detto **indice**. Questa struttura è utilizzabile soltanto quando le tuple di una tabella hanno una dimensione fissa, infatti non è quasi masi usata nei DBMS reali perchè difficilmente sono soddisfatte le condizioni per la sua applicazione.

  Ad ogni file viene assegnato un numero n di blocchi contigui e ciascun blocco è dotato di un numero m di posizioni disponibili per le tuple (quindi è un array bidimensionale $n \times m$). Ogni tupla è dotata di un valore numerico i che rappresenta l'indice della tupla in i-esima posizione nell'array.

  - **Inserimento**:
    * Iniziale: l'indice i viene ottenuto incrementando un contatore
    * Successivo: in una posizione libera oppure alla fine del file
  - **Cancellazioni**: creano delle posizioni libere

- **Struttura sequenziale ordinata**: è un file sequenziale dove le tuple sono ordinate secondo una **chiave di ordinamento** (diversa dalla chiave primaria). Questa struttura rende efficienti le operazioni che hanno bisogno dell'ordinamento utilizzato, ad esempio:

  - Elenco ordinato
  - Range query
  - Operazioni aggregate

  In caso di aggiornamento è necessario mantenere l'ordinamento, quindi servono periodiche riorganizzazioni.

  - **Inserimento**: I passi da eseguire sono:
    1. Individuare il blocco B che contiene **la tupla che precede**, nell'ordine della chiave, la tupla da inserire
    2. Se B contiene spazio sufficiente per la nuova tupla allora si inserisce la tupla in B

3. Altrimenti si aggiunge un nuovo blocco, detto **overflow page**, alla struttura e si inserisce la tupla nel nuovo blocco e si aggiusta la catena di puntatori.

– **Scansione sequenziale ordinata secondo la chiave**: è sufficiente seguire i puntatori

– **Cancellazione di una tupla**: I passi da seguire sono:

1. Individuare il blocco B che contiene la tupla da cancellare
2. Cancellare la tupla da B
3. Aggiustare la catena di puntatori

– **Riorganizzazione**: si assegnano le tuple ai blocchi in base ad opportuni coefficienti di riempimento, riaggiustando i puntatori.

Esempio 3.2. Il seguente esempio mostra una struttura sequenziale ordinata secondo la chiave `Filiale`:

chiave primaria

	Filiale	Conto	Cliente	Saldo
Blocco 1	A	102	Rossi	1000
	B	110	Rossi	3020
	B	198	Bianchi	500
Blocco 2	E	17	Neri	345
	E	102	Verdi	1200
	E	113	Bianchi	200
	H	53	Neri	120
	F	78	Verdi	3400

Figura 39: Esempio di struttura sequenziale ordinata secondo la chiave `Filiale`

3.3.6 Indici

Per aumentare le prestazioni degli accessi alle tuple memorizzate nelle strutture fisiche (**file sequenziale**), si introducono strutture ausiliarie dette strutture di accesso ai dati, o **indici**. Queste strutture velocizzano l'accesso casuale via chiave di ricerca, cioè un insieme di attributi utilizzati nell'indice della ricerca. Ci sono due tipi di indici su file sequenziali:

- **Indice primario**: La chiave di ordinamento del file sequenziale **coincide** con la chiave di ricerca dell'indice.

  Ogni record dell'indice primario contiene una coppia $\langle v_i, p_i \rangle$, dove:

  – v_i è il valore della chiave di ricerca

  – p_i è il puntatore al primo record nel file sequenziale con chiave v_i

  Esistono due varianti dell'indice primario:

- **Indice denso**: per ogni occorrenza della chiave presente nel file esiste un corrispondente record nell'indice
- **Indice sparso**: solo per alcune occorrenze della chiave presenti nel file esiste un corrispondente record nell'indice, tipicamente una per blocco

Le operazioni che si possono eseguire su un indice primario sono:

- **Ricerca** di una tupla con chiave di ricerca K.
  - Se l'indice è **denso** K è presente nell'indice.
    1. Scansione sequenziale dell'indice per trovare il record (K, p_k)
    2. Accesso al file attraverso il puntatore p_k

    Il costo è 1 accesso indice + 1 accesso blocco dati

Esempio 3.3. Un esempio di ricerca con indice primario denso è il seguente: (ricerca dei conti della filiale B)

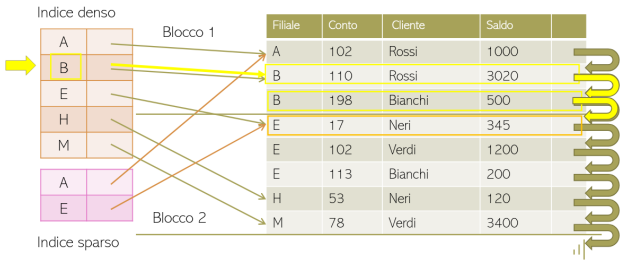


Figura 40: Esempio di ricerca con indice primario denso

  - Se l'indice è **sparso** K potrebbe non essere presente nell'indice
    1. Scansione sequenziale dell'indice fino al record (K', p_k), dove K' è il valore più grande che sia minore o uguale a K
    2. Accesso al file attraverso il puntatore p_k e scansione del file (blocco corrente) per trovare le tuple con chiave K

    Il costo è 1 accesso indice + 1 accesso blocco dati

Esempio 3.4. Un esempio di ricerca con indice primario sparso è il seguente: (ricerca dei conti della filiale B)

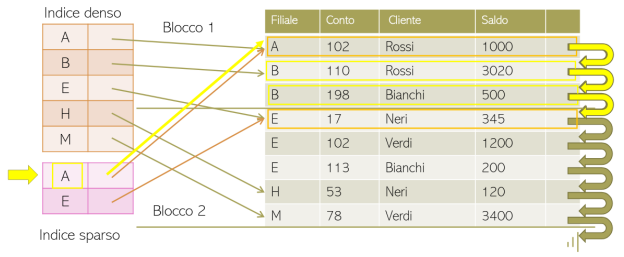


Figura 41: Esempio di ricerca con indice primario sparso

    * **Inserimento** di un record nell'indice. Si effettuano gli stessi passi dell'inserimento in un file sequenziale (3.3.5), solo che nel blocco di memoria secondaria al posto di tuple ci sono record di indice.
      · Se l'indice è **denso** l'inserimento nell'indice avviene solo se la tupla inserita nel file ha un valore di chiave K che non è già presente
      · Se l'indice è **sparso** l'inserimento avviene solo quando, per effetto dell'inserimento di una nuova tupla, si aggiunge un blocco dati alla struttura; in tutti gli altri casi l'indice rimane invariato

    Gli indici sono costosi, quindi bisogna fare un compromesso sul numero di indici da mantenere e il numero di attributi ad accesso sequenziale.
    * **Cancellazione** di un record nell'indice. Si effettuano gli stessi passi della cancellazione in un file sequenziale (3.3.5).
      · Se l'indice è **denso** la cancellazione nell'indice avviene solo se la tupla cancellata nel file è l'ultima tupla con valore di chiave K
      · Se l'indice è **sparso** la cancellazione nell'indice avviene solo quando K è presente nell'indice e il corrispondente blocco viene eliminato; altrimenti, se il blocco sopravvive, va sostituito K nel record dell'indice con il primo valore K' presente nel blocco.

- **Indice secondario**: La chiave di ordinamento e la chiave di ricerca sono diverse.

  Ogni record dell'indice secondario contiene una coppia $\langle v_i, p_i \rangle$, dove:

  – v_i è il valore della chiave di ricerca
  – p_i è il puntatore al **bucket di puntatori** che individuano nel file sequenziale tutte le tuple con chiave v_i

  Gli indici secondari **sono sempre densi** perchè non si può sfruttare l'ordinamento delle tuple.

  Le operazioni che si possono eseguire su un indice secondario sono:

  – **Ricerca** di una tupla con chiave di ricerca K.
    1. Scansione sequenziale dell'indice per trovare il record (K, p_k)
    2. Accesso al bucket B di puntatori attraverso il puntatore p_k
    3. Accesso al file attraverso i puntatori del bucket B

    Il costo è 1 accesso indice + 1 accesso al bucket + n accessi alle pagine dati dove n è il numero di puntatori nel bucket B

    Esempio 3.5. Un esempio di ricerca con indice secondario è il seguente: (ricerca dei conti di "Verdi")

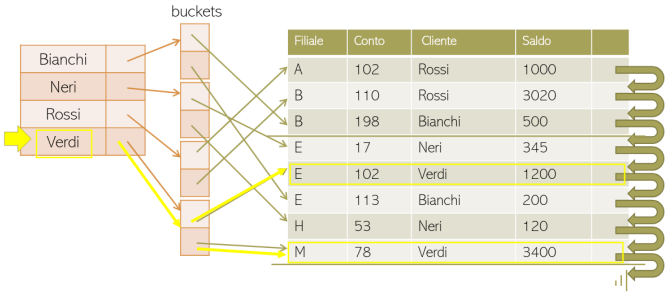


Figura 42: Esempio di ricerca con indice secondario

– **Inserimento e cancellazione**: si effettuano gli stessi passi dell'inserimento e della cancellazione in un indice primario denso, con in più l'aggiornamento dei bucket.

3.3.7 Struttura ad albero

Quando l'indice aumenta di dimensioni, non si può più gestire in memoria centrale, di conseguenza deve essere gestito in memoria secondaria. Per migliorare le prestazioni degli accessi ad un indice di grandi dimensioni, si utilizza il B+-Tree, una struttura ad albero con le seguenti caratteristiche:

- Ogni **nodo viene memorizzato in una pagina/blocco di memoria secondaria**.
- I legami tra i nodi sono **puntatori a pagine di memoria secondaria**.
- Sono presenti **molti figli** e di conseguenza ha **pochi livelli**.
- L'albero è **bilanciato**, cioè la lunghezza dei percorsi che collegano la radice ai nodi foglia è costante
- Inserimenti e cancellazioni non alterano le prestazioni dell'accesso ai dati: l'albero si mantiene bilanciato

Struttura

La struttura di un B+-Tree con **fan-out** $= n$, cioè con n figli per ogni nodo, è la seguente:

- **Nodo foglia**:

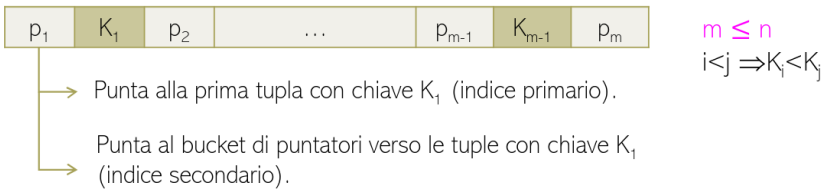


Figura 43: Struttura di un nodo foglia di un B+-Tree con fan-out $= n$

Può contenere fino a $n-1$ valori ordinati di chiave di ricerca e fino a n puntatori ai dati. Ogni nodo foglia ha un **vincolo di ordinamento**, cioè

- Le chiavi dentro ad un nodo foglia sono ordinate e
- I nodi foglia sono ordinati tra di loro. Dati due nodi foglia L_i e L_j con $i < j$, allora:

$$\forall k_t \in L_i; \forall k_s \in L_j; k_t < k_s$$

Il puntatore p_m del nodo L_i punta al nodo foglia L_{i+1} se esiste:

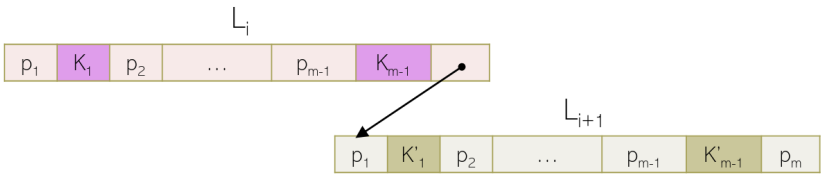


Figura 44: Puntatore p_m del nodo L_i che punta al nodo L_{i+1}

Una variante del B+-Tree è quella in cui i nodi foglia contengono direttamente le tuple. In questo caso si ha una struttura fisica integrata dati/indice.

- **Nodo intermedio**: È una sequenza di $m' \leqslant n$ valori ordinati di chiave (può contenere fino a n puntatori a nodo). Ogni chiave K_i è seguita da un puntatore p_i:

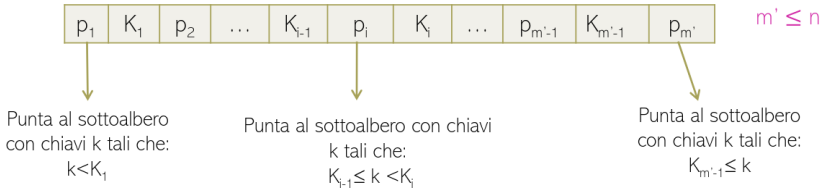


Figura 45: Struttura di un nodo intermedio di un B+-Tree con fan-out $= n$

Il fan-out è un parametro che fornisce un limite massimo, ma bisogna anche fornire un limite minimo per evitare che i nodi siano troppo vuoti, questi limiti sono detti **vincoli di riempimento**:

- **Vincoli per i nodi foglia**: Ogni nodo foglia può contenere un numero di valori di chiave (#chiavi) vincolato come:

$$\left\lceil \frac{n-1}{2} \right\rceil \leqslant \#\text{chiavi} \leqslant n-1$$

- **Vincoli per i nodi intermedi**: Ogni nodo intermedio può contenere un numero di puntatori (#puntatori) vincolato come:

$$\left\lceil \frac{n}{2} \right\rceil \leqslant \#\text{puntatori} \leqslant n$$

Esempio 3.6. Consideriamo ad esempio un B+-Tree con fan-out = 4 con contenuto:

$$A, B, D, E, F, G, L, M, N, P$$

Il fan-out ci dice che per:

- I nodi foglia i vincoli di riempimento sono:

$$2 \leqslant \#chiavi \leqslant 3$$

- I nodi intermedi i vincoli di riempimento sono:

$$2 \leqslant \#puntatori \leqslant 4$$

- La radice è l'unico nodo per cui i vincoli di riempimento **minimi** non si applicano

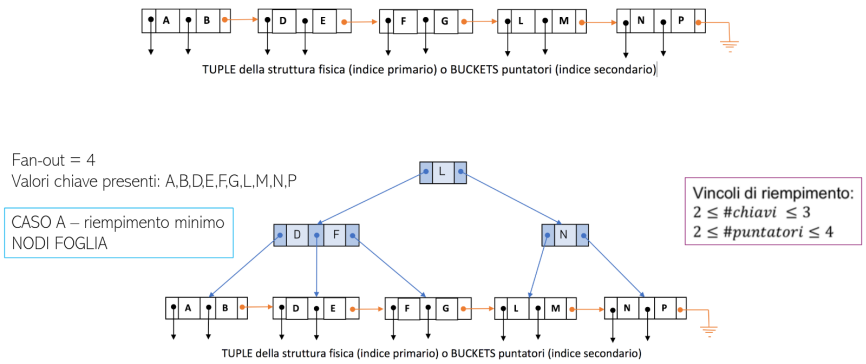


Figura 46: Esempio di B+-Tree con vincolo di riempimento minimo

La radice L punta:

- A sinistra: al sottoalbero con chiave K < "L"
- A destra: al sottoalbero con chiave K $\geqslant$ "L"

I figli puntano alle tuple con chiave uguale a quella del nodo foglia

In questo caso l'albero per chiavi desiderate rispetta il vincolo di riempimento minimo. Per rispettare il vincolo di riempimento massimo bisogna fare un altro albero:

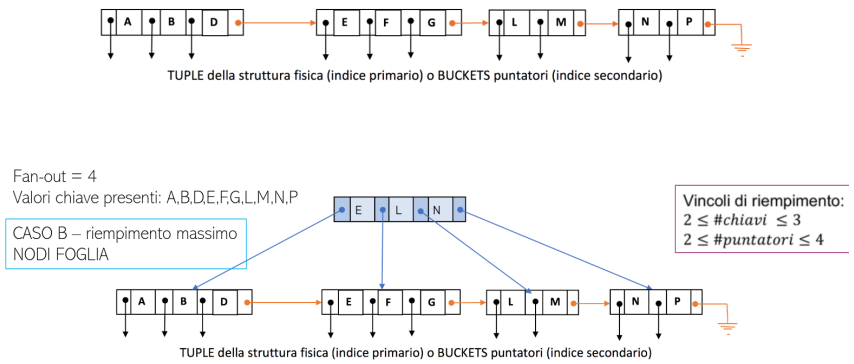


Figura 47: Esempio di B+-Tree con vincolo di riempimento massimo

Le etichette per la radice sono assegnate secondo:

- Sinistra: Chiavi comprese tra: "E" $\leqslant$ K $<$ "L"
- Centro: Chiavi comprese tra "L" $\leqslant$ K $<$ "N"
- Destra: Chiavi K $\geqslant$ "N"

Ricerca con chiave K

Per effettuare la ricerca di una tupla con chiave di ricerca K si effettuano i seguenti passi:

1. Cercare nel **nodo radice** il più piccolo valore di chiave maggiore di K
   - Se tale valore esiste (supponiamo sia K_i) allora seguire il puntatore p_i

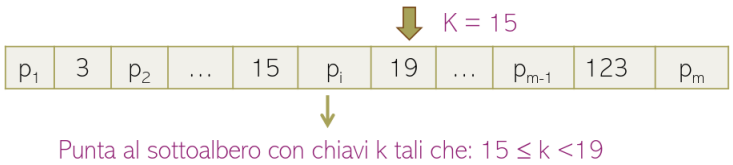


Figura 48: Ricerca di una tupla con chiave di ricerca K in un B+-Tree

   - Se tale valore non esiste seguire il puntatore p_m

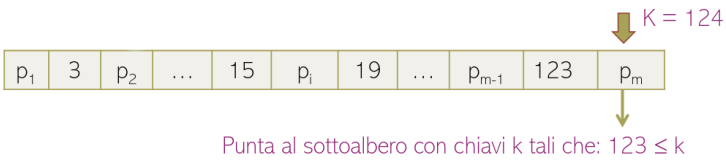


Figura 49: Ricerca di una tupla con chiave di ricerca K in un B+-Tree quando il più piccolo valore di chiave maggiore di K non esiste

2. Se il nodo raggiunto è un **nodo foglia** cercare il valore K al suo interno e seguire il corrispondente puntatore verso le tuple, se non esistono riprendere il passo 1

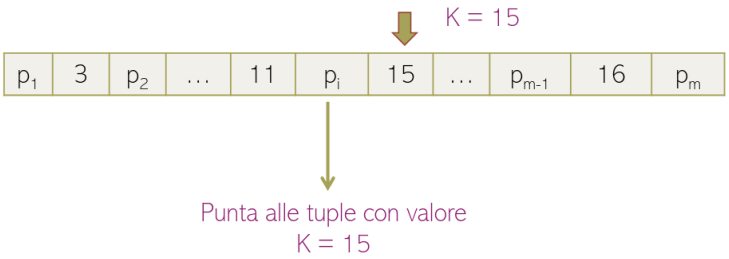


Figura 50: Ricerca di una tupla con chiave di ricerca K in un B+-Tree quando si raggiunge un nodo foglia

Esempio 3.7. Ad esempio ricerchiamo la tupla con chiave di ricerca K = "G":

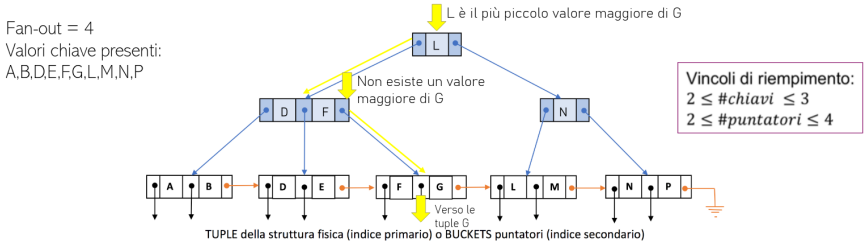


Figura 51: Esempio di ricerca di una tupla con chiave di ricerca K in un B+-Tree

Il costo della ricerca dipende dalla profondità dell'albero in termini di **accessi alla memoria secondaria**, dove ogni nodo rappresenta un blocco. Il costo in una struttura ad albero è pari alla profondità dell'albero, che nel B+-Tree è indipendente dal percorso ed è funzione del fan-out n e del numero di valori chiave presenti nell'albero #valoriChiave

$$\text{prof}_{\text{B+-Tree}} \leqslant \underbrace{1}_{\text{Radice}} + \underbrace{\log_{\lceil \frac{n}{2} \rceil} \left(\frac{\overbrace{\#\text{valoriChiave}}^{\text{Nodi foglia massimi}}}{\lceil \frac{n-1}{2} \rceil} \right)}_{\#\text{ Livelli}}$$

Inserimento

Per inserire una tupla in un B+-Tree bisogna mantenere i vincoli di riempimento, quindi vengono eseguiti i seguenti passi:

1. **Ricerca** del nodo foglia NF dove K va inserito

2.
   - Se K è presente in NF:
     - Se è **indice primario**: non si fa nessuna azione
     - Se è **indice secondario**: si aggiornano i bucket di puntatori
   - Altrimenti si inserisce K in NF rispettando l'ordine:
     - Se è **indice primario**: si inserisce K e si aggiorna il puntatore alla prima tupla
     - Se è **indice secondario**: si inserisce K e si crea il bucket

   Se all'inserimento di K viene violato il vincolo di riempimento massimo, bisogna eseguire uno **split** del nodo NF in due nodi foglia. Lo split viene eseguito se in NF esistono già n valori di chiave e bisogna eseguire i seguenti passi:

   (a) Creare due nodi foglia NF_1 e NF_2
   (b) Inserire i primi $\lceil \frac{n-1}{2} \rceil$ valori di chiave in NF_1
   (c) Inserire i restanti in NF_2
   (d) Aggiornare il padre, cioè se il padre viola il vincolo di riempimento massimo bisogna propagare lo split e quindi creare una nuova radice/livello se necessario

Esempio 3.8. Un esempio di split di un nodo foglia con fan-out = 5 è il seguente:

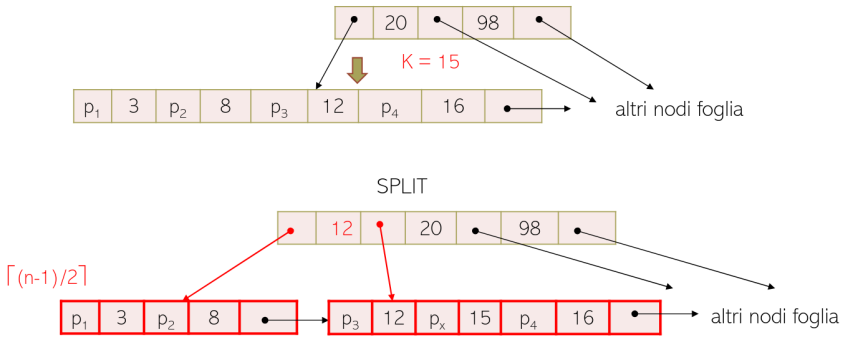


Figura 52: Esempio di split di un nodo foglia in un B+-Tree con fan-out = 5

Se si aggiungesse la chiave 15 tra il 12 e il 16 si violerebbe il vincolo di riempimento massimo: $\#chiavi = 5$, quindi si esegue lo split del nodo foglia.

Cancellazione

Per cancellare una tupla in un B+-Tree con chiavee K bisogna mantenere i vincoli di riempimento, quindi vengono eseguiti i seguenti passi:

1. **Ricercare** il nodo foglia NF che contiene K
2. Cancellare K e il suo puntatore. Potrebbe violare il vincolo di riempimento minimo, in quel caso bisogna fare il **merge** di NF con un fratello. Per fare

il merge, se nel nodo da unire esistono $\left\lceil \frac{n-1}{2} \right\rceil - 1$ valori chiave, si procede come segue:

(a) Individuare il nodo fratello

(b) Si genera un unico nodo foglia

(c) Si aggiorna il padre

- Se non c'è un nodo foglia fratello che unito rispetta il vincolo di riempimento massimo, si fa prima il merge e poi lo split
- Se anche il padre viola il vincolo di riempimento massimo, si propaga il merge fino ad arrivare alla radice

Esempio 3.9. Un esempio di merge di un nodo foglia con fan-out = 5, in cui si vuole eliminare la tupla con chiave K = "3" è il seguente:

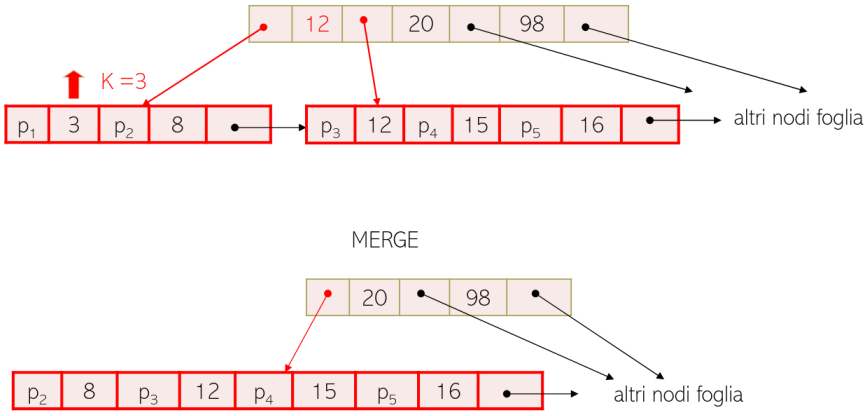


Figura 53: Esempio di merge di un nodo foglia in un B+-Tree con fan-out = 5

Se si eliminasse il nodo con chiave 3 si violerebbe il vincolo di riempimento minimo, quindi si deve fare il merge con il nodo fretello.

3.3.8 Struttura ad accesso calcolato (hash)

Questa struttura garantisce un **accesso associativo** alle tuple, cioè la locazione fisica dei dati dipende dal valore assunto dalla chiave. Si potrebbe utilizzare un array indicizzato dalla chiave, ma questo non è pratico perchè i valori della chiave sono tanti, quindi si utilizza una **funzione di hash** che mappa i valori di chiave in indirizzi di memorizzazione delle tuple:

$$h: K \rightarrow B$$

dove:

- K è il dominio delle chiavi

- B è il dominio degli indirizzi di memorizzazione

Il vantaggio di questa struttura è che il costo della ricerca è costante e uguale ad un accesso alla memoria secondaria. Lo svantaggio è quello delle **collisioni**, cioè valori di chiavi diversi possono produrre lo stesso indirizzo. Una funzione di hash **buona** minimizza il numero di collisioni.

Definizione 3.1. Il fattore di riempimento definisce la frazione di spazio fisico mediamente utilizzata da un blocco

Siano:

- T il numero di tuple
- F il fattore blocco, cioè il numero di tuple che ci stanno in un blocco
- f il fattore di riempimento

Si ha che il numero di blocchi è definito come:

$$\#B = \left\lceil \frac{T}{F \cdot f} \right\rceil$$

Il DBMS alloca un numero di bucket di puntatori uguale al numero di blocchi $\#B$. Si definiscono due funzioni:

- Funzione di **folding**: trasforma i valori chiave in numeri interi positivi

$$f : K \to \mathbb{Z}^+$$

- Funzione di **hashing**:

$$h : \mathbb{Z}^+ \to B$$

Esempio 3.10. Un esempio di tabella di hash è il seguente:

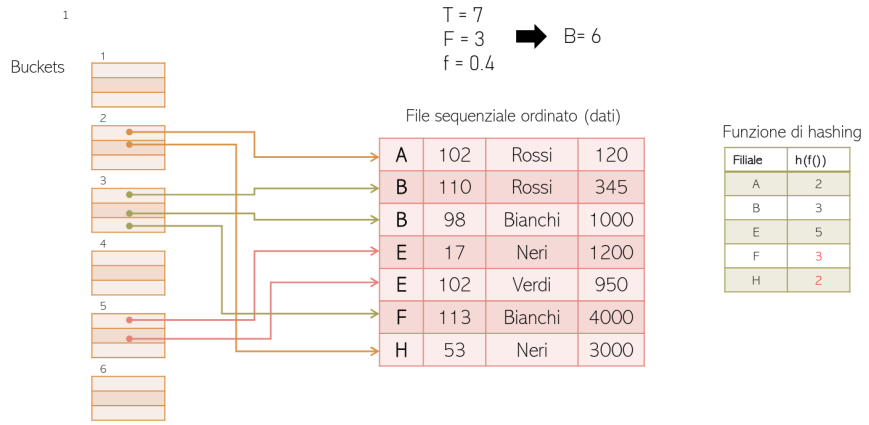


Figura 54: Esempio di creazione di una tabella di hash

Ricerca con chiave K

Per effettuare la ricerca di una tupla con chiave di ricerca K si effettuano i seguenti passi:

1. Calcolare $b = h(f(K))$ (costo zero)
2. Accedere al bucket b (costo: 1 accesso alla pagina di memoria secondaria)
3. Accedere alle n tuple attraverso i puntatori del bucket (costo: m accessi a pagine di memoria secondaria, con $m \leqslant n$)

L'inserimento e la cancellazione hanno costo simile alla ricerca.

3.3.9 Collisioni

La struttura ad accesso calcolato funziona se i buckets conservano un basso coefficiente di riempimento. Infatti il problema delle strutture ad accesso calcolato è la gestione delle collisioni.

Definizione 3.2 (Collisione)**.** Si verifica quando, dati due valori di chiave K_1 e K_2 con $K_1 \neq K_2$, risulta:

$$h(f(K_1)) = h(f(K_2))$$

La probabilità che uno stesso bucket riceva t chiavi su n inserimenti è:

$$p(t) = \binom{n}{t} \left(\frac{1}{B}\right)^t \left(1 - \frac{1}{B}\right)^{n-t}$$

dove B è il numero totale di bucket. La probabilità di avere più di F collisioni, con F il fattore blocco, cioè il numero di puntatori nel bucket, è:

$$p_K = 1 - \sum_{i=0}^{F} p(i)$$

Gestione delle collisioni

Un numero eccessivo di collisioni porta alla saturazione del bucket corrispondente. Per gestirlo si introducono dei **bucket di overflow** collegati al bucket originale, che contengono i puntatori alle tuple che hanno causato la saturazione del bucket originale. Le prestazioni della ricerca peggiorano perchè bisogna accedere ai bucket di overflow.

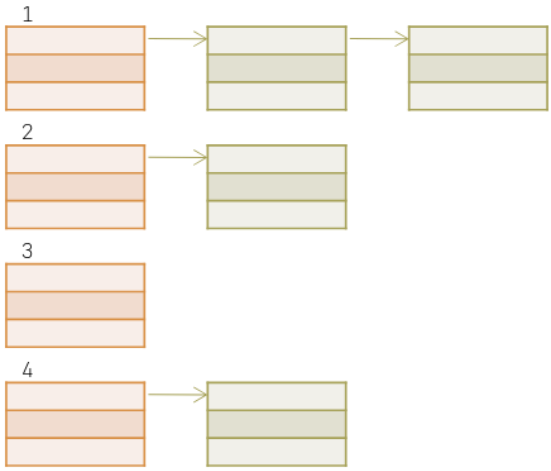


Figura 55: Esempio di bucket di overflow per gestire le collisioni in una struttura ad accesso calcolato

4 Transazioni concorrenti

Il componente dell'architettura del DBMS che si occupa di gestire le transazioni concorrenti è il **gestore della concorrenza** che riceve richieste di accesso ai dati e decide se autorizzarle o meno, eventualmente riordinandol o ritardandole, come se fosse uno scheduler.

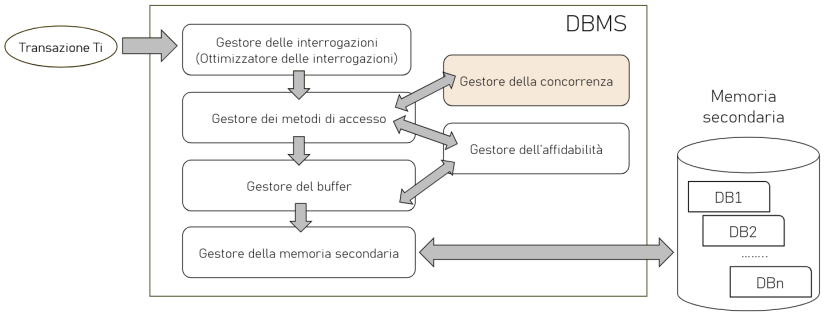


Figura 56: Gestore della concorrenza nell'architettura di un DBMS

L'unità di misura per caratterizzare il carico applicativo di un DBMS è il numero di transazioni al secondo (TPS, Transactions Per Second). Per gestire un carico elevato, tipico delle applicazioni gestionali (100 o 1000 tps), un DBMS deve eseguire le transazioni in modo concorrente.

4.1 Anomalie

L'esecuzione concorrente di transazioni senza controllo può generare **anomalie** o problemi di correttezza ed è quindi necessario introdurre dei meccanismi di controllo nell'esecuzione delle transazioni per evitare queste anomalie.

Definizione utile 4.1 (Notazione)**.** Verrà utilizzata la seguente notazione:

- t_i indica una transazione
- $r_i(x)$ è un operazione di lettura eseguita dalla transazione t_i sulla risorsa x
- $w_i(x)$ è un operazione di scrittura eseguita dalla transazione t_i sulla risorsa x
- `bot` indica l'inizio di una transazione (begin of transaction)
- `commit` indica l'applicazione definitiva degli effetti di una transazione (commit)
- `eot` indica la fine di una transazione (end of transaction)

4.1.1 Perdita di aggiornamento

Questa anomalia si verifica quando gli effetti di una transazione concorrente sono persi. Si considerino le seguenti transazioni:

$$t_1 : \mathrm{bot};\ r_1(x);\ x = x + 1;\ w_1(x);\ \mathrm{commit};\ \mathrm{eot}$$
$$t_2 : \mathrm{bot};\ r_2(x);\ x = x + 1;\ w_2(x);\ \mathrm{commit};\ \mathrm{eot}$$

L'esecuzione concorrente di t_1 e t_2 può portare alla perdita di aggiornamento, come si può vedere dalla seguente simulazione:

Operazioni eseguite da t1	Segmento di memoria centrale di t1	Memoria secondaria [buffer]	Segmento di memoria centrale di t2	Operazioni eseguite da t2
Stato iniziale		x = 2	Stato iniziale	
bot		2		
$r_1(x)$	2 ←	[2]		
x=x+1	3	[2]		bot
	3	[2] →	2	$r_2(x)$
	3	[2]	3	x=x+1
	3	[3] ←	3	$w_2(x)$
	3		3	commit
	3			eot
$w_1(x)$	3 →	[3]		
commit	3	3		
eot		3		

Il valore finale di x è 3, mentre una qualsiasi esecuzione seriale avrebbe prodotto per x il valore 4!

Figura 57: Esempio di perdita di aggiornamento

4.1.2 Lettura inconsistente

Questa anomalia si verifica quando accessi successivi ad uno stesso dato, all'interno di una transazione, ritornano valori diversi. Si considerino le seguenti transazioni:

$$t_1 : \mathrm{bot};\ r_1(x);\ r_1'(x);\ \mathrm{commit};\ \mathrm{eot}$$
$$t_2 : \mathrm{bot};\ r_2(x);\ x = x + 1;\ w_2(x);\ \mathrm{commit};\ \mathrm{eot}$$

L'esecuzione concorrente di t_1 e t_2 può portare alla lettura inconsistente, come si può vedere dalla seguente simulazione:

Operazioni eseguite da t1	Segmento di memoria centrale di t1	Memoria secondaria [buffer]	Segmento di memoria centrale di t2	Operazioni eseguite da t2
Stato iniziale		x = 2	Stato iniziale	
bot		2		
$r_1(x)$	2 ←	[2]		
	2	[2]		bot
	2	[2] →	2	$r_2(x)$
	2	[2]	3	x=x+1
	2	[3]	3	$w_2(x)$
	2	3	3	commit
	2	3		eot
$r_1'(x)$	3 →	[3]		
commit	3	3		
eot		3 ←		

Il valore di x è diventato 3, ma la transazione t_1 non ha modificato x!

Figura 58: Esempio di lettura inconsistente

4.1.3 Lettura sporca

Questa anomalia si verifica quando viene letto un dato che rappresenta uno stato intermedio nell'evoluzione di una transazione. Si considerino le seguenti transazioni:

$$t_1 : \mathrm{bot};\ r_1(x);\ \mathrm{commit};\ \mathrm{eot}$$
$$t_2 : \mathrm{bot};\ r_2(x);\ x = x + 1;\ w_2(x);\ \ldots \mathrm{abort};$$

L'esecuzione concorrente di t_1 e t_2 può portare alla lettura sporca, come si può vedere dalla seguente simulazione:

Operazioni eseguite da T1	Segmento di memoria centrale di T1	Memoria secondaria [buffer]	Segmento di memoria centrale di T2	Operazioni eseguite da T2
Stato iniziale		x = 2	Stato iniziale	
bot		2		
		2		bot
		[2] →	2	$r_2(x)$
		[2]	3	x=x+1
		[3] ←	3	$w_2(x)$
$r_1(x)$	3 ←	[3]	3	abort
commit	3	2		
eot		2		

Il valore di x letto da t_1 è 3, ma la transazione t_2 non ha ancora eseguito il commit!

Figura 59: Esempio di lettura sporca

4.1.4 Aggiornamento fantasma

Questa anomalia si verifica quando avviene l'osservazione di uno stato intermedio che non soddisfa i vincoli di integrità. Si considerino le seguenti transazioni:

$$\text{Risorse} : x, y, z$$
$$\text{Vincoli} : x + y + z = 100$$

$$t_1 : \text{bot};\ r_1(y);\ r_1(x);\ r_1(z);\ s = x + y + z;\ \text{eot}$$
$$t_2 : \text{bot};\ r_2(y);\ y = y + 10;\ r_2(z);\ z = z - 10;\ w_2(y);\ w_2(z);\ \text{commit};\ \text{eot}$$

L'esecuzione concorrente di t_1 e t_2 può portare all'aggiornamento fantasma, come si può vedere dalla seguente simulazione:

Operazioni eseguite da T1	Segmento di memoria centrale di T1	Memoria secondaria [buffer]	Segmento di memoria centrale di T2	Operazioni eseguite da T2
Stato iniziale		(x,y,z)=20,20,60	Stato iniziale	
bot		20,20,60		
r1 (y)	-,20,- ←	20,[20],60		
	-,20,-	20,[20],60		bot
	-,20,-	20,[20],60 →	-,20,-	r2 (y)
	-,20,-	20,[20],60	-,30,-	y = y + 10
	-,20,-	20,[20],[60] →	-,30,60	r2 (z)
	-,20,-	20,[20],[60]	-,30,50	z = z - 10
	-,20,-	20,[30],[60] ←	-,30,50	w2 (y)
	-,20,-	20,[30],[50] ←	-,30,50	w2 (z)
	-,20,-	20,30,50	-,30,50	commit
	-,20,-	20,30,50		eot
r1 (x)	20,20,- ←	[20],30,50		
r1 (z)	20,20,50 ←	[20],30,[50]		
s=x-y-z	20,20,50 ←	[20],30,[50]		
eot		20,30,50		

S = 90 → VIOLAZIONE VINCOLO

Figura 60: Esempio di aggiornamento fantasma

4.1.5 Inserimento fantasma

Questa anomalia si verifica quando avvengono valutazioni diverse di valori aggregati a causa di inserimenti intermedi. La transazione che valuta un valore aggregato relativo a tutti gli elementi che soddisfano un predicato di selezione (ad esempio il voto medio degli studenti del terzo anno) può subire l'anomalia di inserimento

fantasma se il valore aggregato viene valutato due volte e tra la prima e la seconda valutazione viene inserito un nuovo studente, quindi i due valori medi letti potrebbero essere diversi. Questa anomalia non si può risolvere facendo riferimento solo ai dati già presenti nella base di dati, ma è necessario tener presente che potrebbero esserci delle nuove tuple che compaiono *improvvisamente*.

4.2 Schedule

Uno schedule è una sequenza di operazioni di lettura e scrittura eseguite su risorse della base di dati da diverse transazioni concorrenti che rappresenta una possibile esecuzione concorrente di diverse transazioni:

$$S : r_1(x);\ r_2(z);\ w_1(x);\ w_2(z);$$

Per evitare anomalie si stabilisce di accettare **solo** gli schedule equivalenti ad uno schedule seriale.

4.2.1 Schedule seriale

Lo schedule seriale è uno schedule dove le operazioni di ogni transazione compaiono in sequenza **senza essere inframezzate** da operazioni di altre transazioni.

Esempio 4.1. Consideriamo i seguenti esempi di schedule:

- $S_1 : r_1(x);\ r_2(x);\ w_2(x);\ w_1(x);$ **non** è seriale
- $S_2 : r_1(x);\ w_1(x);\ r_2(x);\ w_2(x);$ **è** seriale

Definizione 4.1 (Schedule serializzabile)**.** Lo schedule serializzabile è uno schedule equivalente ad uno schedule seriale. Due schedule sono equivalenti se **producono gli stessi effetti** sulla base di dati.

4.3 Tecniche di gestione della concorrenza

Per le seguenti tecniche di gestione della concorrenza, si assume l'ipotesi di **commit-proiezione**, cioè si suppone che le transazioni abbiano esito noto. Quindi si tolgono dagli schedule tutte le operazioni delle transazioni che non vanno a buon fine.

4.3.1 View-equivalenza

Definizione 4.2 (Relazione LEGGE_DA)**.** Dato uno schedule S si dice che un'operazione di lettura $r_i(x)$, che compare in S, `LEGGE_DA` un'operazione di scrittura $w_j(x)$, che compare in S, se $w_j(x)$ **precede** $r_i(x)$ in S e non vi è alcuna operazione di scrittura $w_k(x)$ tra le due. Con $j \neq i$.

Esempio 4.2. Un esempio di utilizzo della relazione LEGGE_DA è il seguente:

- $S_1 : r_1(x);\ r_2(x);\ w_2(x);\ w_1(x);\ \texttt{LEGGE_DA}(S_1) = \varnothing$
- $S_2 : r_1(x);\ w_1(x);\ r_2(x);\ w_2(x);\ \texttt{LEGGE_DA}(S_2) = \{(r_2(x), w_1(x))\}$
- $S_3 : r_1(x);\ w_1(x);\ w_2(x);\ r_2(x);\ \texttt{LEGGE_DA}(S_3) = \varnothing$

Definizione 4.3 (SCRITTURE_FINALI)**.** Dato uno schedule S si dice che un'operazione di scrittura $w_i(x)$, che compare in S, è una **SCRITTURA FINALE** se è l'ultima operazione di scrittura della risorsa x in S.

Esempio 4.3. Un esempio di utilizzo della relazione SCRITTURE_FINALI è il seguente:

- $S_1 : r_1(x);\ r_2(x);\ w_2(x);\ w_3(y);\ w_1(x);$
  $\texttt{SCRITTURE_FINALI}(S_1) = \{w_3(y), w_1(x)\}$
- $S_2 : r_1(x);\ w_1(x);\ r_2(x);\ w_2(x);\ w_3(y);$
  $\texttt{SCRITTURE_FINALI}(S_2) = \{w_2(x), w_3(y)\}$

Definizione 4.4 (View-equivalenza)**.** Due schedule S_1 e S_2 sono view-equivalenti ($S_1 \approx_V S_2$) se possiedono le stesse relazioni LEGGE_DA e SCRITTURE_FINALI.

Definizione 4.5 (View-serializzabilità)**.** Uno schedule S è view-serializzabile (VSR) se esiste uno schedule seriale S' tale che $S \approx_V S'$.

Attenzione: gli schedule seriali S' si generano considerando tutte le possibili permutazioni **delle transazioni** (**non** delle operazioni delle transazioni) che compaiono in S. Inoltre **non si deve** cambiare l'ordine delle operazioni di ciascuna transazione perchè ciò significherebbe modificare la transazione.

Gli schedule corrispondenti alle anomalie di:

- Perdita di aggiornamento
- Letture inconsistenti
- Aggiornamenti fantasma

non sono view-serializzabili. Inoltre l'algoritmo per il test di view-equivalenza tra due schedule è di complessità lineare, mentre l'algoritmo per il test di view-serializzabilità di uno schedule è di complessità esponenziale.

In conclusione si è osservato che la view-serializzabilità richiede l'ipotesi di commit-proiezione, e algoritmi di complessità troppo elevata, quindi non è applicabile nei sistemi reali.

Esercizio 4.1 (Esercizio perdita di aggiornamento)**.** Consideriamo le seguenti transazioni:

$$t_1 : r_1(x);\ w_1(x)$$
$$t_2 : r_2(x);\ w_2(x)$$

Eseguite con lo schedule:

$$S_{pa} = r_1(x);\ r_2(x);\ w_2(x);\ w_1(x);$$

Questo schedule è view serializzabile?

Calcoliamo le relazioni:

- $\texttt{LEGGE_DA}(S_{pa}) = \varnothing$
- $\texttt{SCRITTURE_FINALI}(S_{pa}) = \{w_1(x)\}$

Bisogna generare tutti i possibili schedule seriali per determinare se esiste uno schedule seriale S' tale che $S_{pa} \approx_V S'$. Gli schedule seriali sono:

- $S_{12} = r_1(x);\ w_1(x);\ r_2(x);\ w_2(x);$
  - $\texttt{LEGGE_DA}(S_{12}) = \{(r_2(x), w_1(x))\}$
  - $\texttt{SCRITTURE_FINALI}(S_{12}) = \{w_1(x)\}$

  Osserviamo che $S_{pa} \not\approx_V S_{12}$
- $S_{21} = r_2(x);\ w_2(x);\ r_1(x);\ w_1(x);$
  - $\texttt{LEGGE_DA}(S_{21}) = \{(r_1(x), w_2(x))\}$
  - $\texttt{SCRITTURE_FINALI}(S_{21}) = \{w_1(x)\}$

  Osserviamo che $S_{pa} \not\approx_V S_{21}$

Avendo provato per tutte le possibili permutazioni si può dire che lo schedule S_{pa} **non è view-serializzabile**.

Esercizio 4.2 (Esercizio di lettura inconsistente)**.** Consideriamo le seguenti transazioni:

$$t_1 : r_1(x);\ r'_1(x)$$
$$t_2 : r_2(x);\ w_2(x)$$

Eseguite con lo schedule:

$$S_{li} = r_1(x);\ r_2(x);\ w_2(x);\ r'_1(x);$$

Questo schedule è view serializzabile?

Calcoliamo le relazioni:

- $\texttt{LEGGE_DA}(S_{li}) = \{(r'_1(x), w_2(x))\}$
- $\texttt{SCRITTURE_FINALI}(S_{li}) = \{w_2(x)\}$

Bisogna generare tutti i possibili schedule seriali per determinare se esiste uno schedule seriale S' tale che $S_{li} \approx_V S'$. Gli schedule seriali sono:

- $S_{12} = r_1(x);\ r'_1(x);\ r_2(x);\ w_2(x);$
  - $\texttt{LEGGE_DA}(S_{12}) = \varnothing$
  - $\texttt{SCRITTURE_FINALI}(S_{12}) = \{w_2(x)\}$

  Osserviamo che $S_{li} \not\approx_V S_{12}$
- $S_{21} = r_2(x);\ w_2(x);\ r_1(x);\ r'_1(x);$
  - $\texttt{LEGGE_DA}(S_{21}) = \{(r_1(x), w_2(x)), (r'_1(x), w_2(x))\}$
  - $\texttt{SCRITTURE_FINALI}(S_{21}) = \{w_2(x)\}$

  Osserviamo che $S_{li} \not\approx_V S_{21}$

Avendo provato per tutte le possibili permutazioni si può dire che lo schedule S_{li} **non è view-serializzabile**.

Esercizio 4.3. Consideriamo le seguenti transazioni:

$$t_1 : r_1(x);\ w_1(x);\ r_1(y);\ w_1(y)$$
$$t_2 : r_2(z);\ r_2(x);\ w_2(x);\ w_2(z)$$

Eseguite con lo schedule:

$$S_1 = r_1(x);\ w_1(x);\ r_2(z);\ r_1(y);\ w_1(y);\ r_2(x);\ w_2(x);\ w_2(z);$$

Questo schedule è view serializzabile?

Calcoliamo le relazioni:

- $\texttt{LEGGE_DA}(S_1) = \{(r_2(x), w_1(x))\}$
- $\texttt{SCRITTURE_FINALI}(S_1) = \{w_2(x), w_1(y), w_2(z)\}$

Bisogna generare tutti i possibili schedule seriali per determinare se esiste uno schedule seriale S' tale che $S_1 \approx_V S'$. Gli schedule seriali sono:

- $S_{12} = t_1 t_2$
- $S_{21} = t_2 t_1$

Al posto di creare tutte le permutazioni degli schedule seriali, cerchiamo di osservare le relazioni LEGGE_DA e SCRITTURE_FINALI per capire se S_1 è view-equivalente ad S_{12} o S_{21}.

Dalla presenza di $w_2(x)$ nelle scritture finali di S_1 si nota che $t_1 < t_2$ perchè $w_2(x)$ è l'ultima scrittura di x in S_1, e questa operazione viene eseguita solo da t_2 e di conseguenza va eseguita dopo t_1. Da ciò sappiamo che sicuramente $S_1 \not\approx_V S_{21}$ e ci rimane da controllare soltanto se $S_1 \approx_V S_{12}$:

$$S_{12} = r_1(x);\ w_1(x);\ r_1(y);\ w_1(y);\ r_2(z);\ r_2(x);\ w_2(x);\ w_2(z);$$

- $\texttt{LEGGE_DA}(S_{12}) = \{(r_2(x), w_1(x))\}$
- $\texttt{SCRITTURE_FINALI}(S_{12}) = \{w_2(x), w_1(y), w_2(z)\}$

Osserviamo che $S_1 \approx_V S_{12}$ e quindi S_1 è view-serializzabile.

Esercizio 4.4. Consideriamo le seguenti transazioni:

$$\begin{aligned} t_1 &: r_1(x);\ w_1(x);\ w_1(y) \\ t_2 &: r_2(y);\ r_2(x) \\ t_3 &: w_3(x);\ r_3(y);\ w_3(y) \end{aligned}$$

$$S_2 = r_1(x);\ w_1(x);\ w_3(x);\ r_2(y);\ r_3(y);\ w_3(y);\ w_1(y);\ r_2(x)$$

Questo schedule è view serializzabile?

Calcoliamo le relazioni:

- $\texttt{LEGGE_DA}(S_2) = \{(r_2(x), w_3(x))\}$
- $\texttt{SCRITTURE_FINALI}(S_2) = \{w_3(x), w_1(y)\}$

Siccome ci sono 6 possibili permutazioni vediamo se è possibile scartarne alcune a priori.

- L'insieme $\texttt{SCRITTURE_FINALI}(S_2)$ implica:
  - $t_1 < t_3$ perchè $w_3(x)$ è l'ultima scrittura di x in S_2 e questa operazione viene eseguita solo da t_3 e di conseguenza va eseguita dopo t_1
  - $t_3 < t_1$ perchè $w_1(y)$ è l'ultima scrittura di y in S_2 e questa operazione viene eseguita solo da t_1 e di conseguenza va eseguita dopo t_3

Si osserva che $t_1 < t_3$ e $t_3 < t_1$ è una contraddizione, cioè non esiste uno schedule seriale che le rispetti entrambe e quindi si può concludere che S_2 **non è view-serializzabile**.

- L'insieme $\texttt{LEGGE_DA}(S_2)$ implica:
  - $t_3 < t_2$ perchè $r_2(x)$ (in t_2) legge da $w_3(x)$ (in t_3) e quindi $(r_2(x), w_3(x)) \in \texttt{LEGGE_DA}$
  - $t_1 < t_3$ per garantire $(r_1(x), w_3(x)) \notin \texttt{LEGGE_DA}$
  - $t_2 < t_3$ per garantire $(r_2(x), w_3(x)) \in \texttt{LEGGE_DA}$
  - $t_2 < t_1$ per garantire $(r_2(x), w_1(x)) \notin \texttt{LEGGE_DA}$
  - $t_3 < t_1$ per garantire $(r_3(x), w_1(x)) \notin \texttt{LEGGE_DA}$

  Si osserva che $t_1 < t_3$ e $t_3 < t_1$ è una contraddizione, quindi si può concludere che S_2 **non è view-serializzabile**.

Esercizio 4.5. Consideriamo le seguenti transazioni:

$$\begin{aligned}
&t_1 : r_1(x);\ w_1(x);\ w_1(y);\ w_1(z)\\
&t_2 : r_2(x);\ w_2(x)\\
&t_3 : r_3(x);\ w_3(y);\ w_3(x)\\
&t_4 : r_4(z)\\
&t_5 : w_5(x);\ w_5(y);\ r_5(z)
\end{aligned}$$

eseguite con lo schedule:

$$\begin{aligned}
S_3 = &r_1(x);\ r_2(x);\ w_2(x);\ r_3(x);\ r_4(z);\ w_1(x);\ w_3(y);\ w_3(x);\ w_1(y);\\
&w_5(x);\ w_1(z);\ w_5(y);\ r_5(z)
\end{aligned}$$

Questo schedule è view serializzabile?

Calcoliamo le relazioni:

- $\texttt{LEGGE_DA}(S_3) = \{(r_3(x), w_2(x)), (r_5(z), w_1(z))\}$
- $\texttt{SCRITTURE_FINALI}(S_3) = \{w_5(x), w_5(y), w_1(z)\}$

Le permutazioni di tutte le transazioni sono 5! e quindi è necessario cercare di scartarne alcune:

- Considerando $\texttt{SCRITTURE_FINALI}(S_3)$ si ha che:
  - $w_5(x) \Rightarrow t_2 < t_5,\ t_1 < t_5,\ t_3 < t_5$
  - $w_5(y) \Rightarrow t_3 < t_5,\ t_1 < t_5$

  Da queste informazioni non si rilevano contraddizioni, quindi si procede con l'analisi di $\texttt{LEGGE_DA}(S_3)$.

- Considerando LEGGE_DA(S_3) si ha che:
  - $(r_3(x), w_2(x)) \Rightarrow t_2 < t_3$
  - $(r_5(z), w_1(z)) \Rightarrow t_1 < t_5$

  Da queste informazioni non si rilevano contraddizioni, quindi si procede osservando cosa **non compare** nell'insieme LEGGE_DA(S_3):
  - $(r_1(x), w_2(x)) \notin$ LEGGE_DA(S_3) $\Rightarrow t_1 < t_2$
  - $(r_1(x), w_3(x)) \notin$ LEGGE_DA(S_3) $\Rightarrow t_1 < t_3$
  - $(r_1(x), w_5(x)) \notin$ LEGGE_DA(S_3) $\Rightarrow t_1 < t_5$
  - $(r_2(x), w_1(x)) \notin$ LEGGE_DA(S_3) $\Rightarrow t_2 < t_1$ Questo è in contraddizione con $t_1 < t_2$ e si può concludere che S_3 **non è view-serializzabile**.

4.3.2 Conflict-equivalenza

Definizione 4.6 (Conflitto)**.** Dato uno schedule S si dice che una coppia di operazioni (a_i, a_j), dove a_i e a_j compaiono nello schedule S, rappresentano un conflitto se:

- appartengono a transazioni diverse ($i \neq j$)
- entrambe operano sulla stessa risorsa
- almeno una di esse è un'operazione di scrittura
- a_i compare in S prima di a_j

Definizione 4.7 (Conflict-equivalenza)**.** Due schedule S_1 e S_2 sono conflict-equivalenti ($S_1 \approx_C S_2$) se possiedono le stesse operazioni e gli stessi conflitti, cioè se le operazioni in conflitto sono nello stesso ordine nei due schedule.

Definizione 4.8 (Conflict-serializzabilità)**.** Uno schedule S è conflict-serializzabile (CSR) se esiste uno schedule seriale S' tale che $S \approx_C S'$.

L'algoritmo per il test di conflict-serializzabilità di uno schedule è di **complessità lineare** ed è il seguente:

Dato uno schedule S:

- Si costruisce il grafo dei conflitti $G(N, A)$ dove:
  - $N = \{t_1, \ldots, t_n\}$ è l'insieme delle transazioni di S
  - $(t_i, t_j) \in A$ se esiste almeno un conflitto (a_i, a_j) in S

- Se il grafo è **aciclico**, allora S è conflict-serializzabile.

  Dimostrazione:

  - Se S è CSR, cioè esiste uno schedule S_r seriale tale che $S \approx_c S_r$, allora il suo grafo è aciclico.

    Ipotesi: Le transazioni di S_r sono ordinate:

    $$t_1, t_2, \ldots, t_n$$

    Poichè $S \approx_C S_r$, allora S_r ha tutti i conflitti di S con operazioni esattamente nello stesso ordine. Quindi nel grafo dei conflitti di S_r ci possono essere solo archi (i, j) con $i < j$, e questo vuol dire che non possono esserci cicli perchè un ciclo richiederebbe un arco (j, i) con $j > i$ che è impossibile.

    Siccome $S \approx_c S_r$ allora hanno lo stesso grafo dei conflitti, e quindi anche il grafo dei conflitti di S è aciclico.

  - Se il grafo S è aciclico, allora S è CSR, cioè esiste uno schedule S_r seriale tale che $S \approx_C S_r$ e quindi S e S_r hanno gli stessi conflitti e quindi lo stesso grafo.

    Ipotesi: il grafo dei conflitti di S è aciclico, e quindi esiste un **ordinamento topologico**, cioè un'enumerazione dei nodi tale che il grafo contiene solo archi (i, j) con $i < j$. Utilizzando questo ordinamento topologico possiamo definire un ordine delle transazioni che produce lo stesso grafo e quindi siccome le transazioni sono ordinate si ha uno schedule seriale.

Richiede comunque l'ipotesi di commit-proiezione.

Esempio 4.4 (Perdita di aggiornamento)**.** Consideriamo le seguenti transazioni:

$$t_1 : r_1(x);\ w_1(x)$$
$$t_2 : r_2(x);\ w_2(x)$$

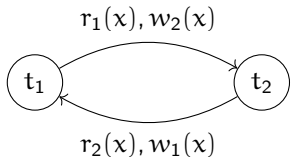
Eseguite con lo schedule:

$$S_{pa} :\ r_1(x);\ r_2(x);\ w_2(x);\ w_1(x);$$

L'insieme dei conflitti è:

$$\text{CONFLITTI}(S_{pa}) = \{(r_1(x), w_2(x)),\ (r_2(x), w_1(x)),\ (w_2(x), w_1(x))\}$$

Generiamo il grafo dei conflitti:



Siccome il grafo ha un ciclo S_{pa} **non è conflict-serializzabile**.

Esempio 4.5 (Lettura inconsistente)**.** Consideriamo le seguenti transazioni:

$$t_1 : r_1(x);\ r_1'(x)$$
$$t_2 : r_2(x);\ w_2(x)$$

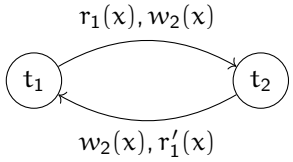
Eseguite con lo schedule:

$$S_{pa} :\ r_1(x);\ r_2(x);\ w_2(x);\ r_1'(x);$$

L'insieme dei conflitti è:

$$\text{CONFLITTI}(S_{pa}) = \{(r_1(x), w_2(x)),\ (w_2(x), r_1'(x))\}$$

Generiamo il grafo dei conflitti:



Teorema 4.1. La conflict-serializzabilità è una condizione sufficiente, ma non necessaria per la view-serializzabilità, cioè $\text{CSR} \subset \text{VSR}$:

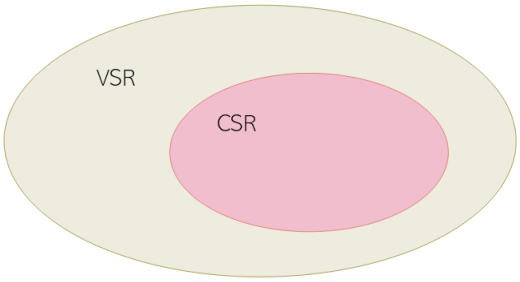


Figura 61: Relazione tra conflict-serializzabilità e view-serializzabilità

Dimostrazione: S è CSR $\Rightarrow$ S è VSR. (Se S non è CSR, allora non si può dire nulla sulla VSR)

- Dimostriamo che $\text{CSR} \subset \text{VSR}$: Basta trovare un controesempio:

$$S \in \text{VSR} \text{ ma } S \notin \text{CSR}$$

Consideriamo:

$$S = r_1(x);\ w_2(x);\ w_1(x);\ w_3(x);$$

Gli insiemi sono:

- $\texttt{LEGGE_DA}(S) = \varnothing$
- $\texttt{SCRITTURE_FINALI}(S) = \{w_3(x)\}$

Quindi questo schedule è VSR perchè esiste uno schedule S_r tale che $S_r \approx_V S$:

$$S_r = r_1(x);\ w_1(x);\ w_2(x);\ w_3(x);$$

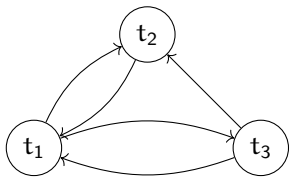
che ha gli stessi insiemi:

- $\texttt{LEGGE_DA}(S_r) = \varnothing$
- $\texttt{SCRITTURE_FINALI}(S_r) = \{w_3(x)\}$

Ora mostriamo che questo schedule non è CSR. Il suo insieme dei conflitti è:

$$\begin{aligned}\text{CONFLITTI}(S) =& \{(r_1(x), w_2(x)),\ (r_1(x), w_3(x)),\ (w_2(x), w_1(x)),\\ & (w_2(x), w_3(x)),\ (w_1(x), w_3(x))\}\end{aligned}$$

Costruiamo il grafo dei conflitti:



- Dimostriamo CSR $\Rightarrow$ VSR, quindi che $S_1 \approx_C S_2$ e dimostriamo $S_1 \approx_V S_2$.
  - S_1 e S_2 devono avere lo stesso insieme di SCRITTURE_FINALI, altrimenti ci sarebbero due scritture sulla stessa risorsa da parte di transazioni diverse in un ordine diverso, e questo produrrebbe un **conflitto diverso**
  - Devono avere lo stesso insieme di LEGGE_DA, altrimenti ci sarebbe un conflitto, composto da una lettura e una scrittura, diverso e quindi siccome il conflitto è diverso, allora S_1 e S_2 non sono conflict-equivalenti

Quindi se S:

- È CSR allora S è VSR
- Non è CSR allora S non è VSR

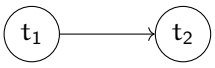
Esempio 4.6. Consideriamo il seguente schedule:

$$S_1 = r_1(x);\ w_1(x);\ r_2(z);\ r_1(y);\ w_1(y);\ r_2(x);\ w_2(x);\ w_2(z);$$

Verifichiamo se S_1 è VSR, per farlo vediamo se prima è CSR controllando l'insieme dei conflitti:

$$\text{CONFLITTI}(S_1) = \{(r_1(x), w_2(x)),\ (w_1(x), r_2(x)),\ (w_1(x), w_2(x))\}$$

Costruiamo il grafo dei conflitti:



Siccome il grafo è aciclico, allora S_1 è CSR e quindi S_1 è VSR.

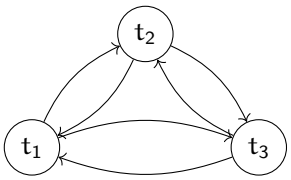
Esempio 4.7. Consideriamo lo schedule:

$$S_2 = r_1(x);\ w_1(x);\ w_3(x);\ r_2(y);\ r_3(y);\ w_3(y);\ w_1(y);\ r_2(x);$$

Verifichiamo se S_2 è VSR, per farlo vediamo se prima è CSR controllando l'insieme dei conflitti:

$$\begin{aligned}\text{CONFLITTI}(S_2) = \{&(r_1(x), w_3(x)),\ (w_1(x), w_3(x)),\ (w_1(x), r_2(x)),\\ &(w_3(x), r_2(x)),\ (r_2(y), w_3(y)),\ (r_2(y), w_1(y)),\\ &(r_3(y), w_1(y)),\ (w_3(y), w_1(y))\}\end{aligned}$$

Costruiamo il grafo dei conflitti:



Il grafo è ciclico, quindi S_2 **non è conflict-serializzabile** e di conseguenza S_2 **non è view-serializzabile**.

Questo schedule lo avevamo verificato anche nell'esercizio 4.4 e avevamo concluso che S_2 non è view-serializzabile, e questo conferma che S_2 non è conflict-serializzabile perchè $\text{CSR} \subset \text{VSR}$.

Esempio 4.8. Consideriamo lo schedule:

$$S_3 = r_1(x);\ r_2(x);\ w_2(x);\ r_3(x);\ r_4(z);\ w_1(x);\ w_3(y);\ w_3(x);\ w_1(y);\ w_5(x);\ w_1(z);\ w_5(y);\ r_5(z);$$

Verifichiamo se S_3 è VSR, per farlo vediamo se prima è CSR controllando l'insieme dei conflitti:

$$\begin{aligned}\text{CONFLITTI}(S_3) = \{&(r_1(x), w_2(x)),\ (r_1(x), w_3(x)),\ (r_1(x), w_5(x)),\\ &(r_2(x), w_1(x)),\ (r_2(x), w_3(x)),\ (r_2(x), w_5(x)),\\ &(w_2(x), r_3(x)),\ (w_2(x), w_1(x)),\ (w_2(x), w_3(x)),\\ &(w_2(x), w_5(x)),\ (r_3(x), w_1(x)),\ (r_3(x), w_5(x)),\\ &(r_4(z), w_1(z)),\ (w_1(x), w_3(x)),\ (w_1(x), w_5(x)),\\ &(w_3(y), w_1(y)),\ (w_3(y), w_5(y)),\ (w_3(x), w_5(x)),\\ &(w_1(y), w_5(y)),\ (w_1(z), r_5(z))\}\end{aligned}$$

Il grafo dei conflitti è ciclico, allora S_3 **non è conflict-serializzabile** e di conseguenza S_3 **non è view-serializzabile**.

Questo schedule lo avevamo verificato anche nell'esercizio 4.5 e avevamo concluso che S_3 non è view-serializzabile.

4.3.3 Locking a due fasi stretto

Il locking a due fasi (2PL, Two-Phase Locking) è il metodo applicato nei sistemi reali per la gestione dell'esecuzione concorrente delle transazioni perchè non richiede di conoscere in anticipo l'esito delle transazioni. **Il locking a due fasi è un sottoinsieme della conflict-serializzabilità.** Gli aspetti che caratterizzano il locking a due fasi sono i seguenti:

- **Meccanismo di base per la gestione dei lock**: Si basa sull'introduzione di alcune **primitive di lock** che consentono alle transazioni di bloccare le risorse sulle quali vogliono agire con operazioni di lettura e scrittura. Le primitive di lock sono le seguenti:
  - $r_lock_K(x)$: richiesta di un lock **condiviso** da parte della transazione t_K sulla risorsa x per eseguire una lettura
  - $w_lock_K(x)$: richiesta di un lock **esclusivo** da parte della transazione t_K sulla risorsa x per eseguire una scrittura
  - $unlock_K(x)$: richiesta da parte della transazione t_K di liberare la risorsa x da un precedente lock

  Le transazioni devono seguire le seguenti regole per applicare le primitive di lock:

  - Ogni **lettura** deve essere preceduta da un r_lock e seguita da un $unlock$. Sono ammessi più r_lock contemporanei sulla stessa risorsa (lock condiviso)

– Ogni **scrittura** deve essere preceduta da un w_lock e seguita da un $unlock$. **Non** sono ammessi più w_lock (oppure w_lock e r_lock) contemporanei sulla stessa risorsa (lock esclusivo)

Se una transazione segue le precedenti regole, allora si dice **ben formata** rispetto al locking.

- **Politica di concessione dei lock sulle risorse**: Il gestore dei lock (o gestore della concorrenza) mantiene per ogni risorsa x le seguenti informazioni:
  - **Stato**: $s(x) \in \{libero, r_lock, w_lock\}$
  - **Transazioni in r_lock**: $c(x) = (t_1, \ldots, t_n)$, dove $t_1, \ldots, t_n$ hanno un r_lock su x

  Il comportamento del gestore dei lock è descritto dalla seguente tablella:

Stato x / Richiesta	LIBERO		R_LOCK		W_LOCK	
$r_lock_K(x)$	Esito OK	Operazioni s(x) = r_lock c(x) ={k}	Esito OK	Operazioni c(x) = c(x)∪{k}	Esito Attesa	Operazioni -
$w_lock_K(x)$	Esito OK	Operazioni s(x) = w_lock	if |c(x)|=1 and k∈c(x) then Esito OK	Operazioni s(x) = w_lock	Esito Attesa	Operazioni -
			else Attesa	-		
$unlock_K(x)$	Esito Errore	Operazioni -	Esito OK	Operazioni c(x) = c(x) – {k} if c(x)=∅ then s(x)=Libero verifica coda	Esito OK	c(x)=∅ s(x)=Libero verifica coda

Tabella 1: Comportamento del gestore dei lock

- **Regola che garantisce la serializzabilità**: Una transazione **dopo aver rilasciato un lock non può acquisirne altri**. Questa regola è visualizzata dal seguente grafico:

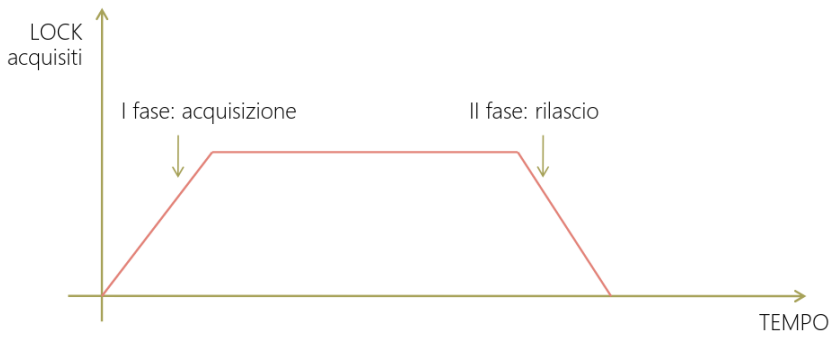


Figura 62: Locking a due fasi

Esempio 4.9 (Perdita di aggiornamento)**.** Consideriamo le seguenti transazioni:

$$t_1 : r_1(x);\ w_1(x)$$
$$t_2 : r_2(x);\ w_2(x)$$

Eseguite con lo schedule:

$$S_{pa} :\ r_1(x);\ r_2(x);\ w_2(x);\ w_1(x);$$

1. $r_1(x)$ richiede un $r_lock_1(x)$
2. $r_2(x)$ richiede un $r_lock_2(x)$
3. $w_2(x)$ richiede un $w_lock_2(x)$, ma lo stato $s(x) = r_lock$, con $c(x) = \{t_1, t_2\}$, quindi $w_2(x)$ per ottenere $w_lock_2(x)$ deve attendere che t_1 rilasci $r_lock(x)$ ma se t_1 rilascia $r_lock(x)$ e quindi inizierebbe la sua fase discendente impedendogli di acquisire altri lock e di conseguenza $w_lock_1(x)$ non è più possibile per la serializzabilità del 2PL

Di conseguenza questo schedule non è 2PL

Il locking a due fasi stretto (Strict 2PL) è una variante del locking a due fasi che introduce una condizione aggiuntiva per rimuovere la necessità dell'ipotesi di commit-proiezione. La condizione aggiuntiva è la seguente: Una transazione può rilasciare i lock solo quando ha eseguito correttamente un **commit** o un **rollback**.

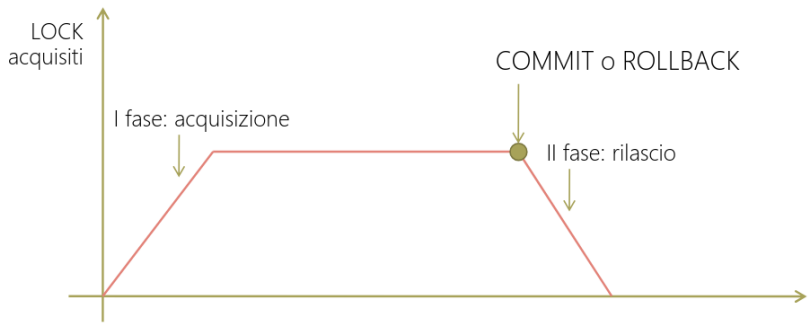


Figura 63: Locking a due fasi stretto

4.3.4 Blocco critico (deadlock)

Il blocco critico è una situazione di blocco che si verifica nell'esecuzione di transazioni concorrenti quando:

- Due transazioni hanno bloccato delle risorse:
  - t_1 ha bloccato r_1
  - t_2 ha bloccato r_2
- Inoltre t_1 è in attesa sulla risorsa r_2 e t_2 è in attesa sulla risorsa r_1

Se il numero medio di tuple per tabella è n e la granularità del lock (cioè il numero di risorse che possono essere bloccate) è la tupla, la probabilità che si verifichi un lock tra due transazioni è:

$$P(\text{deadlock di lunghezza } 2) = \frac{1}{n^2}$$

Esempio 4.10. Un esempio di deadlock utilizzando il locking a due fasi stretto è il seguente:

t1	M.C.	Database			M.C.	t2
		c(x)	s(x)	Val		
bot		∅	L	2		
$r_lock_1(x)$		{1}	RL	2		
$r_1(x)$	2	{1}	RL	2		
x = x +1	3	{1}	RL	2		
	3	{1}	RL	2		bot
	3	{1,2}	RL	2		$r_lock_2(x)$
	3	{1,2}	RL	2	2	$r_2(x)$
	3	{1,2}	RL	2	3	x=x+1
	3	{1,2}	RL	2	3	$w_lock_2(x) \rightarrow$ ATTESA
$w_lock_1(x) \rightarrow$ ATTESA	3	{1,2}	RL	2	3	

BLOCCO CRITICO

Figura 64: Esempio di deadlock con locking a due fasi stretto

Le tecniche per risolvere i deadlock sono le seguenti:

- **Timeout**: se una transazione è in attesa su una risorsa da troppo tempo viene abortita
- **Prevenzione**:
  - Una transazione blocca tutte le risorse a cui deve accedere in lettura o scrittura in una sola volta (o blocca tutto o non blocca nulla)
  - Ogni transazione t_i acquisisce un timestamp TS_i all'inizio della sua esecuzione. La transazione t_i può attendere una risorsa bloccata da t_j solo se vale una certa condizione sui timestamp, ad esempio $TS_i < TS_j$, altrimenti t_i viene abortita e fatta ripartire con lo stesso timestamp
- **Rilevamento**: Si esegue un'analisi periodica della tabella di lock per rilevare la presenza di un deadlock che corrisponde ad un ciclo nel grafo delle relazioni di attesa. Questa tecnica è la più utilizzata nei sistemi reali.

Inoltre la scelta di gestire i lock in lettura come lock condivisi introduce il problema della **starvation** che tuttavia nel contesto delle DMBS risulta poco probabile, ma comunque risolvibile con le tecniche precedenti (in particolare il rilevamento).

Definizione 4.9 (Starvation)**.** Se una risorsa x viene costantemente bloccata da una sequenza di transazioni che acquisiscono `r_lock` su x, un'eventuale transazione in attesa di scrivere su x viene bloccata per un lungo periodo fino alla fine della sequenza di letture.

Figura 65: Esempio di starvation

4.3.5 Gestione della concorrenza in SQL

Siccome risulta molto pesante per il sistema gestire l'esecuzione concorrente di transazioni con una conseguente diminuzione delle prestazioni, SQL prevede la possibilità di **rinunciare del tutto o in parte al controllo di concorrenza** per aumentare le prestazioni del sistema. È quindi possibile, a livello di una singola transazione, decidere di **tollerare alcune anomalie** di esecuzione concorrente. La seguente tabella mostra le possibili combinazioni di anomalie che si possono tollerare:

Livello di isolamento	Perdita di update	Lettura sporca	Lettura inconsistente	Update fantasma	Inserimento fantasma
serializable	✓	✓	✓	✓	✓
repeatable read	✓	✓	✓	✓	
read committed	✓	✓			
read uncommitted	✓				

Tabella 2: Anomalie tollerate in SQL

Nota:

- Tutti i livelli richiedono il 2PL stretto per le scritture
- **Serializable**: Richiede il 2PL stretto anche per le letture e applica il **lock sulle tabelle** per evitare l'inserimento fantasma
- **Repeatable read**: Applica il 2PL stretto anche per le letture, ma applica i **lock a livello di tupla** e non di tabella. Consente inserimenti e quindi non evita l'anomalia di inserimento fantasma. **Attenzione**: in PostgreSQL la repeatable read è implementata in modo più stringente che evita anche l'anomalia di inserimento fantasma
- **Read committed**: Richiede lock condivisi per le letture, ma non il 2PL stretto
- **Read uncommitted**: Non applica lock in lettura

5 Ottimizzazione

Siccome SQL è un linguaggio dichiarativo, l'ottimizzatore deve trovare un'espressione equivalente in un linguaggio procedurale (ad esempio l'algebra relazionale) per generare un piano di esecuzione.

5.1 Compilazione della query

Data una query:

1. Il primo passo è quello di verificare che sia scritta correttamente, quindi si esegue un'analisi lessicale e sintattica
2. Successivamente si svolge l'**ottimizzazione algebrica** che si basa sulle regole dell'algebra relazionale:
   - **Anticipo delle selezioni**
   - **Anticipo delle proiezioni**

   L'ottimizzazione algebrica è indipendente dal contenuto del database.
3. Dopo l'ottimizzazione algebrica si esegue l'**ottimizzazione dipendente dai metodi accesso** che si basa sulle statistiche e sulle strutture fisiche (profili delle tabelle)
4. Infine si genera un piano di esecuzione che verrà eseguito per ottenere il risultato della query

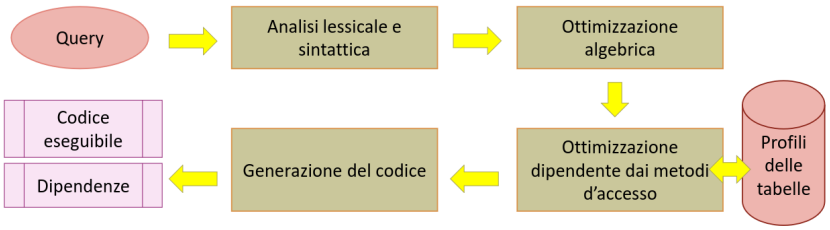


Figura 66: Ottimizzazione algebrica di una query

5.2 Ottimizzazione dipendente dai metodi di accesso

Per ottimizzare in base ai metodi di accesso bisogna tenere in considerazione le operazioni tipiche di accesso supportate dal DBMS, che sono:

- Scansione delle tuple di una tabella
- Ordinamento delle tuple
- Accesso diretto tramite indice
- Join con diverse implementazioni

5.2.1 Scansione sequenziale

Questa operazione viene fatta molto spesso e contestualmente ad altre operazioni:

- Scansione + proiezione senza eliminazione di duplicati
- Scansione + selezione in base ad un predicato semplice
- Scansione + inserimento o cancellazione o modifica

Il costo della scansione sequenziale è definito come il numero di pagine della relazione R:

$$NP(R)$$

dove NP() è il numero di pagine.

5.2.2 Ordinamento

L'ordinamento viene usato per:

- Ordinare il risultato di una query tramite la clausola `ORDER BY`
- Eliminare i duplicati (implica passare per un ordinamento) tramite l'operatore `SELECT DISTINCT`
- Raggruppare tuple tramite l'operatore `GROUP BY`

L'ordinamento è **eseguito su memoria secondaria** utilizzando l'algoritmo **Z-way Sort-Merge** che si basa su due fasi:

1. **Sort interno**: Si legge una pagina alla volta; tutte le tuple di ogni pagina vengono ordinate facendo uso di un algoritmo di sort interno (es. Quick-Sort); ogni pagina così ordinata, detta anche **"run"**, viene scritta su memoria secondaria in un file temporaneo
2. **Merge**: Applicando uno o più passi di merge, le run vengono unite, fino a produrre un'unica run ordinata.

Il costo di questo algoritmo, considerando solo gli accessi alla memoria secondaria, e considerando $Z = 2$ e $NB = 3$ è:

- Nella fase di sort interno, si leggono $NP(R)$ pagine e si scrivono $NP(R)$ pagine, quindi $2 \cdot NP(R)$
- Ad ogni passo di merge, si leggono $NP(R)$ pagine e si scrivono $NP(R)$ pagine, quindi $2 \cdot NP(R)$
- Il numero di passi di merge è:

$$\lceil \log_2(NP(R)) \rceil$$

Quindi il costo complessivo è:

$$2 \times NP(R) \times (\lceil \log_2(NP(R)) \rceil + 1)$$

Esempio 5.1. Un esempio di ordinamento con l'algoritmo Z-way Sort-Merge è il seguente:

Supponiamo di dover ordinare una tabella che composta di NP pagine e di avere a disposizione solo NB buffer in memoria centrale, con $NB < NP$.

Per semplicità consideriamo il caso base a due vie ($Z = 2$), e supponiamo di avere a disposizione solo 3 buffer in memoria centrale ($NB = 3$).

Si hanno 4 pagine composte da 3 tuple ciascuna, e si vuole ordinare in base alla prima colonna:

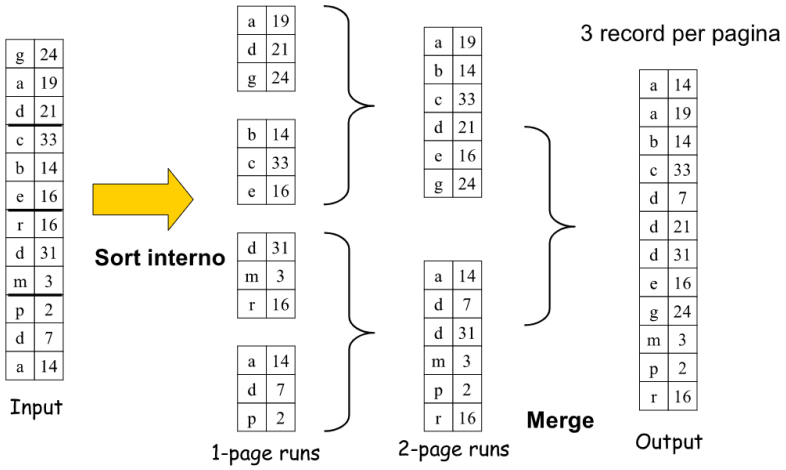


Figura 67: Esempio di ordinamento con Z-way Sort-Merge

Durante il sort interno si carica e ordina una pagina alla volta ottenendo delle run ordinate. Si ottengono 4 run ordinate che passano alla fase di merge a due vie, siccome $z = 2$, e si ottengono 2 run ordinate che vengono mergeate ulteriormente fino ad ottenere un'unica run ordinata.

Con $NB = 3$ si hanno solamente 3 buffer a disposizione, quindi si associano 2 buffer alle run da mergeare e 1 buffer per l'output. Si legge la prima pagina da ciascuna run e si genera la prima pagina dell'output, quando tutti i record di una pagina di run sono stati consumati, si legge un'altra pagina della run:

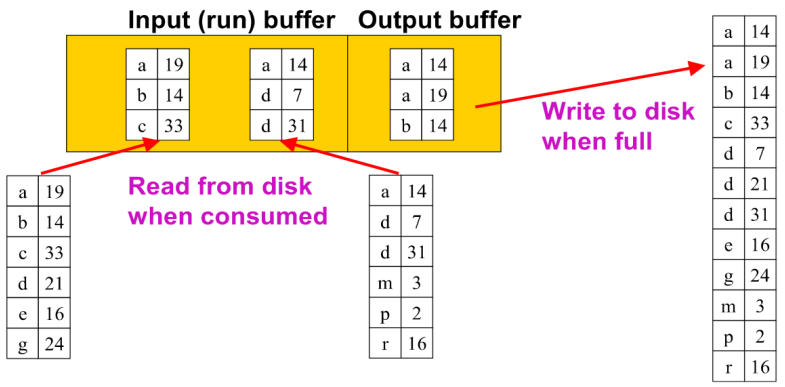


Figura 68: Esempio di utilizzo dei buffer

5.2.3 Accesso diretto tramite indice

Le operazioni che fanno uso dell'indice sono le selezioni con:

- Condizione atomica di uguaglianza ($A = v$). Può usare gli indici:
  - B+ tree
  - Hash
- Condizione di range ($A \geqslant v_1 \wedge A \leqslant v_2$). Può usare l'indice B+ tree, ma non l'indice hash perchè valori simili avrebbero valori molto diversi.
- Condizioni costruite da **congiunzione** di condizioni di uguaglianza ($A = v_1 \wedge B = v_2$). In questo caso si sceglie l'indice in base alla **condizione più selettiva**, l'altra condizione viene verificata filtrando i record restituiti dall'indice.
- Condizioni costruite da **disgiunzione** di condizioni di uguaglianza ($A = v_1 \vee B = v_2$). In questo caso si possono utilizzare più indici in parallelo e poi fare il merge o unione dei risultati.

5.2.4 Join

Il JOIN è l'operazione più costosa da eseguire, quindi l'ottimizzatore è particolarmente attento ad ottimizzare questa operazione, tanto da prevedere diversi algoritmi:

- **Nested loop join**: è l'algoritmo più diffuso e funziona in tutti i casi
- **Merge scan join**
- **Hash-based join**

Dal punto di vista logico il join è commutativo, ma dal punto di vista fisico c'è una differenza sostanziale tra $A \bowtie B$ e $B \bowtie A$ che cambia le prestazioni.

Nested loop join

Date 2 relazioni in input R e S tra cui è definito il predicato di JOIN PJ, distinguiamo:

- **Relazione esterna** R
- **Relazione interna** S

L'algoritmo funziona come segue:

```
∀ tupla t_R di R { // Relazione esterna
  ∀ tupla t_S di S { // Relazione interna
    se la coppia (t_R, t_S) soddisfa PJ
    allora aggiungi (t_R, t_S) al risultato
  }
}
```

Esempio 5.2. Consideriamo il seguente esempio di join tra le relazioni R e S:

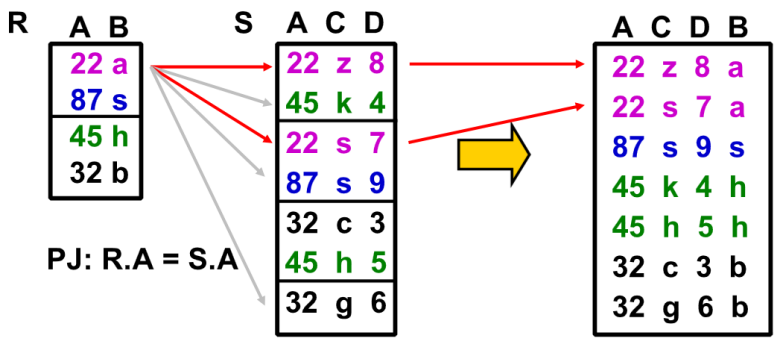


Figura 69: Esempio di nested loop join

R deve verificare per ogni riga di S che il predicato sia verificato.

Per calcolare il costo di nested loop join bisogna conoscere la dimensione del buffer, quindi consideriamo 1 pagina per caricare R e 1 pagina per caricare S:

- Bisogna leggere R completamente una volta
- Per ogni riga di R ($NR(R) = \#RIGHE\ R$) bisogna leggere completamente S

Il costo totale è quindi:

$$NP(R) + NR(R) \cdot NP(S)$$

A seconda del numero di righe di R cambia il numero di volte che bisogna leggere $NP(S)$, quindi la relazione esterna deve essere quella più piccola per minimizzare il costo.

Nested loop join con indice

Data una relazione esterna R e una interna S, si ha che la scansione completa di S può essere sostituita dalla scansione basata sull'indice costruito sugli attributi di join di S.

Esempio 5.3. Consideriamo il seguente esempio di nested loop join con indice tra le relazioni R e S:

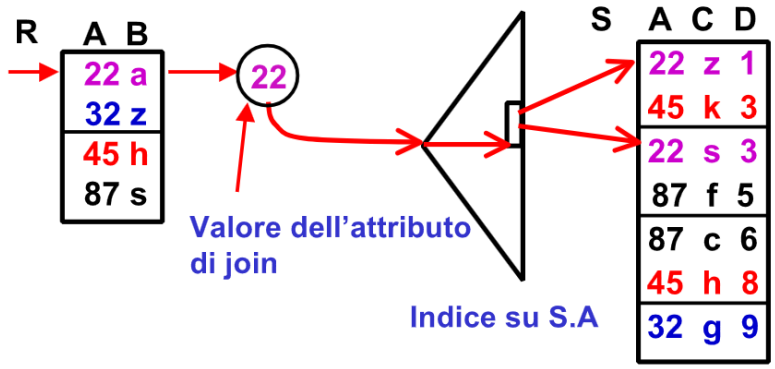


Figura 70: Esempio di nested loop join con indice

L'indice su $S.A$ permette di caricare solo i blocchi di S che contengono l'attributo $A = 22$.

Se come indice si usasse un B+tree, il costo sarebbe:

$$NP(R) + NR(R) \cdot \left(ProfonditaIndice + \frac{NR(S)}{VAL(A, S)} \right)$$

La tabella R deve comunque essere caricata tutta $(NP(R))$ e rimane invariato anche il numero di volte che si esegue il ciclo esterno $(NR(R))$. Ciò che cambia è il numero di blocchi/pagine di S che si devono caricare per ogni riga di R $(\left(ProfonditaIndice + \frac{NR(S)}{VAL(A,S)} \right))$. L'idea è invece di caricare tutte le pagine di S si caricano solo quelle che contengono $S.A = R.A$.

- $\frac{NR(S)}{VAL(A,S)}$ rappresenta la stima di quanti blocchi di S contengono A uguale al valore interessato; questa stima è anche chiamata **selettività** dell'attributo A.
- $VAL(A, S)$ è il numero di valori distinti di A che compaiono in S

Esempio 5.4. Supponiamo che $A = Nome_Regioni$, quindi il numero di possibili valori di A è 21. Consideriamo:

- $NR(S) = 21000$

- Il numero di righe per ogni regione è:

$$\frac{NR(S)}{VAL(A,S)} = \frac{21000}{21} = 1000$$

  quindi si ha l'assunzione che il dato sia uniforme, cioè che ci siano più o meno lo stesso numero di righe per ogni regione

- Nel caso pessimo si ha 1 blocco/pagina per riga.

Quindi il costo del nested loop join con indice è:

$$NP(R) + NR(R) \cdot \left(ProfonditaIndice + \frac{NR(S)}{VAL(A,S)} \right) =$$
$$NP(R) + NR(R) \cdot (ProfonditaIndice + 1000)$$

Block nested-loop join

Il block nested-loop join è una variante del nested loop join in cui ciascuna pagina di S (tabella interna) viene letta per ogni pagina di R (tabella esterna), invece che per ogni riga di R. Lo pseudocodice è il seguente:

```
∀ pagina P_R di R { // Relazione esterna
 ∀ pagina P_S di S { // Relazione interna
  ∀ tupla t_R in P_R {
   ∀ tupla t_S in P_S {
    se la coppia (t_R, t_S) soddisfa PJ
    allora aggiungi (t_R, t_S) al risultato
   }
  }
 }
}
```

Il costo di esecuzione di questo algoritmo dipende dallo spazio a disposizione nei buffer. Supponiamo di avere 1 buffer per R e 1 buffer per S, allora:

- Bisogna leggere R completamente una volta, quindi $NP(R)$
- Per ogni pagina di R ($NP(R)$) bisogna leggere completamente S quindi $NP(R) \cdot NP(S)$

Il costo totale è quindi:

$$NP(R) + NP(R) \cdot NP(S)$$

Esempio 5.5. Consideriamo il seguente esempio di block nested-loop join tra le relazioni R e S:

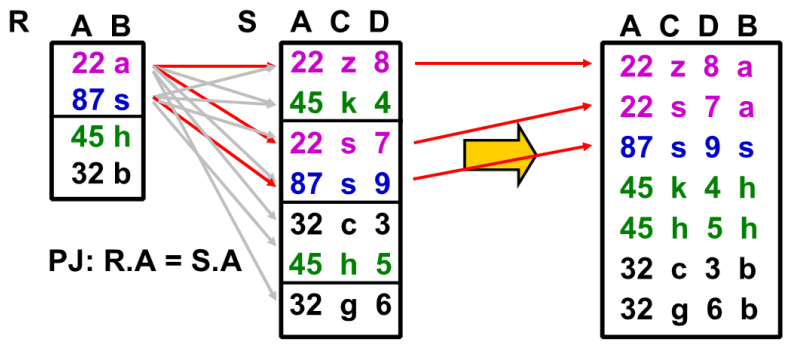


Figura 71: Esempio di block nested-loop join

Merge scan join

Il merge scan join è applicabile quando entrambi gli insiemi di tuple in input sono **ordinati** sugli attributi di join e il join è un **equi-join** (cioè il predicato di join è una condizione di uguaglianza). Ciò accade se per entrambe le relazioni R e S vale almeno una delle seguenti condizioni:

- La relazione è fisicamente ordinata sugli attributi di join (file sequenziale ordinato come struttura fisica).
- Esiste un indice sugli attributi di join della tabella che consente una scansione ordinata delle tuple.

La logica di questo algoritmo sfrutta il fatto che **entrambi gli input sono ordinati** per evitare di fare confronti inutili, e questo fa sì che il numero di letture totali sia:

$$NP(R) + NP(S)$$

se si accede sequenzialmente alle due relazioni ordinate fisicamente.

- Con indici B+tree su entrambe relazioni il costo può arrivare al massimo (con indici già nel buffer) a:
$$NR(R) + NR(S)$$
- Con una relazione ordinata e un solo indice il costo è al massimo:
$$NP(R) + NR(S) \quad \text{(o viceversa)}$$

Esempio 5.6. Consideriamo il seguente esempio di merge scan join tra le relazioni R e S:

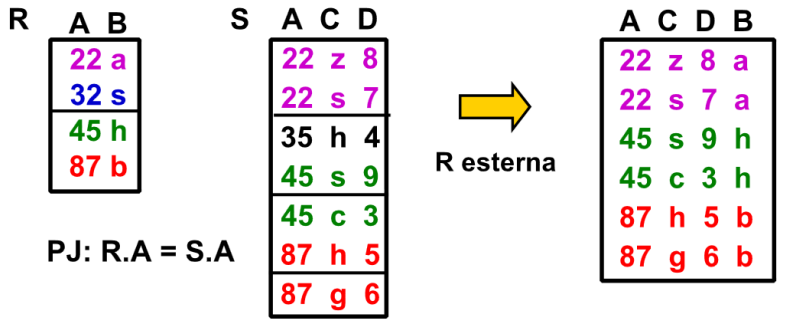


Figura 72: Esempio di merge scan join

Hash-based join

L'algoritmo di Hash Join, applicabile solo in caso di equi-join, non richiede né la presenza di indici né input ordinati, e risulta particolarmente vantaggioso in caso di relazioni molto grandi. L'idea su cui si basa l'Hash Join è semplice: Si suppone di avere a disposizione una funzione hash h, che viene applicata agli attributi di join delle due relazioni (ad es. $R.J$ e $S.J$).

- Se t_R e t_S sono 2 tuple di R e S, allora è possibile che sia $t_R.J = t_S.J$ solo se $h(t_R.J) = h(t_S.J)$
- Se, viceversa, $h(t_R.J) \neq h(t_S.J)$, allora sicuramente si ha $t_R.J \neq t_S.J$

Da questa idea si hanno diverse implementazioni, che hanno in comune il fatto che R e S vengono partizionate sulla base dei valori della funzione di hash h, e che la ricerca di **matching tuples** avviene solo tra partizioni relative allo stesso valore di h.

Il costo di questo algoritmo è:

- Costruzione dell'hash map:

$$NP(R) + NP(S)$$

- Accesso alle matching tuples che nel caso pessimo può costare:

$$NP(R) * NP(S)$$

5.3 Scelta del piano di esecuzione

Data un'espressione ottimizzata in algebra i passi da seguire sono:

1. Si generano **tutti i possibili** piani di esecuzione (alberi) alternativi (o un sottoinsieme) ottenuti considerando le seguenti dimensioni:
   - Operatori alternativi applicabili per l'accesso ai dati: ad esempio, scan sequenziale o via indice
   - Operatori alternativi applicabili nei nodi: ad esempio, nested-loop join o hash-based join

- L'ordine delle operazioni da compiere (associatività)

2. Si valuta con **formule approssimate** il costo di ogni alternativa in termini di accessi alla memoria secondaria richiesti (è una stima)
3. Si sceglie l'albero con il costo stimato minimo

Nella valutazione delle stime di costo si tiene conto del profilo delle relazioni, solitamente memorizzato nel Data Dictionary e contenente per ogni relazione T:

- Stima della cardinalità (#tuple): $\mathrm{CARD}(T)$
- Stima della dimensione di una tupla: $\mathrm{SIZE}(T)$
- Stima della dimensione di ciascun attributo A: $\mathrm{SIZE}(A, T)$
- Stima del numero di valori distinti per ogni attributo A: $\mathrm{VAL}(A, T)$
- Stima del valore massimo e minimo per ogni attributo A: $\mathrm{MAX}(A, T)$ e $\mathrm{MIN}(A, T)$

5.4 Riepilogo dei costi

Le operazioni tipiche supportate sono:

1. Scansione delle tuple di una relazione **Costo:** $\mathrm{NP}(R)$
   - Scan sequenziale per la ricerca (selezione)
   - Scan sequenziale per la proiezione
2. Ordinamento di un insieme di tuple
   - Z-way Sort-Merge **Costo:**

$$\underbrace{2\mathrm{NP}(R)}_{\text{Sort interno}} + \overbrace{2\mathrm{NP}(R)}^{\text{1 passo merge}} \cdot \underbrace{(\lceil \log_2(\mathrm{NP}(R)) \rceil + 1)}_{\text{Merge}}$$

3. Accesso diretto tramite indice
   - Hash **Costo costante** ≈ 1 bisogna:
     - Caricare indice
     - Accesso diretto tuple (NP che dipende da dove si trovano le tuple)
   - B+tree **Costo dipende dall'altezza dell'albero**

$$\mathrm{ProfonditaIndice} \leqslant 1 + \log_{\lceil \frac{n}{2} \rceil} \left(\frac{\mathrm{CARD}(R)}{\mathrm{VAL}(A, R)} \right)$$

4. Join
   - Nested loop join **Costo:**
     - Senza indice:

$$\mathrm{NP}(R) + \mathrm{NR}(R) \cdot \mathrm{NP}(S)$$

- Con indice:

$$NP(R) + NR(R) \cdot \left(ProfonditaIndice + \frac{NR(S)}{VAL(A, S)} \right)$$

- Merge-scan join **Costo:**
  - Senza indice (sequenziale ordinato)

$$NP(R) + NP(S)$$

  - Con indice

$$NR(R) + NR(S)$$

- Hash based join **Costo:**
  - Costruzione dell'hash map:

$$\underbrace{NP(R) + NP(S)}_{\text{Per costruire hash}} + \underbrace{NP(R) \cdot NP(S)}_{\text{Per match nel caso pessimo}}$$

  - Accesso alle matching tuples che nel caso pessimo può costare:

$$NP(R) * NP(S)$$

5.4.1 Esercizi

Esercizio 5.1. Consideriamo la seguente base di dati:

- Ingrediente(<u>codice</u>, nome, calorie)
- Composizione(<u>ricetta, ingrediente</u>, quantità)
- Ricetta(<u>codiceRicetta</u>, nome, regione, tempoPreparazione)

e i seguenti vincoli di integrità:

- Composizione.ricetta $\rightarrow$ Ricetta
- Composizione.ingrediente $\rightarrow$ Ingrediente

Data la seguente query:

```
1 select R.codiceRicetta, I.nome, I.calorie
2 from Ricetta R
3   join Composizione C on R.codiceRicetta = C.ricetta
4   join Ingrediente I on C.ingrediente = I.codice
5 where R.regione = 'Veneto'
```

1. Calcolare il costo dell'interrogazione in termini di numero di accessi alla memoria secondaria con le seguenti ipotesi:
   - La selezione (R.regione = 'Veneto') su Ricetta richiede una scansione sequenziale della tabella Ricetta. Inoltre la selezione rimane nel buffer

- L'ordine di esecuzione del join è:

$$(\mathrm{Ricetta} \bowtie \mathrm{Composizione}) \bowtie \mathrm{Ingrediente}$$

- Le operazioni di join vengono eseguite con la tecnica nested loop join con 1 pagina di buffer a disposizione per ogni tabella
- Il risultato del primo join (Ricetta $\bowtie$ Composizione) viene interamente mantenuto nel buffer
- Il numero di pagine e righe per ogni tabella è il seguente:

Tabella	NP	NR	VAL(Regione)	VAL(Ricetta)
Ricetta	12	260	20	
Ingrediente	40	1200		
Composizione	200	13000		260

2. Come cambia il costo se è disponibile un indice B+tree sull'attributo "codice" della tabella ingrediente con profondità 2?

Svolgimento:

1. L'ordine delle operazioni è:

   (a) Selezione su Ricetta:
   Siccome la scansione avviene in modo sequenziale bisogna leggere tutta la tabella, quindi il costo è:

   $$\mathrm{NP}(\mathrm{Ricetta})$$

   Siccome il risultato della selezione rimane nel buffer, non viene effettuata nessuna scrittura e quindi la scrittura ha costo 0.

   (b) Join su Composizione
   In join va il risultato della selezione su Ricetta, di cui abbiamo già calcolato il costo e che si trova ancora nel buffer, e quindi non tutta la tabella. Il join diventa quindi:

   $$\underbrace{\mathrm{Sel.Veneto}(\mathrm{Ricetta})}_{\mathrm{R}} \bowtie \underbrace{\mathrm{Composizione}}_{\mathrm{S}}$$

   Il costo è quindi:

   $$\mathrm{NP}(\mathrm{R}) + \mathrm{NR}(\mathrm{R}) \cdot \mathrm{NP}(\mathrm{S})$$

   - $\mathrm{NP}(\mathrm{R})$: è il costo della selezione, ma
     $\mathrm{R} = \mathrm{Sel.Veneto}(\mathrm{Ricetta})$ si trova già nel buffer, quindi questo costo si riduce a 0 perchè lo abbiamo già caricato in memoria.

- NR(R): è il numero di righe di Ricetta selezionate da Sel.Veneto che è data dalla **selettività** dell'attributo regione nella tabella Ricetta:

$$\mathrm{NR(R)} = \frac{\mathrm{NR}(\mathit{Ricetta})}{\mathrm{VAL}(\mathit{regione}, \mathrm{Ricetta})} = \frac{260}{20} = 13$$

- NP(S): è il numero di pagine di Composizione, che è 200

Quindi il costo totale è:

$$\mathrm{NP(R)} + \mathrm{NR(R)} \cdot \mathrm{NP(S)} = 0 + 13 \cdot 200 = 2600$$

(c) Join su Ingrediente
L'operazione del secondo join è data dal risultato del primo join, in join con Composizione, quindi:

$$\underbrace{\mathit{Risultato\ primo\ join}}_{\mathrm{R}} \bowtie \underbrace{\mathit{Ingrediente}}_{\mathrm{S}}$$

Il costo è quindi:

$$\mathrm{NP(R)} + \mathrm{NR(R)} \cdot \mathrm{NP(S)}$$

- NP(R): siccome il risultato del primo join viene interamente mantenuto nel buffer il costo per caricare R è 0
- NR(R): è il numero di righe del risultato del primo join, cioè il numero medio di tuple di Composizione per ogni Ricetta ($\frac{260}{20} = 13$) moltiplicato per il numero di Ricette del Veneto. Le ricette del veneto sono ($\frac{\mathrm{NR}(\mathit{Composizione})}{\mathrm{VAL}(\mathit{ricetta}, \mathit{Composizione})}$), Quindi:

$$\frac{\mathrm{NR}(\mathit{Composizione})}{\mathrm{VAL}(\mathit{ricetta}, \mathit{Composizione})} \cdot \frac{\mathrm{NR}(\mathit{Ricetta})}{\mathrm{VAL}(\mathit{regione}, \mathrm{Ricetta})}$$
$$= \frac{13000}{260} \cdot \frac{260}{20} = 650$$

- NP(S): è il numero di pagine di Ingrediente, che è 40

Quindi il costo totale è:

$$\mathrm{NP(R)} + \mathrm{NR(R)} \cdot \mathrm{NP(S)} = 0 + 650 \cdot 40 = 26000$$

Il costo totale della query è quindi la somma dei costi di ogni singola operazione:

$$12 + 2600 + 26000 = 28612$$

2. L'indice influisce solo sull'ultimo join perchè è definito sulla tabella Ingrediente.codice, quindi il costo totale è dato dalla somma dei costi di ogni singola operazione, con il costo del secondo join che cambia:

$$(\mathrm{op1}) + (\mathrm{op2}) + (\mathrm{op3}')$$

Dove:

$$op3' : \underbrace{\text{Risultato primo join}}_{R} \bowtie \underbrace{\text{Ingrediente}}_{S}$$

e il costo è:

$$NP(R) + NR(R) \cdot \left(ProfonditaIndice + \frac{NR(S)}{VAL(codice, S)} \right)$$

- $NP(R)$: siccome il risultato del primo join viene interamente mantenuto nel buffer il costo per caricare R è 0
- $NR(R)$: È lo stesso valore calcolato in precedenza, cioè 650
- $\left(ProfonditaIndice + \frac{NR(S)}{VAL(codice,S)} \right)$: La profondità dell'indice, data dal testo, è 2 e il numero di righe di Ingrediente è 1200, quindi rimane da calcolare il numero di valori distinti di codice nella tabella Ingrediente. Siccome codice è la chiave primaria di Ingrediente, il numero di valori distinti è uguale al numero di righe, quindi diventa:

$$2 + \frac{1200}{1200} = 3$$

Il costo totale di $(op3')$ è quindi:

$$0 + 650 \cdot 3 = 1950$$

Il costo totale dell'interrogazione con l'indice è quindi:

$$12 + 2600 + 1950 = 4562$$

Esercizio 5.2. Consideriamo la base di dati dell'esercizio precedente e la seguente query:

```
select C.Ingrediente, C.quantita, C.ricetta, R.regione
from Composizione C
  join Ricetta R on C.ricetta = R.codiceRicetta
where R.regione = 'Veneto'
  and C.quantita = 4
```

Le ipotesi da considerare sono le seguenti:

- La selezione su Composizione (C.quantita = 4) viene eseguita con una scansione sequenziale e il risultato viene salvato in 180 pagine di memoria secondaria
- La selezione su Ricetta (R.regione = 'Veneto') viene eseguita con una scansione sequenziale e il risultato viene mantenuto nel buffer
- L'esecuzione del join $Ricetta \bowtie Composizione$ è fatta con la tecnica di nested loop join

- Il numero di pagine e righe per ogni tabella è il seguente:

Tabella	NP	NR	VAL(Regione)	VAL(Ricet.)	VAL(qtà)
Ingrediente	30	270			
Composizione	780	19800		90	50
Ricetta	150	900	18		

1. Calcolare il costo della query
2. Come cambia il costo in presenza di un indice B+tree di profondità 2 sull'attributo ricetta di Composizione?

Svolgimento:

1. Le operazioni sono effettuate nel seguente ordine:
   (a) Selezione su Composizione
   Il costo è dato da:
   $$\underbrace{NP(R)}_{\text{Lettura}} + \underbrace{NP(R)}_{\text{Scrittura}}$$
   Bisogna leggere tutta la tabella Composizione, quindi il costo è
   $$NP(R) + NR(R) = 780 + 180 = 960$$
   (b) Selezione su Ricetta
   Il costo è dato da:
   $$\underbrace{NP(R)}_{\text{Lettura}} + \underbrace{NP(R)}_{\text{Scrittura}}$$
   Siccome il risultato viene mantenuto nel buffer, non viene effettuata nessuna scrittura e quindi la scrittura ha costo 0. Bisogna leggere tutta la tabella Ricetta, quindi il costo è
   $$NP(R) + NR(R) = 150 + 0 = 150$$
   (c) Join su Ricetta
   Il join viene effettuato tra il risultato della selezione su Ricetta con il risultato della selezione su Composizione, quindi:
   $$\underbrace{(\text{Ricetta con selezione})}_{R} \bowtie \underbrace{(\text{Composizione con selezione})}_{S}$$
   Il costo è dato da:
   $$NP(R) + NR(R) \cdot NP(S)$$
   - $NP(R)$: è il costo della selezione su Ricetta, ma siccome il risultato della selezione su Ricetta si trova già nel buffer, questo costo si riduce a 0

- $\mathrm{NR(R)}$: è il numero di righe del risultato della selezione su Ricetta, che è dato dalla selettività dell'attributo regione nella tabella Ricetta:

$$\mathrm{NR(R)} = \frac{\mathrm{NR}(Ricetta)}{VAL(regione, Ricetta)} = \frac{900}{18} = 50$$

- $\mathrm{NP(S)}$: è il numero di pagine del risultato della selezione su Composizione, che è 180

Il costo totale del join è quindi:

$$0 + 50 \cdot 180 = 9000$$

Il costo totale della query è quindi la somma dei costi di ogni singola operazione:

$$960 + 150 + 9000 = 10110$$

2. Osserviamo che l'attributo ricetta di Composizione è l'attributo di join per la tabella interna (S), quindi la presenza dell'indice ha un impatto solo sul costo del join, che diventa:

$$\mathrm{NP(R)} + \mathrm{NR(R)} \cdot \left(ProfonditaIndice + \frac{\mathrm{NR(S)}}{VAL(ricetta, \mathrm{S})} \right)$$

dove S è NR(Composizione con selezione)

- $\mathrm{NP(R)}$: è il costo della selezione su Ricetta, ma siccome il risultato della selezione su Ricetta si trova già nel buffer, questo costo si riduce a 0
- $\mathrm{NR(R)}$: è lo stesso valore calcolato in precedenza, cioè 50
- $\left(ProfonditaIndice + \frac{\mathrm{NR(S)}}{VAL(ricetta,\mathrm{S})} \right)$: Il costo viene stimato come:

$$\begin{aligned}
&\left(2 + \frac{\mathrm{NR}(Composizione)}{VAL(qta, Comp)} \cdot \frac{1}{VAL(ricetta, Comp)} \right) \\
&= \left(2 + \frac{19800}{50} \cdot \frac{1}{90} \right) \\
&= \left(2 + \frac{19800}{4500} \right) \\
&= (2 + 4.4) = 6.4 \approx 7
\end{aligned}$$

Il costo totale del join con l'indice è quindi:

$$0 + 50 \cdot 7 = 350$$

Il costo totale dell'interrogazione con l'indice è quindi:

$$960 + 150 + 350 = 1460$$

6 MongoDB

MongoDB è un database NoSQL document based e distribuito che implementa le seguenti features:

- **Replicazione**: consente di "disseminare" più copie identiche dello stesso dato in server diversi per garantire una maggiore resistenza ai guasti. Con la replicazione si aumenta l'**affidabilità**.
- **Sharding**: consiste nella suddivisione di un dataset di grandi dimensioni in parti più piccole. Con lo sharding si aumenta la **scalabilità orizzontale**. Ogni shard viene replicato.

Definizione utile 6.1. La scalabilità orizzontale consiste nell'aggiungere più nodi al sistema per aumentare la capacità di gestione dei dati, mentre la scalabilità verticale consiste nell'aumentare le risorse (CPU, RAM, storage) di un singolo nodo.

6.1 Replicazione

La replicazione in MongoDB è basata su un'architettura di **replica set**, che è una collezione di processi "mongod" (mongo daemon) che mantengono una copia di uno stesso dataset. Questo garantisce:

- Ridondanza e alta disponibilità
- Tolleranza ai guasti
- Fare aggiornamenti hardware o software di un singolo nodo senza dover interrompere il servizio

In un replica set viene identificato un nodo **primario** e vengono replicati dei nodi **secondari**. I nodi secondari si sincronizzano continuamente con il nodo primario tramite un **hartbeat**, che è un messaggio inviato periodicamente dal nodo primario ai nodi secondari per comunicare che è ancora attivo:

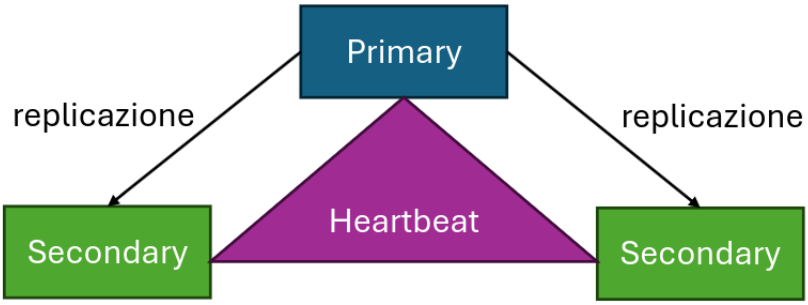


Figura 73: Replica set in MongoDB

- **Nodo primario**: gestisce tutte le operazioni di **scrittura** e mantiene un log (oplog) dei cambiamenti. Ogni replica set ha **un solo** nodo primario.

- **Nodi secondari**: replicano le operazioni scritte nell'oplog, quindi ripetono soltanto quello che ha fatto il nodo primario.

Se il nodo primario diventa non disponibile, allora un nodo secondario "qualificato" viene eletto nodo primario tramite un **algoritmo di elezione**, ad esempio il protocollo di consenso Raft.

6.1.1 Algoritmo di elezione Raft

L'algoritmo di elezione viene attivato in caso di:

- Un errore di comunicazione tra qualsiasi nodo secondario e il nodo primario, o in caso di comunicazione oltre un tempo di timeout (solitamente 10 secondi). La perdita del nodo primario viene identificata in modo automatico
- Inizializzazione di un replica set
- Modifica di un replica set, cioè quando viene aggiunto o rimosso un nodo dal replica set
- Votazione per determinare il nuovo nodo primario. Questa votazione non richiede più di 12 secondi, compreso il tempo in cui ci si deve accorgere che il nodo primario non è più disponibile

Il nodo secondario con **timestamp di scrittura più recente** è quello con maggiori probabilità di vincere l'elezione.

Per evitare elezioni continue, una volta conclusa un'elezione, i nodi entrano in un periodo di "congelamento" durante il quale non possono iniziare nuove elezioni. Finchè un'elezione non è conclusa, il replica set non può eseguire operazioni di scrittura.

6.1.2 Oplog

È un log che contiene tutte le operazioni eseguite sui database. Le operazioni di scrittura vengono prima eseguite sul nodo primario e poi trasmesse sugli oplog dei nodi secondari. Per supportare la replicazione è necessario che tutti i membri del replica set si scambino informazioni attraverso la condivisione dei propri oplog. Inoltre le operazioni nell'oplog sono idempotenti, cioè applicazioni multiple della stessa operazione producono lo stesso risultato di una singola applicazione.

6.1.3 Replicazione delle scritture

Per garantire l'affidabilità delle repliche bisogna avere un meccanismo per stabilire che i dati siano segnalati come scritti solo quando un certo numero di nodi ha effettuato tale scrittura.

La **write concern** è un meccanismo che consente di specificare il numero di nodi che devono confermare la scrittura prima che questa venga considerata completata.

- w: è il numero di nodi del replica set che devono aver confermato la scrittura prima che l'operazione sia considerata conclusa. I valori possibili sono:

  - 0: non si aspetta che il nodo primario notifichi la scrittura nel proprio oplog
  - 1: si aspetta soltanto il nodo primario
  - $majority$: si aspetta che la maggioranza dei nodi del replica set confermi la scrittura
  - un numero specifico n: si aspetta che n nodi confermino la scrittura.

  Più un valore è basso più si riduce la latenza e più è alto più si garantisce la persistenza.

- j: è un flag che richiede la conferma che l'operazione di scrittura sia stata scritta sul disco
- $wtimeout$: è un limite di tempo oltre il quale se l'operazione di scrittura non è terminata si restituisce un errore

6.1.4 Replicazione delle letture

Di default le operazioni di lettura vengono redirette sul nodo primario, ma è possibile distribuire le letture su più nodi per migliorare le prestazioni e bilanciare il carico secondo i seguenti parametri:

- $primary$: si legge solo dal nodo primario
- $primaryPreferred$: si legge dal nodo primario se è disponibile, altrimenti si legge da un nodo secondario
- $secondary$: si legge solo dai nodi secondari
- $secondaryPreferred$: si legge da un nodo secondario se è disponibile, altrimenti si legge dal nodo primario
- $nearest$: si legge dal nodo più vicino (con latenza più bassa)

Se si legge da un nodo secondario bisogna **garantire la consistenza dei dati letti da una copia** (Read Concern). Per farlo si può specificare un livello di read concern:

- **Local**: ritorna il dato disponibile più recente nel nodo al momento in cui viene eseguita la query indipendentemente dallo stato di replicazione
- **Majority**: ritorna il dato che è stato confermato dalla maggior parte dei nodi del replica set. Questo garantisce un'alta consistenza
- **Linearizable**: ritorna il dato confermato da tutti. Questo garantisce la massima consistenza, ma è anche il più lento
- **Snapshot**: ritorna i dati confermati dalla maggioranza dei nodi in un certo momento (anche antecedente al momento corrente)

6.2 Architettura dei replica set

Un replica set è composto da 3 membri di default, di cui uno è un nodo primario e gli altri due sono nodi secondari. Le best practice per i replica set sono:

- Devono contenere un numero **dispari** di membri votanti perchè questo evita situazioni di stallo dovute a priorità di voto.
- Se i membri non sono dispari è presente un **arbitro**. L'arbitro non mantiene una copia dei dati e non può diventare un nodo primario
- Includere la presenza di membri nascosti o ritardati che si occupano di task particolari, ad esempio backup o reporting.
  - I nodi nascosti non possono diventare nodi primari e non sono direttamente accessibili dalle applicazioni client
  - I nodi ritardati sono predisposti per replicare le operazioni del nodo primario con un certo ritardo preimpostato e vengono utilizzati come una fonte di backup continuo
- Assicurarsi che almeno un membro di ciascun replica set si trovi in un data center alternativo

6.3 Sharding

Lo sharding consiste nel dividere il database in parti più piccole dette **shard**. Lo shard rappresenta il replica set che memorizza una porzione dei dati:

- **Mongos**: funge da distributore di query e si occupa di reindirizzare in modo efficiente le query tra i vari shard. Una volta ottenuti i risultati parziali li consolida producendo una risposta definitiva
- **Config server**: mantiene i metadati relativi allo sharding

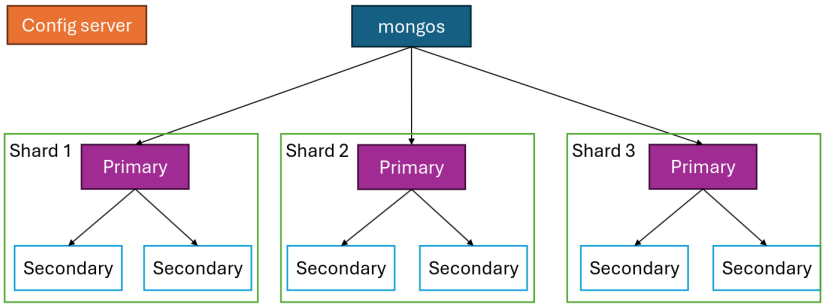


Figura 74: Sharding in MongoDB

I vantaggi dello sharding sono:

- Aumentare velocità di lettura e scrittura
- Aumentare la capacità di archiviazione
- Sfruttare la località del dato (zone sharding)

6.3.1 Architettura dello sharding

Lo sharding viene implementato a livello di collezione e permette la suddivisione dei documenti in **chunk**. La distribuzione del dato avviene tramite la **shard key** che è ottenuta a partire da una o più proprietà dei documenti. Esistono due principali strategie per implementare la suddivisione dei documenti in chunk:

1. **Ranged sharding**: i dati vengono suddivisi in base ad intervalli continui dei valori della chiave di shard. Le caratteristiche di questo metodo sono:
   - Questo metodo è ottimale per range query sui valori chiave
   - Il dominio dell'attributo chiave shard determina il numero di possibili shard
   - Se si hanno valori più probabili si possono ottenere chunk più numerosi e chunk più piccoli
   - Se si hanno valori che crescono o decrescono in modo monotonico, producono più probabilmente delle operazioni di inserimento che coinvolgono un unico chunk
2. **Hashed sharding**: opera calcolando il valore di hash della chiave di shard e assegnando ciascun chunk in base ad un range determinato da questo valore di hash. Le caratteristiche di questo metodo sono:
   - Valori di chiave simile avranno un valore di hash molto diverso, quindi un chunk diverso
   - Non sono efficienti con range query

6.3.2 Balancer

Il balancer è un processo in background che si assicura che la quantità di dati assegnati a ciascun shard sia la più bilanciata possibile. Quando la quantità di dati assegnata ad un particolare shard raggiunge un certo threshold, il balancer redistribuisce con principio di località i chunk tra gli shard. La redistribuzione funziona solo per chunk piccoli, quindi:

- **Jumbo chunk**: se un chunk è troppo grande, ma non può essere diviso in automatico, esso richiede un intervento manuale
- **Chunk indivisibile**: è un chunk caratterizzato da un singolo valore di chiave shard, si effettua quindi resharding, cioè si modifica la chiave di shard.

6.4 Interrogazioni in presenza di shared data

In presenza di shard, le query vengono inviate e gestite dal processo **mongos**, che agisce da distributore delle query e decide a quale shard devono essere inviate. Agisce effettivamente da proxy delle richieste. Le funzioni utilizzate da mongos per gestire le query sono:

- `find()`
- `sort()`
- `limit()`

6.4.1 Proprietà ACID

- **A**tomicità:
  - Tutte le operazioni che coinvolgono **un solo** documento sono sempre atomiche, anche in caso di attraversamento di proprietà che sono suddivise in più documenti o array
  - Si stima che l'80%-90% delle applicazioni reali richiede query monodocumento. In MongoDB c'è un limite di 16mb per documento, quindi se si supera questo limite è necessario suddividere i dati in più documenti.
- **C**onsistenza: ci sono due livelli di consistenza:
  - **Strong consistency**: garantisce che ogni lettura fatta leggerà sempre l'ultimo valore confermato. Questo però introduce problemi di prestazioni, ma garantisce più affidabilità
  - **Eventual consistency**: garantisce che quando i dati sono stati aggiornati e propagati, tutte le letture future ritorneranno eventualmente l'ultimo stato confermato. Questo fornisce alte prestazioni.

  MongoDB garantisce un livello di consistenza a metà tra i due livelli.
- **I**solamento: MongoDB gestisce i livelli di isolamento tramite il meccanismo di **write concern** e **read concern**.
  - **Read concern**:
    * Local: ritorna il dato più recente disponibile nel nodo, ma non è detto che tutte le transazioni vedano lo stesso dato
    * Majority: garantisce che il dato letto sia quello confermato dalla maggioranza dei nodi e questo garantisce che tutte le transazioni vedano lo stesso risultato
    * Snapshot: ritorna il dato presente nello snapshot della maggioranza dei nodi, questo garantisce che tutte le transazioni vedano lo stesso risultato
  - **Write concern**
    * Majority: richiede che la maggioranza dei nodi in un replica set confermi il dato
- **D**urability (persistenza): MongoDB usa il meccanismo Write-Ahead Log sul disco ogni 100 millisecondi per garantire la persistenza dei dati. Le operazioni vengono quindi scritte prima nel file di log e poi vengono eseguite, in questo modo è possibile ripristinare lo stato del database in caso di errori.

7 Esercizi

7.1 Query

Si consideri la seguente base di dati:

- AUTO(<u>targa</u>, num.telaio, modello, colore, km, cilindrata, posti, proprietario)

- PRENOTAZIONE(codice, parcheggio, posto, auto, data_prenotazione, data_ora_inizio, data_ora_fine, tipo, pagato)
- PARCHEGGIO(nome, comune, indirizzo, coperto)

con i seguenti vincoli di integrità:

- PRENOTAZIONE.auto → AUTO
- PRENOTAZIONE.parcheggio → PARCHEGGIO

7.1.1 Esercizio 1

Trovare tutti i parcheggi in cui vengono effettuate più prenotazione di tipo online che prenotazioni fisiche.

```sql
-- Numero di prenotazioni online
select P.nome
from parcheggio P
  join prenotazione R on P.nome = R.parcheggio
where tipo = 'online'
group by P.nome
having count(*) > -- num prenotazioni online di un parcheggio
  (
    -- Numero di prenotazioni fisiche
    select count(*)
    from prenotazione R1
    where R1.tipo = 'fisica'
      and R1.nome = P.nome
  )
```

La query esterna viene eseguita per ogni riga della query esterna. Usando il data binding non si ha bisogno del group by perchè si stanno selezionando le righe di uno specifico parcheggio e quindi si vogliono contare tutte.

7.1.2 Esercizio 2

Trovare tutti i parcheggi in cui non sono mai state fatte prenotazioni fisiche nel 2024

- Usando `except`

```sql
-- Tutti i parcheggi
select P.nome
from parcheggio P

except

-- Tranne quelli con una prenotazione nel 2024
select R.parcheggio
from prenotazione R
where R.tipo = 'fisica'
  and R.data_prenotazione >= '01/01/2024'
  and R.data_prenotazione <= '31/12/2024';
  -- equivale a
  -- extract(year from R.data_prenotazione) = '2024'
```

- Usando `not in`

```sql
select P.nome
from parcheggio P
where P.nome not in (
  select R.parcheggio
  from prenotazione R
  where R.tipo = 'fisica'
    and R.data_prenotazione >= '01/01/2024'
    and R.data_prenotazione <= '31/12/2024'
);
```

7.1.3 Esercizio 3

Trovare quanti parcheggi del comune di Verona hanno avuto un numero medio di prenotazioni complessive nel 2024 superiore al numero medio di prenotazioni dei parcheggi del comune di Padova.

```sql
create temp view nprenotazioni as (
  select P.nome, P.comune, count(*) as num
  from parcheggio P
    join prenotazioni R on P.nome = R.parcheggio
  where R.data_prenotazione >= '01/01/2024'
    and R.data_prenotazione <= '31/12/2024'
  group by P.nome, P.comune
);

select count(*)
from nprenotazioni N
where N.comune = 'Verona'
  and N.num > (
    select avg(N1.num)
    from nprenotazioni N1
    where N1.comune = 'Padova'
  );
```