BASE DI DATI

Corso A

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

Ultima Compilazione - November 5, 2025

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.

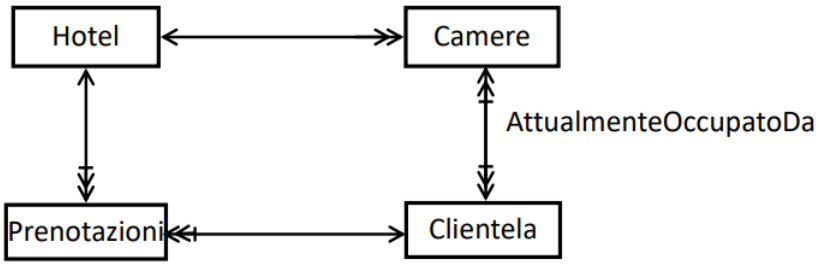
2.2 Progettazione Concettuale

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.

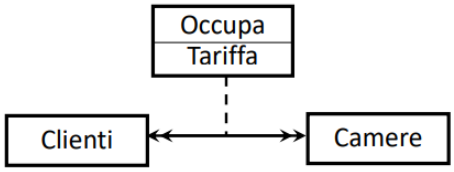
3. **Trattino**: Nessun limite inferiore, di conseguenza per ogni elemento della classe di partenza può anche non essere presente alcun elemento nella classe d'arrivo.

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

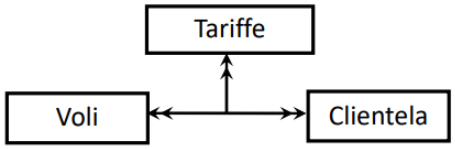
A volte le associazioni potrebbero complicarsi perchè:

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:

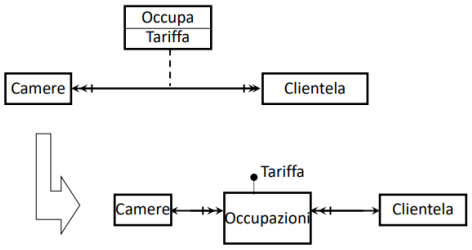


Figure 1: Reificazione di Associazione con Attributo

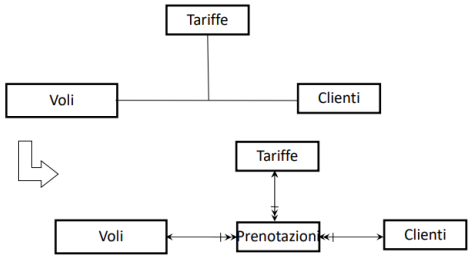


Figure 2: Reificazione di Associazione Ternaria

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.

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

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

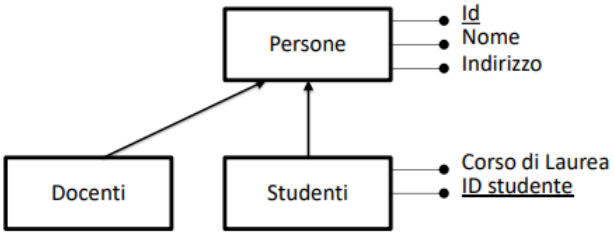
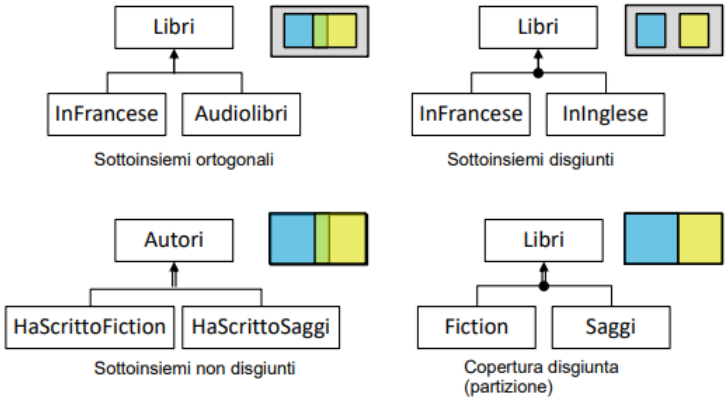


Figure 3: Notazione grafica di riferimento

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**.

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

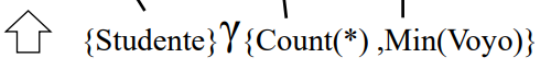
2. **Giunzione (Join)**: L'operazione $R \bowtie_{R.A=S.B} S$ combina le righe (tuple) di R ed S **solo quando** un attibuto A di R **è uguale** ad un attributo B di S. Questa operazione è derivata da:

$$R \bowtie_{R.A=S.B} S = \rho_{R.A=S.B}(R \times S)$$

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



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.