BASI DI DATI

Corso A

Autore
Giuseppe Acocella
2025/26
https://github.com/Peenguino

Ultima Compilazione - February 10, 2026

Contents

1 Introduzione

I database sono insiemi di dati omogenei gestiti in collezioni. Alla base di questo definiamo tabelle, i cui campi possono fare riferimento ad altre tabelle del database. Il topic di studio del corso è quello dei **database**, ma in questa introduzione definiamo un confronto con i **Data Warehouse**, per evidenziarne le **differenze**, anche non essendo parte del programma del corso.

1.1 OLTP vs OLAP

Mettiamo a confronto i due tipi di **ambiti applicativi**:

1. **OLTP - Database (transazionale)**:
   (a) Utilizzo comune.
   (b) Molti users.
   (c) Dati analitici e relazionali.
   (d) Relazioni statiche.
   (e) Una query altera solitamente pochi record della tabella.
   (f) Mirato all'utilizzo da parte delle applicazioni.
   (g) Aggiornamenti frequenti.
   (h) Visione dei dati correnti.
   (i) Pensato per le transazioni.
2. **OLAP - Data Warehouse (analitico)**:
   (a) Pochi utenti esperti.
   (b) Dati multidimensionali.
   (c) Relazioni dinamiche.
   (d) Una query altera molti record della tabella.
   (e) Mirato ai soggetti.
   (f) Aggiornamenti rari ma massivi.
   (g) Visione dei dati storica.
   (h) Pensato per l'analisi di trend.

Per questa motivazione, se dei dati presenti in un database, dovessero servire per un analisi di trend andrebbe effettuata un operazione abbastanza complessa di estrazione e preparazione per l'immagazzinamento nel Data Warehouse. Entrambi (DB e DW) seguono una politica **schema first**, ossia viene prima definito uno schema (insieme di campi) su cui verrà basata la successiva popolazione della collezione di informazioni.

Big Data Un esempio di collezione di dati che **non segue** una politica **schema first**, basandosi infatti sulle proprietà di volume, varietà e velocità non possono mantenere la rigidità impostata da uno schema. Solitamente sono quindi associati a sistemi NoSQL o approcci Data Lake.

1.2 Data Base Management Systems - DBMS

Un DBMS (sistema per basi di dati) è un sistema centralizzato o distribuito che offre opportuni linguaggi per:

1. Definire lo **schema** di una DB.
2. Scegliere le **strutture dati** a **supporto** della DB.
3. **Memorizzare** dati seguendo i vincoli definiti dai schemi del DB.
4. Recuperare e modificare dati del DB tramite interrogazioni (**query**).

Solitamente si pone tra i programmi applicativi e l'effettivo database per permettere l'interazione vincolata tra i due.

Dati gestiti dai DBMS Solitamente in un DB sono contenuti:

1. **Metadati**: Descrivono permessi, applicazioni, parametri quantitativi sui dati effettivi. Seguono uno schema definito dal DBMS stesso.
2. **Dati**: Rappresentazioni di fatti conformi alle definizioni degli schemi del DB.
   (a) Sono organizzati in **insiemi** strutturati ed **omogenei**, tra i quali sono definite delle **relazioni**.
   (b) Sono accessibili tramite **transazioni**, operazioni atomiche che non hanno **mai effetti parziali**.
   (c) Sono protetti da accessi non autorizzati e preservati da possibili malfunzionamenti.
   (d) Sono utilizzabili in maniera concorrente da più utenti.

DBMS a Modello Relazionale Il modello relazionale è il più comune tra i DBMS commerciali e si basa sull'astrazione della **relazione**, ossia la **tabella** vista come un insieme di record con campi ben definiti. Questo ci permette di poter creare tabelle ed interrogarle con un linguaggio ad alto livello.

Funzionalità dei DBMS Elenchiamo quindi le proprietà garantite da un DBMS:

1. Linguaggio per la definizione di un DB.
2. Linguaggio per l'uso dei dati nel DB.
3. Meccanismi di controllo del DB.
4. Strumenti per la gestione admin del DB.
5. Strumenti per lo sviluppo delle app che richiedono dati dal DB.

1.2.1 Livelli di Vista Dati dei DBMS

Per garantire le proprietà di **indipendenza fisica** e **logica** dei dati è stato proposto l'approccio di tre livelli di descrizione dei dati.

1. **Indipendenza Fisica**: Le applicazioni che utilizzano il DB **non** devono essere modificati in seguito a modifiche dell'organizzazione fisica dei dati nel DB.
2. **Indipendenza Logica**: Le applicazioni che utilizzano il DB **non** devono essere modificati in seguito a modifiche dello schema logico del DB.

Gli effettivi livelli di vista sono invece:

1. **Livello Fisico**: Gestione effettiva dell'immagazzinamento dei dati nel DB, ad esempio in questo livello si scelgono le strutture dati e gli algoritmi utilizzati dal DBMS per navigare tra i dati.
2. **Livello Logico**: Descrizione della struttura degli insiemi di dati e delle relazioni tra di loro, astraendo completamente dalla loro gestione fisica.
3. **Livello Vista Logica**: Sottoinsieme del livello logico esposto alle applicazioni esterne.

1.2.2 Meccanismi di Controllo Dati e Transazioni dei DBMS

I DBMS cercano di garantire queste proprietà sui dati immagazzinati in un DB:

1. **Integrità**: Mantenimento delle proprietà definite dallo schema.
2. **Sicurezza**: Protezione dei dati da usi non autorizzati.
3. **Affidabilità**: Protezione in caso di malfunzionamenti hardware/software.

Transazioni - Operazioni Atomiche Una transazione è una sequenza di azioni di lettura e scrittura in memoria permanente e di elaborazioni di dati in memoria temporanea secondo queste proprietà:

1. **Atomicità**: Le transazioni terminate prematuramente sono trattate come se non fossero mai iniziate. I loro effetti sul DB sono nulli.
2. **Persistenza**: Le transazioni terminate con successo sono permanenti, ossia non alterabili neanche da malfunzionamenti.
3. **Serializzabilità**: L'esecuzione concorrente di più transazioni è vista come un esecuzione seriale di transazioni.

Riepilogo Pro e Contro Utilizzo DBMS Elenchiamo rapidamente i pro e i contro di questo approccio:

1. **Pro**: Indipendenza dei dati, recupero efficiente dei dati, integrità e sicurezza, accessi interattivi e concorrenti, amministrazione e riduzione dei tempi di sviluppo delle applicazioni.
2. **Contro**: Necessaria la definizione di uno schema, gestiscono solo dati strutturati ed omogenei, ottimizzati per app OLTP e non per OLAP.

2 Progettazione di una Base Dati

La nascita dei database è causata da alcune problematiche presenti in sistemi di gestione informazioni più datati. Un classico esempio di problematica è quella della **ridondanza logica**, ossia un informazione ripetuta più volte tra vari record. In queste casistiche si preferisce astrarre e fare riferimento solo una volta ad un dato.

2.1 Attori e Fasi della Progettazione di DB

Elenchiamo attori e fasi della progettazione di una base dati.

Attori della Progettazione Elenchiamoli:

1. **Committente**: Azienda che commissiona la creazione di una base dati, per una propria necessità.
2. **Consulente**: Progettisti del DB.
3. **Utente**: Chi usufruirà del DB, solitamente un dipendente del committente.
4. **DB Administrator**: Amministratore del DB.

Fasi della Progettazione Si definiscono fasi specifiche della progettazione di un DB:

1. **Specifica Requisiti Committente**: Il committente deve definire le proprie necessità ed il consulente deve raccogliere le informazioni.
2. **Progettazione Concettuale**: Realizzazione di uno schema concettuale orientato agli oggetti che deve essere osservabile ed approvato dal committente. In questa fase si astrae completamente da dettagli tecnici d'implementazione/ottimizzazione proprio perchè deve risultare semplice al committente e non deve causare ridondanza logica.
3. **Progettazione Logica**: Concretizzazione della progettazione concettuale tramite linguaggi relazionali.
4. **Progettazione Fisica**: Allocazione fisica delle tabelle generate dal linguaggio relazionale della progettazione logica.

Progettazione come Modellazione Spesso il committente stesso non riesce a rendere esplicite tutte le sue necessità, di conseguenza è compito del consulente capire a fondo i comportamenti ed i dettagli necessari alla modellazione logica.

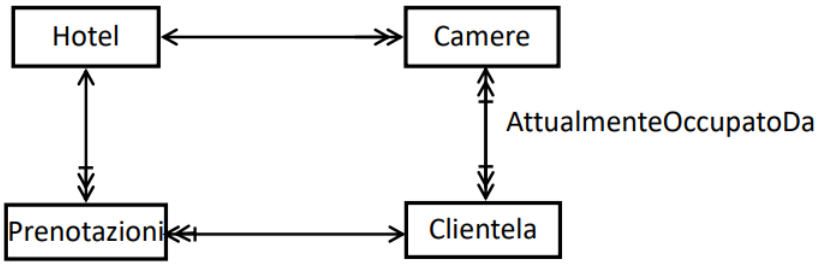
2.2 Progettazione Concettuale

Questo linguaggio si basa su 3 operatori diversi:

1. **Classi (aka Collezioni)**: Ad esempio la classe Persone di entità persona. Formalmente una classe modella un insieme di entità omogenee. Queste possono essere entità fisiche, avvenimenti o **modelli (progetti)** di entità.

2. **Associazioni**: Insieme di fatti binari, ad esempio associazione di un proprietario ad un auto.

3. **Sottoclassi**: Sottoinsieme di una classe, come Studenti può esserlo di Persone.



2.2.1 Associazioni e Cardinalità

Formalmente le associazioni sono insiemi di coppie, quindi delle relazioni. Il tipo di freccetta che indica l'associazione è detta cardinalità e corrisponde informalmente alla domanda:

"Per ogni elemento della classe A quanti della classe B?"

Chiaramente va fatto sulla stessa direzione per entrambi i versi. Questo permette di risolvere le ambiguità causate dalla terminologia comune che è detta:

"Uno a Molti" oppure "Molti a Molti"

che in qualche modo genera ambiguità perchè è come se si tenesse in conto solo di un verso. E' fondamentale ricordarsi quindi che la caratterizzazione della cardinalità di un associazione va fatta in entrambi i versi.

Graficamente quindi avremo queste possibilità alle estremità delle associazioni:

1. **Freccia Singola**: Per ogni elemento della classe di partenza è presente un elemento nella classe d'arrivo.

2. **Freccia Doppia**: Per ogni elemento della classe di partenza sono presenti più elementi nella classe d'arrivo.

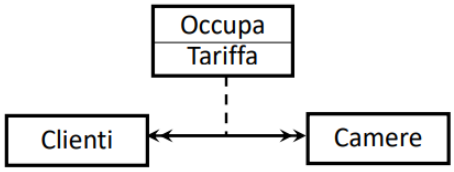
3. **Trattino**: Nessun limite inferiore, di conseguenza per ogni elemento della classe di partenza può anche non essere presente alcun elemento nella classe d'arrivo.

Esistono diverse notazioni grafiche, ma queste dispense fanno riferimento a quelle utilizzate durante gli esercizi del corso, quindi questa sarà la notazione comune di riferimento.

2.2.2 Associazioni Ternarie/con Attributi e Reificazione

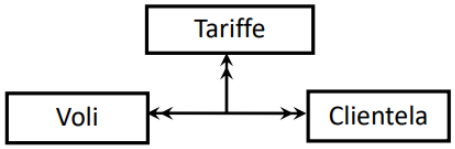
A volte le associazioni potrebbero complicarsi perchè:

1. Potrebbero avere la **necessità** di avere degli **attributi**. Ad esempio in un caso di associazione tra Clienti e Stanze, la tariffa potrebbe non essere nè attributo di Clienti e nè di Stanze. In questo caso si assegna un **attributo** all'**associazione**.



2. Potrei invece immaginare l'attributo come **vera** e **propria entità** di una specifica classe Tariffe. In quel caso non starei semplicemente dando un attributo

   all'associazione ma starei componendo un **associazione ternaria**.



Reificazione Si preferisce, nei casi illustrati sopra, semplificare la gestione dell'associazione tramite processo di reificazione, ossia la creazione di una classe di supporto aggiuntiva che permetta la gestione regolare delle associazioni viste prima. Si illustrano le reificazioni delle associazioni viste sopra:

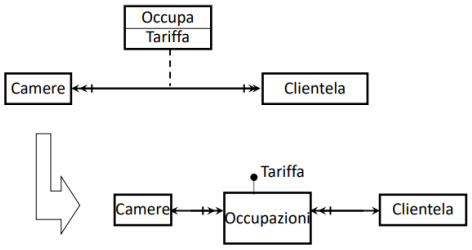


Figure 1: Reificazione di Associazione con Attributo

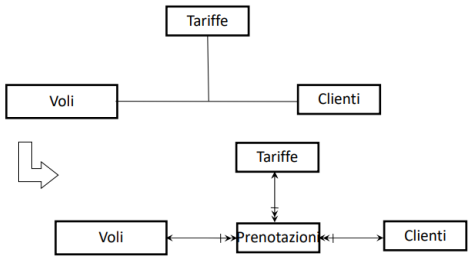


Figure 2: Reificazione di Associazione Ternaria

Un dettaglio da notare è la **cardinalità** sulla nuova classe di supporto, infatti in **direzione** della **nuova classe** sarà presente un associazione di **uno a molti**.

2.2.3 Sottoclassi e Tipologie

Una **sottoclasse** è un sottoinsieme di elementi di una classe, per i quali prevediamo di raccogliere ulteriori informazioni.

$$\text{Studenti} \subseteq \text{Persone}$$

$$\text{Libri Rari} \subseteq \text{Libri}$$

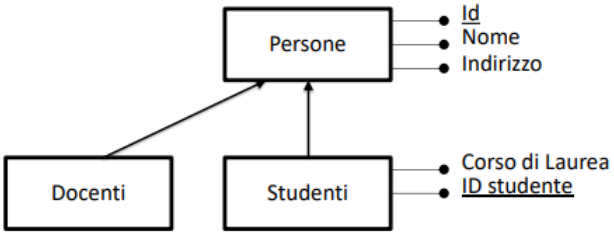
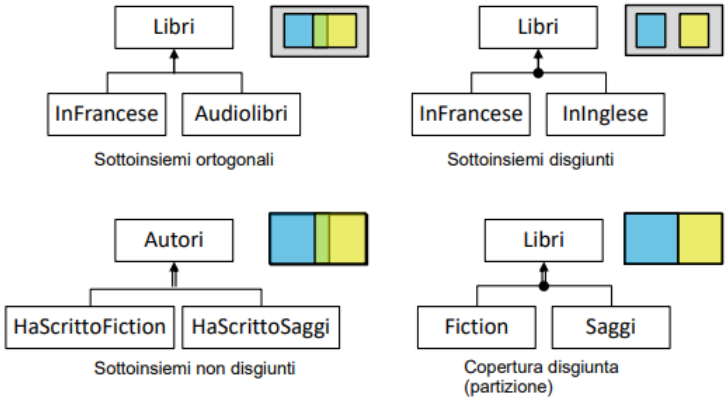


Figure 3: Notazione grafica di riferimento

Esistono varie tipologie di sottoclassi:



2.3 Progettazione Logica

Il **tipo** di **schema logico** presentato in questo corso è lo **schema relazionale**.

Basata su **schemi relazionali** detti informalmente **tabelle** di valori elementari con chiavi primarie:

Impiegati			
IdImpiegato	Nome	Stipendio	IdReparto*
232	Lucia	1200	Y1
143	Luigi	1500	X2

In questo contesto **non esiste differenza** tra **attributi** ed **associazioni** dato che le chiavi esterne permetteranno il **riferimento** ad **altre tabelle**.

Reparti	
IdReparto	Budget
Y1	100000
X2	200000

Solitamente quindi si definisce **uno dei campi** della **tabella** come **chiave primaria**, ossia che **identifica** univocamente **la riga**. Invece il campo che permette la dereferenziazione univoca di una riga in un altra tabella è detta **chiave esterna**, che corrisponde alla **chiave primaria** della **tabella esterna**.

Chiave e Superchiave Minimale Definiamo **superchiave** qualsiasi **insieme di attributi** che **non** può **ripetersi** in **righe diverse**. Si definisce **chiave** una **superchiave minimale**, ossia un insieme di attributi a cui non possiamo rimuovere alcun attributo. Sarà quindi scelta del progettista scegliere una **chiave primaria tra** tutte le **chiavi possibili**.

2.3.1 Da Schema Concettuale a Schema Logico - Fasi

Elenchiamo e descriviamo le fasi necessarie per il passaggio da schema **concettuale** a **logico**.

1. Aggiungere una **chiave primaria** artificiale ad ogni collezione che ne ha bisogno.
   (a) Una chiave deve essere **immutabile**, **muta**, ossia non deve portare con se alcuna informazione, ed **invisibile** agli utenti.
2. Tradurre le **associazioni** e le **inclusioni** in **chiavi esterne**.
   (a) Una associazione $1 - N$ diventano **chiavi esterne**.
   (b) Una associazione $M - N$ diventano **tabelle** con due chiavi esterne che puntano alle tabelle tra cui esiste l'associazione.
3. Tradurre gli **attributi multivalore** in **tabelle**.
4. **Appiattire** gli **attributi complessi**, ossia da struct a lista di parametri semplici.

Nello schema logico le **sottoclassi** vengono rappresentate come **nuove tabelle**, con chiavi esterne in direzione della superclasse.

Persone	
PId	Nome

Impiegati		
Reparto	Stipendio	PId*

Consulenti		
Progetto	Tariffa	PId*

3 Algebra Relazionale

Insieme di operatori su relazioni che restituiscono altre relazioni. La forza di questo sistema di operazioni è che sono componibili, e formano successivamente ulteriori operazioni.

3.1 Operatori Primitivi

Elenchiamo gli operatori primitivi:

1. **Proiezione**: Data una tabella R, la funzione $\Pi_{A,B}(R)$ genera una nuova tabella in cui si mostrano solo le colonne A, B della tabella R.
2. **Restrizione**: Data una tabella R, la funzione $\rho_{cond}(R)$ genera una nuova tabella in cui si mostrano solo le righe in cui vale la condizione *cond*.
3. **Unione**: Unione $R \cup S$, l'operazione può essere effettuata solo tra tabelle con lo stesso schema.
4. **Differenza**: Differenza $R - S$, l'operazione può essere effettuata solo tra tabelle con lo stesso schema.
5. **Prodotto Cartesiano**: Date due tabelle R ed S, produce una nuova tabella con tutte le possibili n-uple formate dal prodotto delle due tabelle.
6. **Ridenominazione**: Ridenominazione $\delta_{A \to B}(R)$, consiste informalmente nella ridenominazione dei campi che definiscono le colonne dello schema. Questo perchè altre operazioni binarie richiedono che le colonne di schemi diversi non abbiano gli stessi nomi.

3.2 Operatori Derivati

Elenchiamo gli operatori derivati:

1. **Intersezione**: Date due tabelle R ed S, restituisce le righe (tuple) che sono sia in R sia in S, dunque è necessario che le due tabelle abbiano lo **stesso schema**.

$$R \cap S = R - (R - S)$$

2. **Giunzione (Join)**: L'operazione $R \bowtie_{R.A=S.B} S$ combina le righe (tuple) di R ed S **solo quando** un attibuto A di R **è uguale** ad un attributo B di S. Questa operazione è derivata da:
$$R \bowtie_{R.A=S.B} S = \rho_{R.A=S.B}(R \times S)$$

3. **Giunzione Naturale**: L'operazione $R \, v \, S$ è una giunzione automatica su tutti gli attributi che hanno lo stesso nome in entrambe le relazioni, quindi non sono specificate condizioni.

Raggruppamento - Group By Data una tabella A si definisce uno dei suoi attributi (A_i) come **dimensione** e si genera una nuova tabella definendo funzioni (f_i) sulle altre colonne dette **misure**. Formalmente la definiamo quindi come $\Gamma_{\{A_i\}\{f_i\}}(R)$.

Studente	Count(*)	Min(Voto)
1	2	25
2	2	21

{Studente}γ{Count(*) ,Min(Voyo)}

Materia	Studente	Voto	Insegnante
DB	1	25	10
LPM	2	23	20
LCS	1	29	20
DB	2	21	30

4 Linguaggio SQL

Linguaggio di interrogazione su calcolo di multiset basato su delle clausole specifiche:

4.1 Clausole SQL

1. **SELECT**: Permette di selezionare specifici **campi della tabella** generata nella clausola FROM.
2. **FROM**: Permette la combinazione di tabelle. In questa clausola abbiamo modo di utilizzare anche la clausola JOIN per unire più tabelle tramite specifici campi. Il modo più comune per l'utilizzo della JOIN è:

$$FROM\ Studenti\ s\ JOIN\ Esami\ e\ ON\ s.SId = e.SId$$

3. **WHERE**: Permette di definire una restrizione sulle righe che soddisfano la condizione passata alla WHERE. Accade spesso di impostare una condizione per la WHERE con una sottoquery come parte dell'espressione di controllo. Esistono diversi tipi di sottoquery:
   (a) **Sottoquery restituisce un singolo valore**: La sottoquery calcola un valore (AVG, MAX, MIN...) e la WHERE utilizza questo valore nel confronto.
   (b) **Sottoquery restituisce una colonna**: Vengono utilizzati operatori specifici come IN, ANY, ALL con operando a destra il campo della colonna esterna e come operando a sinistra il risultato della sottoquery.

```
1    SELECT name
2    FROM employees
3    WHERE dept_id IN (
4        SELECT id
5        FROM departments
6        WHERE location = 'Rome'
7        );
8
```

   (c) **Sottoquery correlata**: Sottoquery che viene valutata per ogni riga della tabella in questione, in modo tale che si possa confrontare un valore a tutti quelli di un campo della tabella.

```
1    SELECT e.name, e.salary
2    FROM employees AS e
3    WHERE e.salary > (
4        SELECT AVG(salary)
5        FROM employees
6        WHERE dept_id = e.dept_id
7        );
8
```

4. **ORDER BY**: Permette l'ordinamento su un campo della tabella.

5. **GROUP BY**: Si definisce come nell'algebra relazionale con delle *dimensioni* e *misure*. Le *dimensioni* vanno riportate come argomenti della clausola **GROUP BY**, mentre le misure come *argomenti* della **SELECT**.

```
1    SELECT s.Nome, AVG(e.Voto)
2    FROM Studenti s, Esami e
3    WHERE s.SId = e. SId
4    GROUP BY s. SId
5
```

   Quindi si genera per ogni gruppo una linea.

6. **HAVING**: Questa clausola può essere applicata solo su dimensioni esplicite della GROUP BY e operazioni su attributi non dimensionali. Quindi la HAVING permette la cancellazione dei gruppi che violano la clausola HAVING. Quindi riassumendo tutte le clausole presentate in una pseudoquery:

```
1    SELECT ...
2    FROM ...
3    WHERE ...
4    GROUP BY ...
5    HAVING ...
6
```

   Questa query eseguirà questi passi:

   (a) Esegue le clausole FROM e WHERE, calcolando quindi una tabella di partenza.

   (b) Partiziona la tabella eseguendo il GROUP BY in base alle dimensioni fornite a questa clausola. Ogni gruppo diventa quindi una linea.

   (c) Elimina i gruppi che violano le clausole passate alla HAVING.

   (d) Proietta le colonne specificate nella clausola SELECT.

4.2 Valore NULL

In SQL è ammesso il valore *NULL* che definisce la non conoscenza di un valore di un campo. Questo però causa problemi in vari contesti, dato che andrà regolato il suo comportamento. Il valore *NULL* va letto come "non conosco il valore", quindi non si confronteranno i campi con un $= NULL$ ma esistono predicati appositi per verificare la presenza del *NULL*, ad esempio $IS\,NULL$.

4.3 Quantificazione Esistenziale/Universale in SQL

1. **Quantificazione Esistenziale**: In SQL è presente l'operatore EXISTS che ci permette di descrivere una condizione esistenziale scorrendo una tabella.

In linguaggio logico avremmo così definito la quantificazione esistenziale:

$$\{s.Nome \mid s \in Studenti \ \wedge \ \exists e \in Esami \, (e.SId = s.SId \wedge e.Voto = 27)\}$$

Come viene tradotta in SQL? Presentiamo un esempio utilizzando l'operatore EXISTS.

```
1    SELECT s.Nome
2    FROM Studenti s
3    WHERE EXISTS
4        (SELECT *
5        FROM Esami e
6        WHERE e.SId = s.SId AND e.Voto > 27)
7
```

2. **Quantificazione Universale**: In SQL non esiste un operatore di quantificazione universale esplicito, quindi il modo più comune è quello di utilizzare un **NOT EXISTS** in combinazione con la **condizione negata alla query interna**. Presentiamo anche in questo caso un esempio di quantificazione universale in linguaggio matematico e successivamente in SQL.

$$\{s.Nome \mid s \in Studenti \wedge \forall e \in Esami(e.SId = s.SId \wedge e.Voto = 27)\}$$

In SQL questa viene tradotta in

```
1    SELECT s.Nome
2    FROM Studenti s
3    WHERE NOT EXISTS
4        (SELECT *
5        FROM Esami e
6        WHERE e.SId = s.SId AND e.Voto <> 30)
7
```

NULL e Quantificazione Universale La quantificazione universale torna esito positivo sugli insiemi vuoti. Bisogna tener conto di questa caratteristica se non vogliamo nel risultato anche righe che erano nulle laddove si testava la condizione.

4.4 Outer Join

Citiamo prima il comportamento di una **Inner Natural Join**:

R

A	B
1	a
2	b
3	c

NATURAL JOIN

S

A	C
1	x
3	y
5	z

$\Longrightarrow$

R $\bowtie$ S

A	B	C
1	a	x
3	c	y

Si legano quindi solo in base alle chiavi esterne coincidenti.

Diverso è invece il comportamento della Outer Join:

1. **Left Outer Join**: Effettua una Natural Join e a questa aggiunge le righe della tabella a sinistra, aggiungendo nei *NULL* dove necessario:

R

A	B
1	a
2	b
3	c

LEFT JOIN

S

A	C
1	x
3	y
5	z

$\Longrightarrow$

R $\bowtie$ S

A	B	C
1	a	x
3	c	y
5		z

2. **Full Join**: Effettua una Natural Join e a questa aggiunge le righe di entrambe le tabelle, aggiungendo nei *NULL* dove necessario:

R

A	B
1	a
2	b
3	c

FULL JOIN

S

A	C
1	x
3	y
5	z

$\Longrightarrow$

R $\bowtie$ S

A	B	C
1	a	x
2	b	
3	c	y
5		z

Questi tipi di Join ci permettono di manipolare risultati nulli, ad esempio:

1. **Contare Esami di Ogni Studente, no Left Join**: Escludo chi non ha sostenuto alcun esame:

```
1    SELECT s.Nome, COUNT(*)
2    FROM Studenti s JOIN Esami e USING (SId)
```

2. **Contare Esami di Ogni Studente, con Left Join**: Includo chi non ha sostenuto alcun esame:

```
1    SELECT s.Nome, COUNT(e.Voto)
2    FROM Studenti s LEFT JOIN Esami e USING (SId)
3    GROUP BY s.SId, s.Nome
```

4.5 Programmazione e SQL

Si mostrano dei concetti più teorici riguardanti il SQL.

4.5.1 Approcci Integrazione SQL

Esistono vari approcci che permettono l'integrazione dell'SQL ai linguaggi di programmazione:

1. **Linguaggio Integrato**: PL/SQL definito ad esempio da Oracle è SQL esteso fino a diventare un linguaggio completo, quindi viene compilato contemporaneamente sia l'SQL sia la parte di programmazione standard.
    - **Vantaggio**: Viene compilato ed eseguito direttamente dal DBMS, permettendo operazioni come controlli per ruoli d'accesso in compilazione.
    - Oracle PL/SQL permette la lettura di tipi del database e il loro utilizzo per la dichiarazione di variabili. In questo caso il codice SQL viene utilizzato come *template*, interpretato secondo SQL e sostituito sintatticamente, per poi eseguire il linguaggio esterno.
2. **Linguaggio Convenzionale + API**: Si utilizza una libreria più un'interfaccia per l'interazione tra il linguaggio convenzionale e il DBMS.
    - In questo contesto, il linguaggio convenzionale non effettua controlli statici sull'SQL, poiché esso viene considerato come una **normale stringa**.
    - **Vantaggio**: Le query vengono inviate al DBMS come stringhe.
    - Tuttavia, non si risolve il disallineamento semantico (mismatch) dovuto a elementi come `NULL` o `MAX_INT`.
3. **SQL Ospitato**: Approccio che utilizza costrutti come `begin SQL` ed `end SQL`, permettendo la condivisione di variabili tra, ad esempio, Python e SQL.
    - Il precompilatore si occupa di compilare e di chiamare l'API, quindi i controlli a tempo di compilazione sono effettuati anche per l'SQL (differenza rilevante rispetto al caso precedente).
    - Possono emergere problemi dovuti alla traduzione SQL, talvolta difficili da individuare.
    - Possono sorgere inoltre problemi di differenza di espressività tra SQL (insiemi, record) e il linguaggio ospite.

Cursore/Buffer Meccanismo che permette di scannerizzare elementi, è associato ad una espressione SQL e permette lo scorrimento di dati riga per riga con operazioni di `FETCH`.

4.5.2 Creazione Tabelle

La definizione di una tabella in SQL è importante da saper leggere, anche se raramente viene richiesto di crearne una esplicitamente.

```
CREATE TABLE Employees(
    Code CHAR(8) NOT NULL,
    Name CHAR(20),
    Birthyear INTEGER CHECK (Birthyear < 2005),
    Qualification CHAR(20) DEFAULT 'Employee',
    Supervisor CHAR(8),
    PRIMARY KEY pk_Employees (Code),
    FOREIGN KEY fk_Employees (Supervisor)
        REFERENCES Employees
);

CREATE TABLE Dependents(
    Name CHAR(20),
    Birthyear INTEGER,
    EmployeeCode CHAR(8),
    FOREIGN KEY fk_Dependents (EmployeeCode)
        REFERENCES Employees
);
```

- Nella **foreign key** va specificato quale attributo punta a quale tabella.
- I comandi `DROP` e `ALTER` permettono rispettivamente di cancellare e modificare una tabella.
- **Vincolo di Foreign Key**: se un valore referenziato viene eliminato, è possibile definire comportamenti diversi, come:
  - Impostare il valore a `NULL`.
  - Usare `ON DELETE CASCADE`, che propaga la cancellazione.

4.5.3 Viste e Viste Materializzate

Viste Materializzate Una vista materializzata è una **tabella fisica** costruita come risultato dell'esecuzione di una query su altre tabelle.

```
CREATE TABLE Name SelectExpression;

CREATE TABLE Supervisors AS
SELECT Code, Name, Qualification, Salary
FROM Employees
WHERE Supervisor IS NULL;
```

Questo approccio è utile quando i dati cambiano raramente, poiché le modifiche nella tabella originale non si riflettono automaticamente nella vista materializzata.

Viste Le viste standard sono **viste virtuali**, cioè query memorizzate che non vengono eseguite finché non vengono richieste.

```
CREATE VIEW Name [(Attribute {, Attribute})]
AS SelectExpression [WITH CHECK OPTION];

CREATE VIEW Supervisors AS
SELECT Code, Name, Qualification, Salary
FROM Employees
WHERE Supervisor IS NULL;
```

Generalmente, le viste non vengono aggiornate direttamente perché non contengono dati reali.

4.6 Motivazioni per l'uso delle viste

- Permettono di nascondere dati, garantendo **indipendenza logica**.
- Consentono **diverse rappresentazioni** degli stessi dati senza duplicazione.
- Possono **semplificare query** complesse.
  - Ad esempio, quando si devono comporre più operazioni `GROUP BY`, è possibile costruire una vista intermedia.
  - Con SQL moderno è possibile usare subquery nella `FROM`, quindi l'uso delle viste per questo scopo è meno frequente ma resta utile per chiarezza.

4.6.1 Procedure e Funzioni

Alcuni DBMS permettono di definire procedure e funzioni interne. Un esempio è il linguaggio `PL/SQL` di Oracle.

```
CREATE FUNCTION countStudents IS
DECLARE
    total INTEGER;
BEGIN
    SELECT COUNT(*) INTO total FROM STUDENTI;
    RETURN (total);
END;
```

4.6.2 Trigger

I **trigger** sono funzioni che vengono eseguite automaticamente al verificarsi di un determinato evento (ad esempio prima o dopo un `INSERT`, `UPDATE` o `DELETE`).

```
CREATE TRIGGER SalaryCheck
BEFORE INSERT ON Employees
DECLARE
    AvgSalary FLOAT;
BEGIN
    SELECT AVG(Salary) INTO AvgSalary
    FROM Employees
    WHERE Department = :new.Department;

    IF :new.Salary > 2 * AvgSalary THEN
        RAISE_APPLICATION_ERROR(-2061, 'Salary too high');
    END IF;
END;
```

- I trigger sono usati quando i vincoli coinvolgono **più tabelle**, poiché una semplice clausola `CHECK` opera solo sulla tupla corrente.
- Possono essere usati per mantenere valori derivati o contatori aggiornati automaticamente.
- I trigger possono vincolare anche operazioni di `DELETE`, mentre `CHECK` non può farlo.

4.6.3 Access Control nel Database

Il creatore di un database ha permessi completi: `CREATE`, `ALTER`, `DROP`. Può inoltre concedere permessi ad altri utenti, anche a livello di singola colonna.

```
GRANT Privileges ON Object
TO Users [ WITH GRANT OPTION ];
```

Indici Gli indici sono strutture dati utilizzate per velocizzare l'accesso ai dati nelle query. Tuttavia, un numero eccessivo di indici rallenta **inserimenti, modifiche e cancellazioni**, poiché anche gli indici devono essere aggiornati.

Catalogo - Metadata Il **catalogo** (o dizionario dati) contiene metadati sul database. Ad esempio:

- Elenco delle tabelle del DB
- Tipi degli attributi
- Vincoli definiti

Ogni riga nella *tabella delle tabelle* rappresenta una tabella del database.

5 Teoria della Normalizzazione

La teoria della normalizzazione permette di definire equivalenze tra schemi relazionali ed anomalie in essi.

Buona Progettazione di Schemi Relazionali Si basa sull'analisi di quattro proprietà:

- **Semantica degli Attributi**: Progettazione mirata all'utilizzo di attributi che appartengano solo alla classe di quello schema.
- **Ridondanza**: Progettazione mirata a schemi in cui non siano presenti anomalie in caso di inserimento, cancellazione o modifica.
- **Valori Nulli**: Per quanto possibile, si progettano schemi in modo tale da evitare attributi che possono essere frequentemente nulli.
- **Tuple Spurie**: Le giunzioni tra schemi non dovrebbero mai generare tuple spurie.

Obiettivi della Teoria Normalizzazione Fornire strumenti formali per la progettazione di basi di dati che non presentino anomalie, e grazie all'equivalenza, permettere anche la ricerca di uno schema equivalente ad uno già esistente ma senza anomalie. I **problemi** quindi di cui si occupa la **teoria della normalizzazione** sono:

- **Equivalenza tra Schemi**: Definire quando due schemi siano equivalenti.
- **Qualità degli Schemi**: Definire criteri di bontà di schemi.
- **Trasformazione di Schemi**: Definire metodi algoritmici per ottenere da uno schema un altro schema che sia migliore di quello di partenza.

Schema di Relazione Universale Definiamo formalmente l'insieme universo degli attributi. Lo schema di relazione universale U di una base di dati relazionale ha come attributi l'unione degli attributi di tutte le relazioni della base di dati.

Notazione Definiamo un po' di notazione utile:

- A, B, C, A_1, A_2 indicano singoli attributi.
- T, X, Y, X_1 indicano insiemi di attributi.
- Potenziali Abbreviazioni:
  - XY è abbreviazione per $X \cup Y$.
  - AB è abbreviazione per $\{A, B\}$.
  - XA è abbreviazione per $X \cup \{A\}$.
- $R(T)$ è un generico schema, r è una sua istanza e t è una n-upla di r.
- Se $X \subseteq T$ allora $t[X]$ indica l'n-upla ottenuta da t considerando solo gli attributi in X.

5.1 Dipendenze Funzionali

Vogliamo formalizzare la **nozione di schema senza anomalie**, occorre quindi **descrivere formalmente** la **semantica** dei dati in una istanza dello schema relazionale. Quindi definiamo come **aspetto estensionale** quello che si riferisce al **contenuto dei dati**, ed **aspetto intensionale** quello che si riferisce al **significato dei dati**.

5.1.1 Definizione Formale di Dipendenza Funzionale e Significato

Dato uno schema $R(T)$ e $X, Y \subseteq T$, una dipendenza funzionale `DF` è un **vincolo** su R del tipo $X \to Y$, ossia

$$X \text{ determina funzionalmente } Y$$

o in maniera equivalente

$$Y \text{ è determinata funzionalmente da } X$$

se vale che per ogni r istanza valida di R un valore di X determina in modo univoco un valore di Y, formalmente definito come:

- $\forall r$ istanza valida di R.
- $\forall t_1, t_2 \in r$ vale che $t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]$

Spiegazione Definizione In un istanza r di $R(T)$ è soddisfatta la dipendenza $X \to Y$ se per ogni coppia di n-uple $t_1, t_2 \in r$ è vero che $t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]$.

Caratteristiche Dipendenze Funzionali Queste dipendenze sono definite solo all'interno di uno schema di relazione e non possono esistere tra attributi di schemi diversi. Oltre a questo sono proprietà intensionali, sono legate alla semantica assegnata all'attributo e non possono quindi essere inferite sulla base dell'osservazione di alcune istanze della relazione.

5.1.2 Definizioni Formali di Chiave, Superchiave e Chiave Primaria

Dato un insieme K di attributi per uno schema di relazione r:

- **Superchiave**: L'insieme K è superchiave se **non contiene** due n-uple distinte t_1, t_2 con $t_1[K] = t_2[K]$. Quindi l'insieme di attributi K identifica univocamente una n-upla.
- **Chiave**: L'insieme K è **chiave se è superchiave minimale**, ossia non contiene un altra superchiave al suo interno. Quindi se eliminassimo degli attributi questa non sarebbe più una superchiave.
- **Chiave Primaria**: Una delle chiavi dello schema di relazione, solitamente si preferisce quella con meno attributi.

5.1.3 Dipendenza Funzionale e Soddisfacibilità su Istanze

Introduciamo un nuovo operatore $\models$ (soddisfa):

- un'istanza r_0 di R **soddisfa** la DF $X \to Y$ se:

$$r_0 \models X \to Y$$

  cioè se:

$$\forall t_1, t_2 \in r_0,\ t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]$$

- un istanza r_0 **soddisfa** un insieme F di DF se:

$$\forall X \to Y \in Fvaler_0 \models X \to Y$$

  cioè se

$$r_0 \models X \to Y \text{ se } \forall t_1, t_2 \in r_0 \text{ vale che } t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]$$

5.1.4 Dipendenze Funzionali Atomiche e Banali

- **Atomica**: Ogni dipendenza del tipo $X \to A_1 A_2 \cdots A_n$ è scomponibile in $X \to A_1, X \to A_2 \cdots, X \to A_n$. La seconda tipologia è detta **atomica**.
- **Banale**: Una dipendenza del tipo $Progetto \to Progetto$ è detta **banale**.

5.2 Utilizzo di Dipendenze Funzionali

Vogliamo utilizzare uno schema di relazione e le sue dipendenze per ricavarne altre, quindi definiamo da adesso in poi lo schema come $R(T, F)$ con F insieme di dipendenze funzionali su T.

5.2.1 Derivazione di Dipendenze Funzionali

Basandoci sulla definizione data prima di soddisfacibilità di istanze, dato $R(T, F)$ diciamo che $F \models X \to Y$ se ogni istanza $r \in R(T)$ che soddisfa F soddisfa anche $X \to Y$.

5.2.2 Assiomatizzazione delle Dipendenze Funzionali

Sia RI un insieme di regole di inferenza per F, utilizzeremo RI per derivare altre dipendenze funzionali a partire da F.

Indichiamo con $F \vdash X \to Y$ il fatto che $X \to Y$ sia derivabile da F usando RI.

Correttezza e Completezza di un Insieme di Regole di Inferenza Esistono due proprietà principali di un sistema d'inferenza:

- **Correttezza**: L'insieme RI se $F \vdash X \to Y$ implica $F \models X \to Y$.
- **Completezza**: L'insieme RI se $F \models X \to Y$ implica $F \vdash X \to Y$.

5.2.3 Assiomi di Armstrong e Derivazioni

Gli assiomi di Armstrong sono l'insieme noto corretto e completo di regole di inferenza. Elenchiamo queste regole:

- **Riflessività**: se $Y \subseteq X$ allora $X \to Y$.
- **Arricchimento**: se $X \to Y$, $Z \subseteq T$ allora $XZ \to YZ$.
- **Transitività**: se $X \to Y$, $Y \to Z$ allora $X \to Z$

Da queste possiamo derivarne altre:

- **Unione**: $\{X \to Y, X \to Z\} \vdash X \to YZ$
- **Decomposizione**: $\{X \to YZ\} \vdash X \to Y$
- **Indebolimento**: $\{X \to Y\} \vdash XZ \to Y$
- **Identità**: $\{\} \vdash X \to Y$

Teorema di Correttezza e Completezza Assiomi di Armstrong Gli assiomi di Armstrong sono corretti e completi. Attraverso gli assiomi stessi si può mostrare l'equivalenza tra implicazione logica $\models$ e derivazione $\vdash$. Questo perchè se una dipendenza è derivabile con gli assiomi di Armstrong allora è anche implicata logicamente (per la correttezza degli assiomi), e viceversa se una dipendenza è implicata logicamente allora è anche derivabile dagli assiomi (per completezza degli assiomi).

- **Correttezza**: $\forall f \;\; F \vdash f \Rightarrow F \models f$
- **Completezza**: $\forall f \;\; F \models f \Rightarrow F \vdash f$

5.2.4 Chiusura di un Insieme

Dato un insieme F di DF la chiusura di F è $F^+ = \{X \to Y \mid F \vdash X \to Y\}$.

Contestualizzando ad una $R(T, F)$ data con $X \subseteq T$ la chiusura di X rispetto ad F denotata con X_F^+ è

$$X_F^+ = \{A_i \in T \mid F \vdash X \to A_i\}$$

5.2.5 Algoritmo della Chiusura Lenta

Vogliamo decidere, data una DF, se questa appartiene a F^+. Applicare ripetutamente gli assiomi di Armstrong ha una complessità esponenziale rispetto al numero di attributi dello schema. Si preferisce quindi utilizzare un algoritmo basato su uno specifico teorema.

Teorema La $DF : X \to Y$ è derivabile da F se e solo se Y è sottoinsieme della chiusura di X rispetto ad F formalmente definito come:

$$F \vdash X \to Y \Leftrightarrow Y \subseteq X_F^+$$

Definizione di Chiavi tramite Dipendenze e Chiusura

- **Superchiave**: Dato lo schema $R(T, F)$, un insieme di attributi $W \subseteq T$ è una **superchiave** di R se $W \to T \in F^+$.
- **Chiave**: Dato lo schema $R(T, F)$, un insieme di attributi $W \subseteq T$ è una **chiave** di R se W è una superchiave e non esiste un sottoinsieme stretto di W che sia superchiave di R.
- **Attributo Primo**: Dato lo schema $R(T, F)$, un attributo $A \in T$ si dice **attributo primo** se e solo se appartiene ad almeno una chiave, altrimenti si dice non primo.

Proprietà Interessanti per Ricerca di Chiavi L'algoritmo per trovare tutte le chiavi si basa su due proprietà:

- Se un attributo A di T **non appare a destra** di alcuna dipendenza in F, allora A **appartiene** ad ogni chiave di R, altrimenti non può essere determinato.
- Se un attributo A di T **appare a destra** di qualche dipendenza in F, **ma non appare a sinistra** di alcuna dipendenza non banale, allora A **non appartiene ad alcuna chiave**.

Esempio di Derivazione Dato $F = \{DB \to E, B \to C, A \to B\}$ trovare $(AD)^+$:

$$X^+ = AD = ADB = ADBE = ADBEC$$

Quindi data questa produzione, possiamo ad esempio notare che AD è superchiave perchè contiene tutti gli attributi ed A **non è superchiave** perchè la produzione $A \to B, A \to BC$ si ferma.

5.3 Copertura Canonica e Decomposizione

L'obiettivo di questo sottocapitolo è quello di definire schemi che siano equivalenti, dopo operazioni di decomposizione.

Definizione Copertura/Equivalenza Due insiemi di DF, F e G sullo schema R sono **equivalenti** $F \equiv G$ se e solo se $F^+ = G^+$. Se vale questa condizione allora si dice che F è **copertura** di G e viceversa.

5.3.1 Definizioni Formali di Attributo Estraneo e Dipendenza Ridondante

- **Definizione Copertura Canonica**: Data una $X \to Y \in F$, si dice che X contiene un **attributo estraneo** se e solo se $(X - \{A_i\}) \to Y \in F^+$, quindi vale

  che $F \vdash (X - A_i) \to Y$.
- **Dipendenza Funzionale Ridondante**: $X \to Y$ è detta ridondante se e solo se $(F - \{X \to Y\})^+ = F^+$ o equivalentemente $F - \{X \to Y\} \vdash X \to Y$.

5.3.2 Definizione Copertura Canonica

F è detta **copertura canonica** se rispetta queste condizioni:

- Tutte le dipendenze DF in F sono atomiche.
- Non esistono attributi estranei.
- Nessuna dipendenza in F è ridondante.

Sulla base di questa definizione è possibile definire il seguente teorema:

Teorema Per ogni insieme di dipendenze F esiste una copertura canonica.

Quindi avendo un teorema per l'esistenza di copertura canonica, possiamo definire un algoritmo per calcolare una copertura canonica, che attraversa tre fasi per validare le proprietà richieste dalla definizione di copertura canonica:

- Trasformare tutte le dipendenze in forma atomica.
- Eliminare tutti gli attributi canonici.
- Eliminare tutte le dipendenze ridondanti.

Tipi di Ridondanza Esistono due tipi differenti di ridondanza:

- **Ridondanza Concettuale**: Non esistono informazioni duplicate ma informazioni che sono ricavabili da altre già presenti nella base di dati.
- **Ridondanza Logica**: Esistono informazioni duplicate.

5.3.3 Definizione di Decomposizione di Schemi e Proprietà Attese - Conservazione Dati e Dipendenze

Dato uno schema $R(T)$, $\rho\{R_1(T_1), \cdots, R_k(T_k)\}$ è una **decomposizione** di R se e solo se $T_1 \cup \cdots \cup T_k = T$.

Proprietà Attese Dopo una buona operazione di decomposizione ci aspettiamo che siano rispettate due condizioni:

- **Conservazione dei Dati**: $\rho\{R_1(T_1), \cdots, R_k(T_k)\}$ è una decomposizione di uno schema $R(T)$ che preserva i dati se e solo se per ogni istanza valida r di R vale che:

  $$r = (\pi_{T1} r) \bowtie (\pi_{T2} r) \bowtie \cdots \bowtie (\pi_{Tk} r)$$

  Quindi dalla definizione di giunzione naturale scaturisce questo teorema:

  - **Teorema**: Se $\rho = \{R_1(T_1), \cdots, R_k(T_k)\}$ è una decomposizione di $R(T)$, allora per ogni istanza r di R vale che $r \subseteq (\pi_{T1} r) \bowtie (\pi_{T2} r) \bowtie \cdots \bowtie (\pi_{Tk} r)$.

  Quindi il vincolo $=$ della definizione non lascia spazio ad altre n-uple, che sono invece potenzialmente presenti nel teorema che si basa sul operatore $\subseteq$. Quindi se la decomposizione avviene senza perdite di dati allora questo vuol dire che non verrà generata alcuna n-upla spuria.

Teorema Informale Decomposizione Senza Perdite Se l'insieme di attributi comuni alle due relazioni $X_1 \cap X_2$ **è chiave** per **almeno una delle due relazioni** decomposte allora la **decomposizione** è **senza perdite**. Nel caso binario questo diventa:

  - Sia $R(T, F)$ uno schema di relazione, la decomposizione $\rho = \{R_1(T_1), R_2(T_2)\}$ preserva i dati se e solo se $T_1 \cap T_2 \to T_1 \in F^+$ oppure $T_1 \cap T_2 \to T_2 \in F^+$.

- **Preservazione di Dipendenze**: Dato lo schema $R(T, F)$, e $T_1 \subseteq T$, la proiezione di F su T_1 è definita come:

$$\pi_{T1}(F) = \{X \to Y \in F^+ \mid XY \subseteq T_1\}$$

  Informalmente stiamo quindi dicendo che una decomposizione preserva le **dipendenze** se **ciascuna** di queste dello schema originario coinvolge attributi che compaiono **tutti insieme in uno degli schemi decomposti**.

5.4 Forme Normali

Una forma normale è una proprietà di una base di dati relazionale che ne garantisce la qualità, solitamente quando una relazione non è normalizzata presenta ridondanze o si presta a comportamenti poco desiderabili durante gli aggiornamenti.

Tipi di Forme Normali Ogni tipo di forma normale definisce specifici vincoli:

- **1FN (Prima Forma Normale)**: Impone una restrizione sul tipo di una relazione: ogni attributo ha un tipo elementare.
- **2FN, 3FN e BCNF**: Impongono restrizioni sulle dipendenze e la Boyce-Codd BCNF è la più naturale e restrittiva.

5.4.1 BCNF - Forma Normale di Boyce-Codd

Una relazione r si dice in BCNF se **per ogni dipendenza funzionale** non banale del tipo $X \to Y$ definita su essa, X contiene una chiave K di r, ovvero X **è superchiave**.

Definizione Formale $R(T, F)$ è in BCNF $\Leftrightarrow$ per ogni $X \to A \in F^+$ non banale, ossia vale che $A \notin X$ quindi X è una superchiave.

Teorema $R(T, F)$ è in BCNF $\Leftrightarrow$ per ogni $X \to A \in F$ non banale vale che X è superchiave.

Corollario $R(T, F)$ con F in copertura canonica è in BCNF $\Leftrightarrow$ per ogni DF atomica non banale $X \to A \in F$ vale che X è una superchiave.

5.4.2 Algoritmo di Analisi

Questo algoritmo prende in input una relazione $R(T, F)$ con F copertura canonica e produce in output un $\rho = \{R_1, R_2, R_m\}$ decomposizione in BCNF che preserva i dati. Possiamo definirne uno pseudocodice:

```
while esiste in ρ una Ri(Ti, Fi) non in BCNF per la DF X → A
    //incrememta contatore relazioni
    n=n+1
    //chiusura di X
    Ta = X+
    //proiezione DF rispetto a Ta
    Fa = π Ta(Fi)
    //rimozione da attrivuti non in X ma nella sua chiusura
    //riaggiunta X alla fine
    Tb = Ti - X+ + X
    //rimuove R non BCNF per X → A  e inserisce quelle corrette
     ρ  = ρ  −  Ri  +  Ri  < Ta, Fa >,  Rn < Tb, Fb >
```

Questo algoritmo **preserva i dati ma non necessariamente le dipendenze**.

In alcune occasioni bisogna ricorrere a forme di normalizzazione indebolita. Mostriamo ad esempio la 3FN che consente di ottenere decomposizioni senza perdite e che preservano tutte le dipendenze.

5.4.3 3FN - Terza Forma Normale

Rilassiamo il vincolo della definizione precedente mettendo il vincolo della BCNF in XOR con la possibilità di A di essere **attributo primo**.

Definizione $R(T, F)$ è in 3FN se per ogni $X \to A \in F^+$, con $A \notin X$ vale che X è una **superchiave oppure** A è **attributo primo**.

Teorema $R(T, F)$ è in 3FN se per ogni $X \to A \in F$ non banale vale che X è una superchiave oppure A è primo, ossia è contenuto in almeno una chiave.

Esistenza 3FN e Caratteristiche Elenchiamo alcune caratteristiche della 3FN:

- **Esistenze e Proprietà Garantite**: Ogni schema del tipo $R(T, F)$ **ammette sempre** una decomposizione che **preserva dati**, **dipendenze ed è in 3FN**.
- **Tempo Ottenimento 3FN**: La decomposizione 3FN può essere ottenuta in tempo polinomiale.
- **Meno Restrizione**: Essendo meno restrittiva, la 3FN accetta anche schemi che presentano anomalie, in particolare la 3FN **tollera le ridondanze sui dati**.

5.4.4 Algoritmo di Sintesi

L'interesse per la 3FN è derivato dal fatto che ci permette di **decomporre qualsiasi schema** in un **insieme di relazioni in 3FN** tramite un algoritmo semplice e **preservando dati e dipendenze**.

In maniera approssimata potremmo quindi dire che l'idea dell'algoritmo sia quella di partizionare una copertura canonica G data in gruppi G_i tali che tutte le dipendenze in ogni G_i abbiano la stessa parte sinistra. Quindi da ogni G_i si definisce uno schema di relazione composto da tutti gli attributi che vi appaiono, la cui chiave, detta chiave sintetizzata, è la parte sinistra comune.

Formalizzazione Algoritmo di Sintesi in Versione Base Definiamo l'effettivo algoritmo, descrivendolo a passi:

- **Input**: Insieme R di attributi ed un insieme F di dipendenze su R.
- **Output**: Decomposizione $\rho = \{S_i\}_{i=1\cdots n}$ di R tale che **preservi dati e dipendenze** ed ogni S_i sia in 3FN rispetto alle proiezioni di F su S_i.
- **Fasi**: L'algoritmo è caratterizzato da queste fasi:
  1. Trova una copertura canonica G di F e poni $\rho = \{\ \}$.
  2. Sostituisce in G ogni insieme $X \to A_1, \cdots, X \to A_h$ di dipendenze con lo stesso determinante, con la dipendenza $X \to A_1 \cdots A_h$.
  3. Per ogni dipendenza $X \to Y$ in G, metti uno schema con attributi XY in ρ.
  4. Elimina ogni schema di ρ contenuto in un altro schema di ρ.
  5. Se la decomposizione **non** contiene alcuno schema i cui attributi costituiscano una superchiave per R, aggiungi ad essa lo schema con attributi W, con W una chiave di R.

6 Realizzazione di un DBMS

6.1 Architettura dei DBMS e Gestione della Memoria

Lo studio della realizzazione dei DBMS è indispensabile per utilizzare tali sistemi in modo efficace. È necessario conoscere l'architettura dei DBMS relazionali centralizzati e le tecniche utilizzate per implementare le funzionalità essenziali, come la gestione dei dati, delle interrogazioni, della concorrenza e dell'affidabilità.

6.1.1 Gerarchia delle Memorie e Implicazioni per i DBMS

La memoria di un sistema di calcolo è organizzata in una gerarchia. Ai livelli più alti si trovano memorie di piccola dimensione, molto veloci e costose; scendendo lungo la gerarchia, la dimensione aumenta mentre velocità e costo diminuiscono.

Le prestazioni di una memoria sono misurate in termini di tempo di accesso, dato dalla somma della latenza (tempo per accedere al primo byte) e del tempo di trasferimento dei dati:

$$\text{Tempo di accesso} = \text{latenza} + \frac{\text{dimensione dati da trasferire}}{\text{velocità di trasferimento}}$$

A causa delle loro grandi dimensioni, i database (DB) risiedono normalmente su dischi o altri tipi di dispositivi. I dati devono essere trasferiti in memoria centrale per l'elaborazione da parte del DBMS.

Trasferimento Dati e Ottimizzazione Il trasferimento dei dati non avviene a livello di singole tuple, ma in termini di **blocchi** (o **pagine**, quando sono in memoria centrale). Poiché le operazioni di I/O (Input/Output) spesso costituiscono il **collo di bottiglia** del sistema, è cruciale ottimizzare l'implementazione fisica del DB attraverso:

- Organizzazione efficiente delle tuple su disco.
- Strutture di accesso efficienti.
- Gestione efficiente dei buffer in memoria.
- Strategie di esecuzione efficienti per le query.

Struttura degli Hard Disk Un hard disk (HD) è un dispositivo elettro-meccanico che conserva informazioni magnetiche su un supporto rotante (piatto). I dischi metallici ruotano a velocità costante e le testine di lettura si muovono radialmente. Una traccia è suddivisa in settori di dimensione fissa, raggruppati logicamente in **blocchi**, che sono l'unità di trasferimento. Trasferire un blocco richiede un tempo di posizionamento delle testine (seek time), un tempo di latenza rotazionale e il tempo di trasferimento vero e proprio (che è spesso trascurabile).

Pagine e Località Un blocco (o pagina) è una sequenza contigua di settori su una traccia, ed è l'unità di I/O utilizzata dal DBMS. La dimensione tipica di una pagina varia da 4 a 64 KB.

- Pagine piccole comportano un maggior numero di operazioni di I/O.

- Pagine grandi aumentano la frammentazione interna (pagine parzialmente riempite).

Importanza Località L'importanza della località è critica, leggere un file memorizzato in settori consecutivi è estremamente più veloce rispetto a leggerlo da blocchi distribuiti a caso.

6.1.2 Gestore della Memoria Permanente e del Buffer

Gestore Memoria Permanente Fornisce un'astrazione della memoria permanente in termini di file logici di pagine fisiche (blocchi), nascondendo i dettagli dei dischi e del sistema operativo.

Gestore del Buffer Si occupa del trasferimento delle pagine tra la memoria temporanea (**Buffer Pool**) e la memoria permanente. Offre agli altri livelli del DBMS una visione della memoria permanente come un insieme di pagine utilizzabili nel buffer, astraendo da quando esse vengono trasferite.

Nel Buffer Pool, ogni pagina ha informazioni associate:

- **NumeroSpilli (Pin count)**: Contatore incrementato quando una pagina viene richiesta dal DBMS (`getAndPinPage`) e decrementato quando viene rilasciata (`unPinPage`). Fino a quando il pin count è maggiore di zero, la pagina non può essere rimpiazzata.
- **SeModificata (dirty/non dirty)**: Indica se la pagina è stata modificata (*dirty* o "sporca"). Se una pagina è *dirty*, deve essere riscritta su disco (`flushPage`) prima di essere rimpiazzata.

Politiche di Rimpiazzamento La politica comune nei sistemi operativi è **LRU** (Least Recently Used). Tuttavia, nei DBMS, LRU non è sempre la scelta migliore perché il "pattern di accesso" ai dati è spesso noto. Ad esempio, per alcuni algoritmi di *join*, la politica migliore può essere **MRU** (Most Recently Used), che rimpiazza la pagina utilizzata più di recente. La bontà di una politica è misurata dall'**hit ratio**, ovvero la frazione di richieste che non provocano un'operazione di I/O.

6.1.3 Strutture e Organizzazioni di Memorizzazione

Struttura Logica di Una Pagina Una pagina ha una struttura logica che include **informazioni di servizio** e un'area contenente le stringhe che rappresentano i record. Il problema del riferimento ai record è risolto tramite l'identificatore **RID**: una coppia `(PID della pagina, Slot)`. L'organizzazione a slot permette la riallocazione interna del record senza modificarne il RID.

Tipi di Organizzazioni Le organizzazioni sono caratterizzate da occupazione di memoria e costo delle operazioni (ricerca, modifica, inserzione).

- **Organizzazione Seriale (Heap file)**: I dati sono memorizzati in modo disordinato (nell'ordine di inserzione). È semplice e a basso costo di memoria, ma poco efficiente per grandi quantità di dati. È l'organizzazione standard di ogni DBMS.

- **Organizzazione Sequenziale**: I dati sono ordinati sul valore di uno o più attributi. Questo facilita le operazioni SQL di ordinamento, raggruppamento e ricerca (`ORDER BY, GROUP BY, WHERE`) sugli attributi ordinati. Tuttavia, è più complessa da gestire in caso di aggiornamento/cancellazione/inserimento, richiedendo periodiche riorganizzazioni. In un file sequenziale ordinato, la ricerca binaria di un record richiede circa $\lceil \log_2 b \rceil + 1$ accessi a blocco/pagina, dove b è il numero di blocchi.
- **Organizzazioni per Chiave**: L'obiettivo è trovare un record, nota la chiave, con il minor numero possibile di accessi a disco (idealmente 1 accesso) [19]. Le alternative sono il metodo **procedurale (hash)** o il metodo **tabellare (indice)**.

Hash File (Metodo Procedurale Statico) I record vengono allocati in una pagina il cui indirizzo è determinato dal valore della chiave tramite una funzione hash $H(\text{key}) \rightarrow$ page address. Le collisioni (overflow) sono gestite di solito con liste collegate (*linked lists*).

- È l'organizzazione più efficiente per l'accesso diretto (puntuale) basato su valori della chiave.
- Non è efficiente per ricerche basate su intervalli di valori (*range queries*).
- Funziona bene solo con file la cui dimensione non varia molto nel tempo (procedurale statico).

6.1.4 Indici: Metodo Tabellare, B-tree e B+-tree

Metodo Tabellare (Indice) Per le ricerche per chiave e per intervallo è preferibile il metodo tabellare, che utilizza un **indice**: un insieme ordinato di coppie $(k, r(k))$, dove k è il valore della chiave e $r(k)$ è il riferimento al record. L'indice è una struttura che contiene informazioni sulla posizione di memorizzazione delle tuple in base al valore del campo chiave. La realizzazione degli indici avviene tipicamente attraverso strutture ad albero multi-livello.

B-tree e B+-tree La struttura più usata e ottimizzata dai DBMS per la gestione degli indici è il **B+-albero**.

Albero di Ricerca di Ordine P Un albero di ricerca di ordine P è un albero i cui nodi contengono al più $P - 1$ valori di ricerca (search value) e P puntatori. Tali alberi possono essere memorizzati su disco, con ogni nodo assegnato a una pagina. Gli algoritmi di inserimento e cancellazione devono garantire il mantenimento dei vincoli, sebbene non garantiscano sempre che l'albero sia bilanciato (nodi foglia allo stesso livello).

B-tree Un B-tree è una soluzione per mantenere l'albero bilanciato. Un nodo interno di un B-tree contiene tree pointer (P_i), chiavi di ricerca (K_i) e data pointer (Pr_i) che puntano al record o alla pagina che lo contiene. Tutti i nodi foglia sono posti allo stesso livello.

B+-tree Un B+-tree è una variazione del B-tree in cui i **data pointer sono memorizzati solo nei nodi foglia**. I nodi foglia sono generalmente collegati tra loro (relazionati) per facilitare le *range query*. I nodi interni hanno la funzione esclusiva di guidare la ricerca.

- **Nodi Interni**: Hanno la forma $\langle P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q \rangle$, dove P_i sono solo *tree pointer*.
- **Nodi Foglia**: Hanno la forma $\langle \langle K_1, Pr_1 \rangle, \ldots, \langle K_q, Pr_q \rangle, P_{\text{next}} \rangle$, dove Pr_i sono i *data pointer* e P_{next} punta al nodo foglia successivo.

Tipologie di Indici

- **Indice Primario**: La **chiave di ordinamento** del file sequenziale **coincide** con la **chiave di ricerca** dell'indice. Esiste un record nell'indice per ogni blocco nel file di dati.
- **Indice Secondario**: La **chiave di ordinamento** e la **chiave di ricerca sono diverse**. Può essere definito su un campo non chiave (anche con valori duplicati). In questo caso quindi il **primo indice** è utilizzato per **mantenere l'ordine**, mentre il **secondo indice** è un **effettivo puntatore** al blocco o al record (RID).

6.1.5 Tecniche di Ordinamento (Sort)

L'ordinamento di archivi è fondamentale per rispondere a interrogazioni con `ORDER BY`, e per eseguire operazioni relazionali come `JOIN`, `SELECT DISTINCT`, e `GROUP BY`.

L'algoritmo più comune utilizzato dai DBMS è lo **Z-way Sort-Merge**. L'algoritmo opera in due fasi, supponendo di dover ordinare un file di N_P pagine con a disposizione $N_B < N_P$ buffer in memoria centrale:

1. **Sort Interno**: Le pagine vengono lette e i record di ogni pagina vengono ordinati (es. Quicksort), creando delle "run" che vengono scritte su disco in un file temporaneo. Avendo N_B buffer, si possono ordinare N_B pagine alla volta.
2. **Merge** (**Fusione**): Le run vengono fuse in uno o più passi fino a ottenere un unico file ordinato.

Miglioramenti sostanziali si ottengono fondendo $N_B - 1$ run alla volta (riservando 1 buffer per l'output). Il costo complessivo delle operazioni di I/O (leggere e riscrivere le pagine) è dato da:

$$\text{Costo I/O} = 2 \times N_P \times \left(\lceil \log_{N_B - 1}(N_P / N_B) \rceil + 1 \right)$$

Un tempo di ordinamento può essere notevolmente ridotto aumentando il numero di buffer utilizzati per la fusione.

6.2 Realizzazione Operatori Relazionali

La realizzazione efficiente di un DBMS richiede che gli operatori relazionali logici (come Proiezione, Selezione, Raggruppamento e Join) siano tradotti in **operatori fisici** realizzati tramite algoritmi specifici. Un operatore logico può essere implementato con diversi algoritmi, ciascuno codificato in un appropriato operatore fisico.

6.2.1 Operatori Fisici e Interfaccia a Iteratore

Gli algoritmi per implementare gli operatori relazionali sono codificati in specifici **operatori fisici**. I DBMS definiscono questi operatori mediante un'interfaccia a **iteratore**. Un iteratore è un oggetto che fornisce i seguenti metodi principali:

- `open`: Inizializza lo stato dell'operatore, alloca i buffer per l'input e l'output e richiama ricorsivamente `open` sugli operatori figli. Viene anche utilizzato per passare argomenti, come la condizione che un operatore `Filter` deve applicare.
- `next`: Richiede la successiva tupla del risultato dell'operatore. L'implementazione di questo metodo include l'invocazione di `next` sugli operatori figli e l'esecuzione di codice specifico dell'operatore, e in genere ritorna un record.
- `isDone`: Indica se ci sono ancora valori da leggere (generalmente booleano).
- `close`: Termina l'esecuzione dell'operatore, rilasciando le risorse allocate.

Esempi di operatori fisici includono `TableScan(R)` per la scansione completa di una relazione R, e `IndexScan(R, Idx)` per la scansione basata su un indice Idx.

6.2.2 Realizzazione degli Operatori Fondamentali

Proiezione (Π) Se la proiezione include la clausola `DISTINCT` (eliminazione dei duplicati), un approccio comune è basato sull'ordinamento: si legge la relazione R, si scrive una relazione temporanea T contenente solo gli attributi della `SELECT`, si ordina T su tutti gli attributi, e infine si eliminano i duplicati. Gli operatori fisici per la proiezione logica $\pi_b\{A_i\}$ (con duplicati) e $\pi\{A_i\}$ (senza duplicati) sono:

- $Project(O, \{Ai\})$: Esegue la proiezione senza eliminazione dei duplicati.
- $Distinct(O)$: Elimina i duplicati dai record, assumendo che l'input O sia ordinato.

Selezione/Restrizione (σ) L'efficienza della selezione dipende dalla presenza o meno di indici.

- **Senza indice e dati disordinati**: Il costo è $N_{pag}(R)$, ovvero il numero di pagine della relazione R. L'operatore fisico corrispondente è `Filter (O,` ψ`)`.
- **Con indice (es. B+-albero)**: Il costo si riduce a $CI + CD$, dove CI è il costo di accesso all'indice e CD è il costo di accesso ai dati. L'operatore fisico corrispondente è `IndexFilter (R, Idx,` ψ`)`, che esegue la restrizione utilizzando l'indice Idx.

Raggruppamento e Aggregazione (Γ) Per le operazioni di aggregazione senza `GROUP BY` (es. `SELECT COUNT(*)`) si visitano i dati e si calcolano le funzioni di aggregazione. Nel caso in cui sia presente la clausola `GROUP BY`, si utilizza tipicamente un approccio basato sull'ordinamento: i dati vengono ordinati sugli attributi del `GROUP BY`, poi visitati per calcolare le funzioni di aggregazione per ogni gruppo. L'operatore fisico è $GroupBy(O, \{Ai\}, \{fi\})$, che raggruppa i record di O sugli attributi $\{A_i\}$ e utilizza le funzioni di aggregazione $\{f_i\}$ (presenti nella `SELECT` e nella `HAVING`). L'operatore restituisce record con gli attributi $\{A_i\}$ e le funzioni $\{f_i\}$, assumendo che i record di O siano ordinati sugli $\{A_i\}$.

6.2.3 Algoritmi di Giunzione (Join)

L'esecuzione di una giunzione (Join) è cruciale; l'approccio di calcolare il Prodotto Cartesiano ($R \times S$) seguito da una restrizione è inefficiente a causa della potenziale dimensione di $R \times S$. Sebbene il Join sia logicamente commutativo, dal punto di vista fisico esiste una chiara distinzione tra l'operando sinistro (esterno o "outer") e l'operando destro (interno o "inner"), che influenza le prestazioni.

Nested Loops Join (Cicli Nidificati) L'algoritmo di base `Nested Loops` consiste nel visitare l'intera relazione interna S per ogni record r della relazione esterna R:

$$\text{foreach record } r \text{ in } R \text{ do} \begin{cases} \text{foreach record } s \text{ in } S \text{ do} \\ \quad \text{if } r_i = s_j \text{ then aggiungi } \langle r, s \rangle \text{ al risultato} \end{cases}$$

Nel caso base (1 buffer per R e 1 per S), il costo totale è:

$$\text{Costo I/O} = N_{pag}(R) + N_{rec}(R) \times N_{pag}(S)$$

Conviene scegliere come **relazione esterna** quella con un minor numero di tuple o, in generale, quella che soddisfa la condizione $N_{rec}(R) \times N_{pag}(S) < N_{rec}(S) \times N_{pag}(R)$, che corrisponde alla relazione con i record più grandi/lunghi. L'ordine delle tuple nel risultato finale coincide con l'ordine eventualmente presente nella relazione esterna R.

Nested Loop A Pagine (`PageNestedLoop`) Molti DBMS utilizzano la variante `Nested Loop a Pagine` (Page Nested Loop), che è più efficiente perché rinuncia a preservare l'ordine della relazione esterna ed esegue il join di tutte le tuple in memoria prima di richiedere nuove pagine della relazione interna S. Logica: Per ogni pagina p_R di R, si visitano tutte le pagine p_S di S ed si esegue il join di tutte le tuple in p_R e p_S. La strategia si estende anche assegnando più buffer a R.

Nested Loop con Indice (`IndexNestedLoop`) Questa variante sostituisce la scansione completa della relazione interna S con una scansione basata su un indice I_{S_j} costruito sugli attributi di join di S. Questo accesso mediante indice riduce notevolmente i costi di esecuzione del Nested Loops Join.

$$\text{foreach record } r \text{ in } R \text{ do} \begin{cases} \text{foreach record } s \text{ in } \texttt{get-through-index}(I_{S_j} = r_i) \text{ do} \\ \quad \text{aggiungi } \langle r, s \rangle \text{ al risultato} \end{cases}$$

L'operatore fisico è `IndexNestedLoop (OE, OI,` ψ_J`)`. L'operando interno OI è spesso un `IndexFilter` sulla relazione interna.

Sort-Merge Join (`SortMerge`) Questo algoritmo è applicabile quando entrambe le relazioni in input (R e S) sono **ordinate sugli attributi di join**. L'ordinamento può essere raggiunto se la relazione è fisicamente ordinata o se esiste un indice sugli attributi di join. L'algoritmo sfrutta l'ordinamento per evitare confronti inutili. Il numero di letture (senza contare il tempo di sort) è dell'ordine di $N_{pag}(R) + N_{pag}(S)$ se si accede sequenzialmente. L'operatore fisico è `SortMerge (OE, OI,` ψ_J`)`.

Implementazione Join	Requisiti Chiave	Costo I/O (Caso Base)	Vantaggio Principale	Operatore Fisico
Nested Loop (Standard)	Nessuno	$N_{pag}(R) + N_{rec}(R) \times N_{pag}(S)$	Simplicità.	`NestedLoop` 2
Nested Loop a Pagine	Nessuno	(Più efficiente di Standard)	Esegue join a livello di pagina in memoria.	`PageNestedLoop` 2
Index Nested Loop	Indice su relazione interna (S)	Riduzione significativa del costo I/O.	Velocizza l'accesso all'operando interno.	`IndexNestedLoop` 2
Sort-Merge	Entrambe le relazioni ordinate su attributi di join.	$\approx N_{pag}(R) + N_{pag}(S)$	Molto efficiente per relazioni già ordinate.	`SortMerge` 3

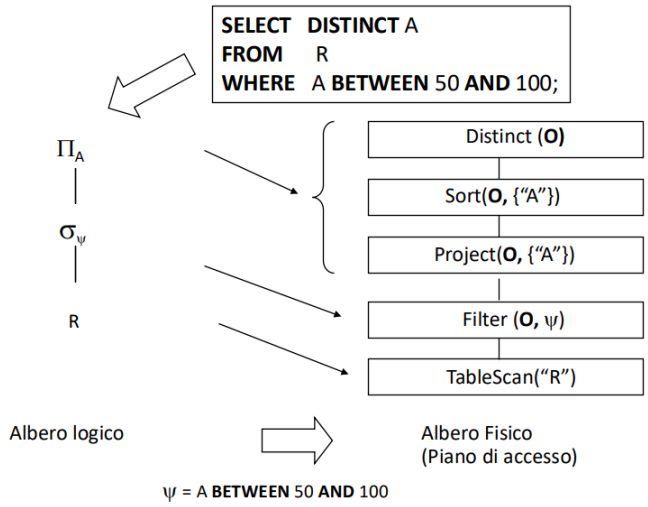
6.3 Piani d'Accesso

I piani d'accesso sono **espressioni algebriche** in cui gli **operatori logici** vengono **sostituiti** da **operatori fisici**.

In questa fase si applicano quindi le ottimizzazioni sulle interrogazioni, si sceglie quindi il piano di costo minimo tra quelli selezionabili.

Fasi del Processo di Ottimizzazione Le fasi attraversate sono le seguenti:

- **Analisi e Semplificazione**: Verifica la correttezza del comando, normalizzazione e semplificazione della condizione.
- **Trasformazione**: Trasformazione dell'albero con regole di equivalenza.
- **Ottimizzazione Fisica**: Definizione del piano di accesso, quindi scelta dell'algoritmo per eseguiro ogni operatore, l'ideale sarebbe trovare i migliori piani d'accesso ma l'obiettivo reale e quello di evitare i peggiori.
- **Esecuzione**: Viene eseguito il piano d'accesso.



6.4 Gestione Transazioni

Una transazione è un programma sequenziale costituito da operazioni che il sistema deve eseguire garantendo atomicità, serializzabilità e persistenza.

Il DBMS deve quindi **gestire** i **dati in memoria permanente**, il **buffer** a disposizione per il trasferimento da memoria centrale a memoria di massa e delle **ottimizzazioni** delle **interrogazioni**.

Proprietà ACID Elenchiamo le proprietà ACID delle transazioni:

- **Atomicity**: Le transazioni devono essere eseguite con la regola del tutto o niente.
- **Consistency**: La transazione deve lasciare il DB in uno stato constente, non violando eventuali vincoli di integrità
- **Isolation**: L'esecuzione di una transazione deve essere indipendente dalle altre.
- **Durabilty**: L'effetto di una transazione conclusa con un `commit` non deve essere perso.

Transazione per il DBMS Anche se una transazione può eseguire molte operazioni, al DBMS interessano solo quelle di lettura/scrittura che definiremo così:

- $r_i[x]$: Operazione che comporta la **lettura di una pagina nel buffer**, se non già presente.
- $w_i[x]$: Operazione che comporta un **eventuale lettura nel buffer** di una pagina e la sua **modifica nel buffer**, ma **non necessariamente** la sua **scrittura** in **memoria permanente**.

Tipi di Malfunzionamento Esistono tre tipi di malfunzionamento:

- **Fallimenti di Transazioni**: Non comportano la perdita di dati in memoria temporanea ne persistente.
- **Fallimenti di Sistema**: Comportano perdite di dati in memoria temporanea ma non in memoria persistente.
- **Disastri**: Comportano perdite di dati in memoria permanente.

6.5 Gestione Affidabilità

Verifica che siano garantite le **proprietà** di **atomicità** e **persistenza** delle transazioni. Quindi questa fase è responsabile dei comandi `begin transaction`, `commit` e `rollback`.

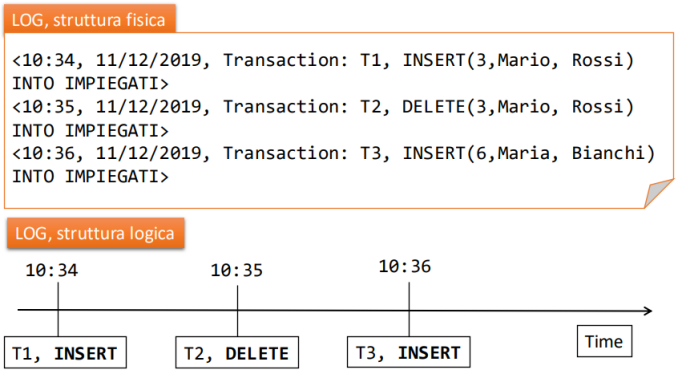
6.5.1 Ripresa a Caldo/Freddo

Esistono due tipi di ripristino del sistema:

- **Ripresa a Caldo**: Ripristino del sistema dopo un malfunzionamento software.
- **Ripresa a Freddo**: Ripristino del sistema dopo un malfunzionamento hardware.

6.5.2 File di LOG e Primitive Undo/Redo

Tutte le operazioni svolte dal DBMS vengono riportate in un file di LOG:



Ogni transazione T vengono indicate le operazioni di `begin`, `commit` e `abort` di T con $B(T), C(T), A(T)$.

Indichiamo invece con $D(T, O, BS)$, $I(T, O, AS)$ e $U(T, O, BS, AS)$ rispettivamente le operazioni di `delete`, `insert` e `update`.

Primitive Undo/Redo I record dei log associati ad una transazione consentono di **disfare (undo)** o **rifare (redo)** le corrispondenti azioni su una base di dati. Un log contiene **due tipi diversi** di **record**:

- **Record di Transazione**: Tengono traccia di ogni tipo operazione relativa ad una transazione, quindi per ciascuna di esse si mantengono record di `begin`, `insert`, `delete`, `update`, `commit` e `abort`.
- **Record di Sistema**: Tengono traccia delle operazioni di sistema, quindi di elementi quali `dump` che permette la copia completa in backup della base di dati e `checkpoint` per contrassegnare periodicamente il file di LOG.

Regole di Scrittura dei LOG Esistono diversi approcci:

- **Regola Write Ahead Log (WAL)**: La parte **BS (Before State)** di ciascun record di log deve essere **scritta prima** che la corrispondente **operazione venga effettuata** nella base di dati.
- **Regola Commit Precedence**: La parte **AS (After State)** di ogni record di log deve essere scritta nel log **prima** di **effettuare il commit della transazione**.

Undo/Redo Mostriamo quali politiche si seguono per disfare/rifare specifiche transazioni sulla base di dati:

- **Undo**: Si segue la **politica della modifica libera**, ossia le modifiche possono essere portate nella base di dati stabile prima che la transizione termini. La **regola** per poter **disfare** invece è quelle del **Write Ahead Log**.

- **Redo**: Secondo il commit libero una transazione può essere considerata terminata normalmente prima che utte le modifiche bengano riportate nella base di dati stabile. La regola per poter rifare una transazione è quindi quella della commit rule, riportando le modifiche di una transizione nel log prima che questa raggiunga il commit.

Checkpoint Si scrive sul LOG un mark che riporta l'elenco delle transazioni attive a tempo di checkpoint $BeginCheckpoint, \{T_1, \cdots, T_n\}$. In parallelo alle normali operazioni delle transazioni, il gestore del buffer riporta sul disco tutte le pagine modificate. Successivamente si scrive sul giornale *EndCheckpoint*, in questo modo si certifica che tutte le scritture avvenute prima del *BeginCheckpoint* siano state riportate su disco.

Attenzione, non abbiamo alcuna garanzia di aver riportato su disco le scritture avvenute tra *BeginCheckpoint* ed *EndCheckpoint*.

Ripresa di Malfunzionamenti in base alla Tipologia Applichiamo **undo/redo** in base al tipo di malfunzionamento.

- **Fallimenti di Transazioni**: Si scrive nel LOG $(T, abort)$ e si applica il `redo`.
- **Fallimenti di Sistema**: La base di dati viene ripristinata con il comando restart, **partendo** quindi dallo **stato** dell'**ultimo checkpoint**.
  - Le transazioni **non terminate** devono essere approcciate in `undo`.
  - Le transazioni **terminate** devono essere approcciate in `redo`.
- **Disastri**: Si riporta in linea la copia più recente della base di dati dal **dump di backup**, secondo il meccanismo della **ripartenza a freddo**.

6.5.3 Procedure della Ripresa a Caldo

Si seguono questi passi:

- Trovare l'**ultimo checkpoint** percorrendo i **LOG a ritroso**.
- Costruire gli insiemi `undo` e `redo`.
- Ripercorrere **nuovamente a ritroso** i LOG, fino all'**azione più vecchia** nel insieme `undo` costruito prima.
- Percorrere il LOG **in avanti**, eseguendo tutte le azioni contenute nell'insieme `redo`.

6.6 Gestione della Concorrenza

Il DBMS deve garantire sia **esecuzione concorrente** di transazioni sia **accesso concorrente** agli stessi **dati**.

Seriale e Serializzabile Definiamo la differenza tra queste due proprietà:

- **Seriale**: Un esecuzione di un insieme di transazioni $\{T_1, \cdots, T_n\}$ si dice seriale se, per ogni coppia di transazioni T_i, T_j tutte le operazioni di T_i vengono eseguite prima di quelle di T_j.
  - Uno schedule $S = \{T_1, T_2, \cdots, T_n\}$ si dice **seriale** se appaiono consecutivamente, senza essere **inframezzate** da **azioni di altre transazioni**.
- **Serializzabile**: Un insieme di transizioni si dice serializzabile se produce lo stesso effetto sulla base di dati di quello ottenibile eseguendo in serie, in qualche ordine, le sole transizioni normalmente.

6.6.1 Problematiche Gestione di Transazioni

In un sistema reale le transazioni vengono eseguite in concorrenza, questo porta alla gestione di un insieme di problematiche note:

Mostriamo degli esempi, categorizzando queste casistiche di race condition:

- **Perdita d'Aggiornamento**:

X=3

Transazione1 (T1)	Transazione2 (T2)
Read(x)	
x=x+1	
	Read(x)
	x=x+1
	Write(x)
	Commit work
Write(x) Commit work	

T2 scrive 4

T1 scrive 4

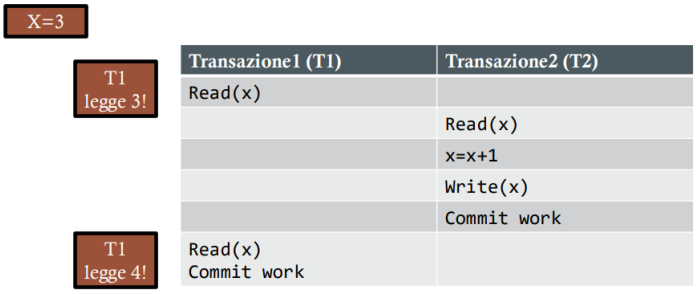
- **Lettura Sporca o Impropria**:

X=3

Transazione1 (T1)	Transazione2 (T2)
Read(x)	
x=x+1	
Write(x)	
	Read(x)
	Commit work
Rollback work	

T2 legge 4!

- **Letture Inconsistenti o Non Riproducibili**:



Transazione1 (T1)	Transazione2 (T2)
Read(x)	
	Read(x)
	x=x+1
	Write(x)
	Commit work
Read(x) Commit work	

Controllo Concorrenza I DBMS implementano **due classi** di **tecniche di controllo** di conocrrenza che grantiscono direttamente la **serializzabilità delle transazioni concorrenti**:

- **Protocolli Pessimistici**: Tendono a ritardare l'esecuzione di transazioni che potrebbero generare conflitti, cercando di prevenire.
  - Basato su fasi di lettura, validazione e scrittura.
- **Protocolli Ottimistici**: Permettono l'esecuzione sovrapposta e non sincronizzata di transazioni ed effettuano controlli sui conflitti solo a fine commit.
  - Metodi basati su lock e timestamps.

6.7 Gestione delle Transazioni

Meccanismo dei Lock Il meccanismo che blocca l'accesso ai dati ai quali una transazione accede ad altre transazioni:

- Lock multi granularità.
- Lock in operazioni di scrittura/lettura.

Quando una risorsa è bloccata, le transazioni che ne richiedono l'accesso vengono messe in coda. Si definiscono due operazioni sui lock, ossia **lock/unlock**.

Serializzatore 2PL Stretto Il serializzatore (**gestore della concorrenza**) ha il compito di **stabilire l'ordine** secondo il quale vanno eseguite le **singole operazioni** per rendere **serializzabile l'esecuzione** di un insieme di **transazioni**.

Strict Two Phase Locking Protocollo che segue queste regole:

- Ogni transazione, prima di effettuare un operazione, acquisisce il blocco corrispondente **chiedendo il lock**.
- Transazioni diverse non ottengono lock in conflitto.
- I lock si rilasciano alla terminazione della transazione, precisamente a tempo di commit.

La gestione a lock può però causare delle attese circolari infinite, dette **deadlocks**.

Risoluzione di Deadlocks Possiamo risolvere il problema dei deadlocks in diversi modi:

- **Timeout**: Ogni operazione di una transazione ha un timeout, entro il quale deve essere completata, altrimenti va in abort.
- **Deadlock Avoidance**: Basata su **acquisizione di tutte le risorse necessarie** oppure **classi di priorità** tra transazioni.
- **Deadlock Detection**: Utilizzare algoritmi e/o grafi per identificare la presenza di cicli nella richiesta di risorse.

Timestamp delle Transazioni Metodo che associa un **timestamp** che rappresenta il momento di inizio della transazione, e si seguono due criteri:

- Ogni transizione **non può leggere o scrivere** un **dato scritto** da una **transazione** con **timestamp maggiore**.
- Ogni transizione **non può scrivere** su un dato **già letto** da una **transazione** con **timestamp maggiore**.

Livelli di Isolamento e Consistenza Esistono diversi livelli di garanzia di proprietà:

- **Serializable**:
  - La transazione leggerà solo cambiamenti fatti da transazioni concluse.
  - Nessun valore letto o scritto da T verrà cambiato da altre transazioni fino alla sua conclusione.
  - Se T legge un insieme di valori acceduti in ricerca, allora l'insieme non viene modificato da altre transazioni fino alla sua conclusione.
- **Repeatable Read**:
  - La transazione legge solo cambiamenti fatti da transazioni concluse.
  - Nessun valore letto o scritto dalla transazione verrà camiato da altre transazioni fino alla sua conclusione.
- **Read Committed**:
  - La transazione legge solo cambiamenti fatti da transazioni concluse.
  - La transazione non vede nessun cambiamento effettuato da transazioni concorrenti non concluse.
- **Read Uncommitted**:
  - La transazione è soggetta ad effetti fantasma, dato che può leggere modifiche fatte ad un oggetto da un altra transazione in esecuzione.