

Emiddio Ingenito - Basi di dati

[Sistemi informativi, informazioni e dati](#)

[DBMS](#)

[Problemi della Memorizzazione su File](#)

[Schema vs istanza](#)

[Astrazione dei Dati in un DBMS](#)

[Livelli di Astrazione del Modello ANSI-SPARC](#)

[Lecture notes](#)

[DB Users](#)

[Modello relazionale](#)

[Lecture notes](#)

[Informazione incompleta](#)

[Vincoli di integrità](#)

[Vincoli di tupla e dominio](#)

[Vincolo di chiave](#)

[Scelta della chiave](#)

[Lesson notes](#)

[Vincolo di integrità referenziale \(o di chiave esterna\)](#)

[Vincoli in PostgreSQL](#)

[Algebra relazionale](#)

[Operatori unari](#)

[Operatori binari](#)

[Equi-Join](#)

[Natural Join](#)

[Operatore divisione](#)

[Viste](#)

[CTE\(Common Table Expressions\)](#)

[Subquery \(o query nidificate/innestate\):](#)

[IN:](#)

[ANY/SOME:](#)

[ALL:](#)

[Outer Join \(Su relazioni 1 a N\)](#)

[Left Outer Join:](#)

[Right join:](#)

[Full Join:](#)

[Funzioni di aggregazione:](#)

[La funzione COUNT\(\):](#)

[Esempio:](#)

[La funzione MIN\(\) e MAX\(\):](#)

[Esempio:](#)

[La funzione AVG\(\):](#)

[Esempio:](#)

[La funzione SUM\(\):](#)

[Esempio:](#)

[La funzione STDDEV\(\):](#)

[GROUP BY:](#)

[HAVING \(Utilizzo funzioni di aggregazione come condizione di selezione\):](#)

[Rappresentazione di gerarchie](#)

[Trigger e asserzioni](#)

[Progettazione di una base di dati](#)

[Progettazione concettuale - Modello ER](#)

[Entità](#)

[Relazioni](#)

[Attributi](#)

[Cardinalità delle relazioni](#)

Cardinalità degli attributi
Identifieri delle entità
Entità debole
Generalizzazioni (Relazioni is-a)
Progettazione logica
 Traduzioni di gerarchie (Relazioni is-a)
 Traduzioni di relazioni
Reverse Engineering
Normalizzazione
 Anomalia di aggiornamento
 Anomalia di cancellazione
 Anomalia di inserimento
 Dipendenze funzionali
 Forma normale di Boyce e Codd (BCNF)
 Proprietà di una buona scomposizione
 Terza forma normale
 Seconda forma normale
 Prima forma normale
Progettazione fisica
 Fattore di blocco
 Strutture primarie
 Indici
 Indici primari
 Indici secondari
 Esercizi in classe:
Sicurezza
 Controllo accesso
 Politiche per l'amministrazione della sicurezza
 Politiche per il controllo dell'accesso
 Politiche Discrezionali (DAC - Discretionary Access Control)
 Meccanismo di Controllo
 Vantaggi
 Svantaggi
 Politiche Mandatorie (MAC - Mandatory Access Control)
 Vantaggi
 Svantaggi
 System R
 Comando GRANT/REVOKE
 SYSAUTH e SYSCOLAUTH
 SYSAUTH
 SYSCOLAUTH
 Revoca ricorsiva
Transazioni
 Problemi di concorrenza
 Proprietà delle transazioni
Sistemi NoSQL
 Caratteristiche principali dei DB NoSQL
 Tipologie di database NoSQL
 Document-Based Databases
Esercizi Algebra Relazionale (π , σ , \bowtie , ρ)
 Schema "Sentenze":
 Schema "Missioni":
 Schema "Supermercato":
 Schema "Arredamento":
 Schema "Gommista":
 Schema "Calendari":
 Schema "Biscotti":
 Schema "Rimborso":

[Schema "Conferenze"](#):
[Schema "Donut"](#):
[Schema "Ambulatorio"](#):
[Schema "Orologi"](#):
Esercizi SQL
[Schema "Pittori](#)
[Schema "Feste"](#)
[Schema "Autobus"](#):
[Schema "Supermercati"](#):
[Schema "Ambulatorio"](#)
[Schema "Biscotti"](#)
[Schema "Conferenze"](#)
[Schema "Donut"](#)
[Schema "Missioni"](#)
Esercitazioni in classe
Esercizi normalizzazione
[Temi d'esame](#)

Sistemi informativi, informazioni e dati

Nello svolgimento di ogni attività individuale o collettiva, è essenziale la disponibilità di informazioni.

Si definisce sistema informativo, l'insieme di risorse umane e materiali, e dei processi utili alla raccolta, archiviazione, elaborazione e scambio delle informazioni necessarie per le attività operative.

La parte del sistema informativo automatizzata mediante tecnologie informatiche (Nelle loro componenti HW e SW), è detta sistema informatico.

Nelle attività umane, le informazioni vengono rappresentate e scambiate mediante tecniche semplici, scritte su carta, tramite disegni, testo, figure, o addirittura scambiate a voce, ricordandole a memoria.

Nei sistemi informatici, le informazioni vengono rappresentate per mezzo di dati, i quali a seguito di una corretta interpretazione, codificano delle informazioni.

Dato = Forma grezza della conoscenza, Stringhe di testo, numeri.

Informazioni = Dati che assumono significato a seguito di interpretazione specifica.

Data comprises raw, unprocessed facts that need context to become useful, while information is data that has been processed, organized, and interpreted to add meaning and value

In modo astratto, possiamo quindi dire che i dati da soli non hanno significato, ma lo assumono a seguito di una corretta interpretazione (diventando, informazione).

Esempio: La stringa "Ferrari" e il numero "8" scritti su un foglio di carta. Senza un contesto, questi due dati non significano nulla. Si parla di un uomo che ha 8 Ferrari? Un uomo di cognome "Ferrari" ha 8 figli?

Se fornisco un contesto, di un ristorante per esempio, allora posso intuire allora che il tavolo 8 ha ordinato una bottiglia di spumante della marca "Ferrari".

DBMS

Un DBMS (Data Base Management System), è un sistema software in grado di gestire collezioni di dati che siano grandi (in termini di dimensioni), condivise (nel senso che più applicazioni ed utenti devono poter accedere ad essi, contemporaneamente e con diverse modalità) e persistenti (Devono rimanere memorizzate per quantità di tempo indefinite,

fino ad esplicita eliminazione), garantendo affidabilità (cioè la capacità di mantenere intatti i dati anche in caso di malfunzionamenti HW e SW, fornendo appositi meccanismi di backup e recovery), privatezza, efficienza.

Una **base di dati** (database) è una collezione strutturata di dati, organizzata secondo un modello logico ben definito e gestita da un **Database Management System (DBMS)**.

L'utilizzo di un database consente di superare le limitazioni della semplice memorizzazione dei dati tramite file tradizionali.

Problemi della Memorizzazione su File

1. Ridondanza dei dati

Nei file tradizionali, uno stesso dato può essere memorizzato in più copie all'interno di diversi file o persino nello stesso file.

La ridondanza va evitata perché può generare problemi come:

2. Incongruenza dei dati

Se un dato ridondato viene aggiornato solo in alcune delle sue occorrenze, le copie rimaste inalterate conterranno valori discordanti.

Si crea la necessità di stabilire quale versione del dato sia quella corretta..

3. Inconsistenza dei dati

L'inconsistenza si verifica quando istanze distinte che dovrebbero rappresentare la medesima entità presentano valori differenti per alcuni attributi.

Il sistema non è più in grado di determinare il valore corretto per i dati.

Esempio: Un file contenente i prodotti in vendita in un negozio potrebbe avere due righe con lo stesso codice a barre ma prezzi diversi. In questo caso, quale valore è corretto?

Tra i compiti del DBMS, c'è anche fornire linguaggi per basi di dati, con cui interagire con essa, in particolare essi si distinguono in diverse categorie:

I linguaggio SQL, di riferimento per il corso, integra le funzionalità di tutte le categorie.

- DDL (Data definition language), responsabile della definizione della struttura dei dati in un database. In SQL, questo corrisponde alla manipolazione delle tabelle attraverso i comandi `CREATE TABLE`, `ALTER TABLE`, e `DROP TABLE`
- DML (Data manipulation language), fornisce i comandi per inserire, modificare, eliminare i dati all'interno di un database. In SQL, corrisponde ai comandi `INSERT`, `UPDATE`, e `DELETE`
- DCL (Data control language), fornisce i comandi per l'assegnamento e la revoca dei permessi agli utenti che interagiscono con le basi di dati.
In SQL, questo corrisponde ai comandi `GRANT`, `REVOKE`, e `DENY`
- DQL (Data query language), permette di interrogare la base di dati, estraendone le informazioni contenute.
In SQL, questo corrisponde al comando `SELECT`

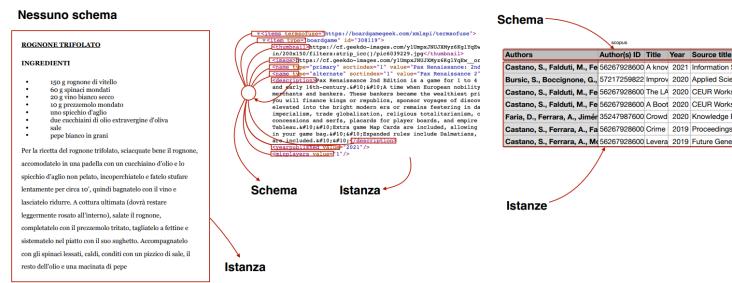
Schema vs istanza

Lo schema di una base di dati è "lo stampino" che definisce come i dati devono essere memorizzati.

Generalmente, una volta fissata, rimane invariata nel tempo.

Usiamo la seguente notazione per indicare lo schema di una relazione: Docenza(Corso, NomeDocente)

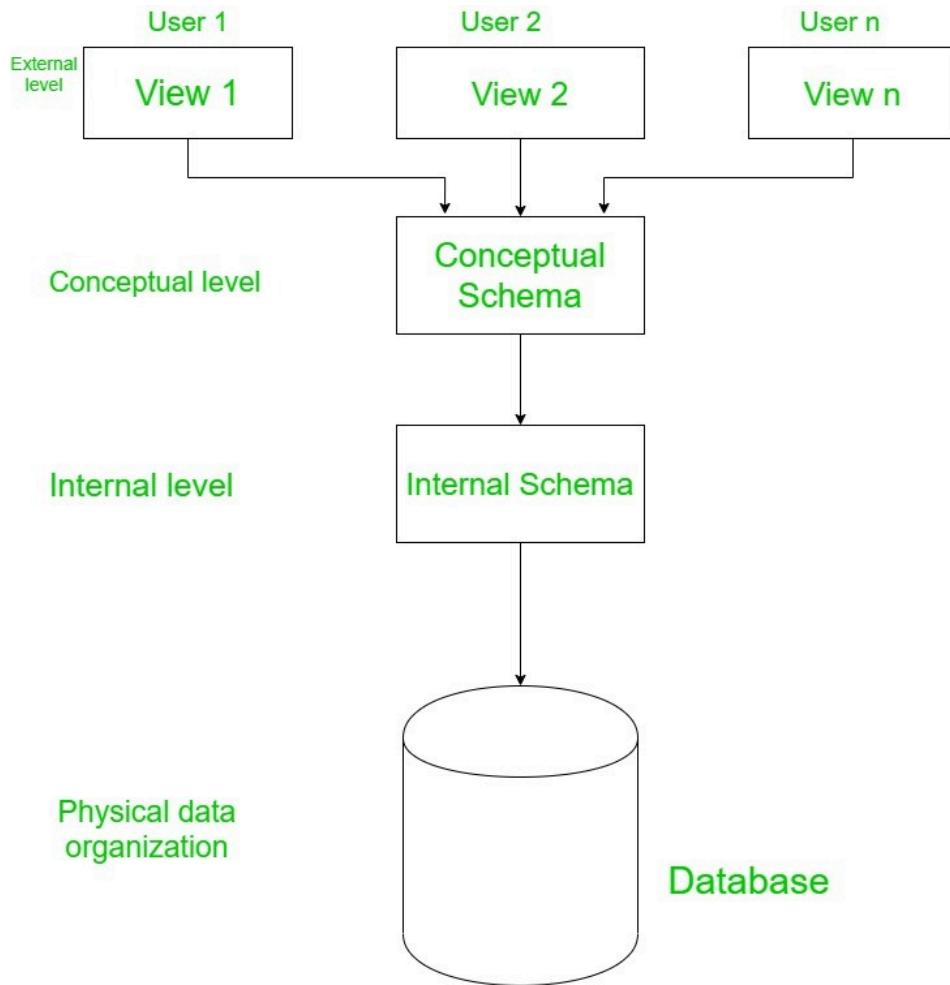
L'istanza di una relazione (tabella) invece, è l'insieme delle sue informazioni (righe della tabella)



Astrazione dei Dati in un DBMS

L'organizzazione e la gestione dei dati all'interno di un **Database Management System (DBMS)** seguono un modello di astrazione standardizzato noto come **modello ANSI-SPARC**. Questo modello suddivide la gestione dei dati in tre livelli distinti: **schema esterno**, **schema logico** e **schema interno**.

Livelli di Astrazione del Modello ANSI-SPARC



The ANSI-SPARC three-level architecture

1. Schema esterno (Livello esterno o di vista)

- Rappresenta la **porzione della base di dati visibile agli utenti o alle applicazioni**, attraverso viste personalizzate e restrittive.
- Ogni applicazione o gruppo di utenti può accedere solo ai dati di cui ha bisogno, senza dover interagire con l'intero database.
- Questo livello garantisce la sicurezza e il controllo degli accessi, evitando esposizione non necessaria delle informazioni.

2. Schema logico (Livello concettuale)

- Describe **l'organizzazione logica della base di dati**, indipendentemente dalla sua memorizzazione fisica.
- Definisce la struttura dei dati e le relazioni tra essi, secondo un determinato **modello logico di rappresentazione**.
- I principali modelli logici utilizzati nei database sono:
 - Modello relazionale** – Organizza i dati in tabelle (relazioni) costituite da record (tuple) con una struttura fissa.

- **Modello gerarchico** – Organizza i dati in una struttura ad **albero**, in cui ogni nodo padre può avere più figli, ma ogni nodo figlio ha un solo padre.
- **Modello reticolare** – Simile al modello gerarchico, ma con una struttura a **grafo**, in cui un nodo figlio può avere più padri.
- **Modello a oggetti** – Estende i concetti della programmazione a oggetti ai database, permettendo di rappresentare le informazioni mediante **oggetti, classi e metodi**.
- **Modello XML** – Utilizza il linguaggio XML per rappresentare dati e struttura nello stesso formato, consentendo una maggiore flessibilità rispetto al modello relazionale.
- **Modelli NoSQL** – Superano alcune limitazioni del modello relazionale, supportando dati non strutturati o semi-strutturati (es. documenti JSON, colonne dinamiche, grafi, ecc.).

3. Schema interno (Livello fisico)

- Definisce il modo in cui i dati vengono **fisicamente memorizzati e organizzati** nei dispositivi di archiviazione (dischi, SSD, ecc.).
- Comprende la gestione di strutture come **file, indici, blocchi di memoria** e tecniche di ottimizzazione per l'accesso efficiente ai dati.
- La sua implementazione è gestita direttamente dal **DBMS**, ed è trasparente per gli utenti e le applicazioni.

L'architettura a livelli così definita, garantisce l'indipendenza dei dati, in particolare:

- Indipendenza fisica dei dati

L'indipendenza fisica si riferisce alla capacità di modificare l'organizzazione fisica dei dati (come la modalità di memorizzazione su disco, l'uso di file indice o tecniche di partizionamento) senza alterare la struttura logica della base di dati e senza impattare le applicazioni che la utilizzano.

Esempio:

Supponiamo che un'azienda memorizzi i dati dei clienti in un file sequenziale su disco. Per migliorare le prestazioni, gli amministratori decidono di utilizzare un indice B-tree per velocizzare le ricerche. Grazie all'indipendenza fisica, questa modifica può essere effettuata senza dover riscrivere le query SQL o modificare la struttura logica delle tabelle. I programmi che accedono ai dati continueranno a funzionare senza alcun cambiamento.

- Indipendenza logica dei dati

L'indipendenza logica riguarda la possibilità di modificare lo schema logico della base di dati senza alterare i programmi applicativi che accedono ai dati. Questo significa che è possibile aggiungere, rimuovere o modificare attributi e relazioni tra tabelle senza impattare il codice delle applicazioni, a meno che queste non facciano riferimento esplicito ai campi modificati.

Esempio:

Un database contiene una tabella Dipendenti(Nome, Cognome, DataNascita, Ruolo). Successivamente, l'azienda decide di aggiungere un nuovo attributo Email. Grazie all'indipendenza logica, le applicazioni che non utilizzano questo nuovo campo continueranno a funzionare senza problemi, poiché la struttura preesistente rimane invariata per loro.

Lecture notes

```
-- nei requisiti della base di dati si sappia che una pellicola può essere associata a più generi cinematografici
-- possibile rappresentazione in tabelle:
```

```
movie(id, title, year, length)
```

```
-- relazione molti-a-molti
```

```
movie_genre(idmovie, idgenre)
```

```
idmovie | idgenre
```

```
=====
i345o23    0001
w485923    0001
9hkqse8     0002
9hkqse8     0003
9hkqse8     0004
```

```
genre(id, name)
id | name
=====
0001 thriller
0002 Drama
0003 Sport
0004 Short
.....
```

-- consideriamo un caso alternativo: si sappia che un film è descritto da uno e un solo genere
-- possibile rappresentazione:

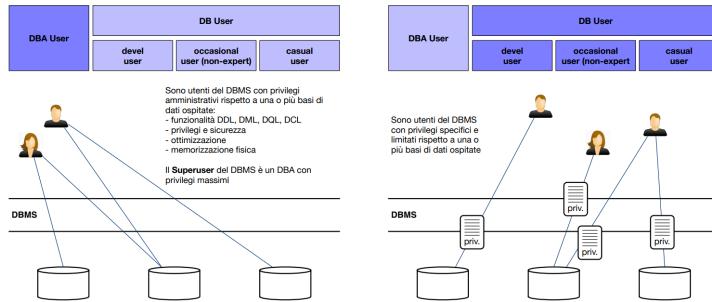
```
movie(id, title, year, length, genre)
genre(id, name)
```

-- le pellicole hanno recensioni. Una recensione fa riferimento a uno specifico film, viene fatta da un recensore (un critico, una testata giornalista). la recensione ha una data e un valore su una scala fissata

-- relazione uno-a-molti
rating(reviewer, idmovie, score, min_rate, max_rate, review_date)
serve un id?
l'uso di id è sempre possibile per dotare la relazione di un identificatore artificiale, tuttavia è sempre importante trovare l'attributo o la combinazione di attributi naturali che identificano i record di una relazione.

in questo caso, la coppia (reviewer, idmovie) è un possibile identificatore della relazione.

DB Users



Nel contesto dei DB, distinguiamo 3 principali tipi di utenti che interagiscono con essi:

- Normal Users (DB User) - Interagiscono solo con determinate basi di dati, e possono eseguire solo alcune specifiche operazioni, in base ai privilegi ad essi concessi.
- Administrator (DBA) - Utenti con privilegi amministrativi: Possono eseguire qualsiasi comando dei linguaggi DML, DDL, DCL, DQL, sui database di cui sono amministratori.
Generalmente, ogni utente è di default amministratore delle basi di dati che essi creano.
- Superuser - Un DBA con privilegi massimi, su ogni database.

Modello relazionale

Il modello relazionale si basa su due concetti principali:

- Relazione, nel suo stretto significato matematico
- Tabella

In matematica, dati due insiemi D_1 e D_2 , indichiamo con la notazione $D_1 \times D_2$ il loro prodotto cartesiano, cioè l'insieme di coppie in cui il primo elemento appartiene a D_1 , e il secondo a D_2

Il numero n delle componenti del prodotto cartesiano (quindi il numero di elementi di ogni n-upla), viene detto grado della relazione prodotto cartesiano.

Il numero di elementi della relazione, viene chiamato cardinalità della relazione.

Per esempio, consideriamo i domini

$$D_1 = \{cane, gatto\}$$

$$D_2 = \{bianco, nero, marrone\}$$

Definiamo la tabella *animale(tipo, colore)* come la relazione matematica $R(D_1, D_2) \in D_1 \times D_2$, in cui il grado è 2, la cardinalità è 6 ($|D_1| \times |D_2| = 2 \times 3 = 6$, la tabella animale può contenere al massimo 6 valori univoci)

In essa, ogni n-upla della relazione contiene dati collegati tra loro.

Esempio:

Se consideriamo la seguente relazione

Juventus	Lazio	3	1
Lazio	Milan	2	0
Juventus	Roma	1	2
Roma	Milan	0	1

Ogni n-upla (riga della tabella), contiene dati in relazione tra loro, per esempio la prima riga, indica che c'è stata una partita "Juventus" vs "Lazio", terminata coi risultati 3 a 1.

Ricordiamo poi, che la relazione è un insieme, pertanto:

- L'ordine tra le n-uple non è rilevante (due tabelle con le stesse righe, ma in ordine diverso, rappresentano la stessa relazione)
- Inoltre, le righe all'interno di una tabella, sono identificate in base al loro contenuto, non in base alla loro posizione.
- I suoi elementi sono distinti (non possono essere presenti righe uguali)

L'unico ordine rilevante, è quello interno alla n-upla (ordine degli attributi).

Ogni dominio all'interno della relazione viene associato ad un nome detto **attributo**, che descrive il suo ruolo. Quindi, se abbiamo una relazione $R(D_1, D_2)$, possiamo riscriverla come $R(A_1, A_2)$, dove A_1 e A_2 sono i **nomi degli attributi**.

In pratica, questi nomi degli attributi sono usati per etichettare le colonne della tabella.

SquadraDiCasa	SquadraOspitata	RetiCasa	RetiOspitata
Juventus	Lazio	3	1
Lazio	Milan	2	0
Juventus	Roma	1	2
Roma	Milan	0	1

Ogni elemento di una relazione $R(D_1, D_2)$, è una tupla scritta con la notazione $t(d_1, d_2)$, dove $d_1 \in D_1$, and $d_2 \in D_2$.

Una volta che abbiamo una tupla $t(d_1, d_2)$, possiamo usare una notazione come $t[A_k]$ per riferirci al valore di un determinato attributo all'interno di quella tupla. Ad esempio, $t[A_2] = d_2$, significa che il valore dell'attributo A_2 nella tupla t è d_2 .

Esempio:

Immagina una relazione tra studenti e corsi, dove:

- $D_1 = \text{Studenti}$ (insieme di tutti gli studenti),
- $D_2 = \text{Corsi}$ (insieme di tutti i corsi).

La relazione potrebbe essere rappresentata da una tabella come questa:

Studente	Corso
Anna	Matematica
Marco	Informatica

In questo caso:

- $A_1 = \text{Studente}$

- $A_2 = \text{Corso}$

Ogni riga della tabella è una tupla $t(\text{Studente}, \text{Corso})$, come $t(\text{Anna}, \text{Matematica})$.

Se volessimo accedere al valore dell'attributo **Corso** per la tupla di Anna, scriveremmo:

$t[\text{Corso}] = \text{Matematica}$

Ci riferiremo alle tabelle in modo "intensivo", con la notazione $R(A_1, \dots, A_n)$, in cui R è il nome della relazione (tabella), e $A_1 \dots A_n$ è un insieme di attributi.

Esempio: Country(ISO3,name,government,currency)

Lecture notes

Modello relazionale

Si considerino i seguenti domini (cioè insiemi di possibili valori sui quali sono definiti due attributi)

$D1 = \{\text{cane, gatto}\}$

$D2 = \{\text{bianco, nero, marrone}\}$

Una relazione R (tabella) è definita come sottoinsieme del prodotto cartesiano dei udomini degli attributi sui quali è definita la relazione R

$$R \subseteq D1 \times D2$$

$\text{animale}(\text{tipo, colore})$

prodotto cartesiano della relazione animale

tipo | colore

=====

cane	bianco
cane	marrone
cane	nero
gatto	bianco
gatto	marrone
gatto	nero

Esempio 2

$D1 = \{\text{the godfather, the batman}\}$

$D2 = \{\text{crime, comedy, action}\}$

$\text{movie}(\text{title, genre})$

the godfather,	crime
the godfather,	comedy
the godfather,	action
the batman,	crime
the batman,	comedy
the batman,	action

Ipotizziamo una relazione movie con relativi domini (tipi di dato)

$\text{movie}(\text{title, year, length})$

$\text{movie}(\text{varchar}(100), \text{char}(4), \text{integer})$

movie(D1, D2, D3)

R(D1, D2, D3) → definizione della relazione in base ai domini degli attributi

R(A1, A2, A3) → definizione della relazione in base al nome degli attributi (schema)

una tupla in una relazione è un record i cui valori sono posizionalmente coerenti con i domini della relazione

t(d1, d2, d3) → d1 ∈ D1, d2 ∈ D2, d3 ∈ D3

t[Ak] → questo denota il valore dell'attributo Ak nella tupla/record t

t[A2] → d2

Schema di una base di dati

BD = { R1(X1), R2(X2), ..., Rn(Xn) }

X1 = A11, ..., A1k

...

Xn = An1, ..., Anh

Informazione incompleta

In determinati contesti, potrebbe capitare di non aver tutte le informazioni disponibili, per ogni tupla.

Esempio:

Data una tabella Persone(Cognome, Nome, Indirizzo, Telefono), il valore dell'attributo Telefono, potrebbe non essere disponibile per tutte le tuple.

Come codificare l'informazione mancante?

Utilizzare un valore del dominio per rappresentare l'assenza di informazione, potrebbe essere una cattiva idea.

Esempio:

Se utilizzassi il valore 0 per indicare il numero di telefono mancante, in questo contesto potrebbe anche andar bene (è impossibile che un numero di telefono sia 0), ma potrebbero esserci contesti in cui il dominio non prevede valori inutilizzati.

L'idea migliore è prevedere un valore speciale detto "valore nullo" per codificare l'assenza di informazione, che nelle tabelle corrisponde al valore `null`.

Quando si definisce una relazione, è possibile specificare quali attributi ammettano valori nulli, e quali no.

Un attributo è `not null` se è indicato con un asterisco di fianco al suo nome. (cioè non ammette valori nulli)

Esempio: person(id, **firstname***, **lastname***, birth_date)

Vincoli di integrità

Date le seguenti tabelle

Studenti

Matricola	Cognome	Nome	Data di nascita
200768	Verdi	Fabio	12/02/2001
937653	Rossi	Luca	10/10/2000
937653	Bruni	Mario	01/12/2000

Esami

Studente	Voto	Lode	Corso
200768	36		05
937653	28	lode	01
937653	30	lode	04
276545	25		01

Corsi

Codice	Titolo	Docente
01	Analisi	Giani
03	Chimica	Melli
04	Chimica	Belli

Notiamo alcuni "difetti":

- Nella tabella "Esami", la prima tupla ha voto 36 (che non è ammissibile, i voti vanno da 0 a 30, o da 18 a 30 se consideriamo i soli esami passati)
- Sempre nella tabella "Esami", c'è un voto "28 e lode", che è incorretto
- Nella tabella "Studenti", le ultime due tuple hanno matricola uguale, ma informazioni diverse (qual'è quello vero? Inconsistenza)
- Nella tabella "Esami", l'ultima tupla dice che lo studente "276545" ha preso 25 nel corso "Analisi", ma lo studente "276545" non esiste nella tabella "Studenti".

Questa base di dati ha quindi evidenti problemi di inconsistenza.

Per evitare tali situazioni, si introduce il concetto di "Vincolo di integrità", cioè un predicato che deve essere vero per tutte le istanze della base di dati (tuple), per indicare che esse rappresentano informazioni corrette all'interno del contesto della base di dati.

Tali vincoli, si distinguono in:

- Vincoli intrarelazionali, cioè se riguardano regole imposte all'interno di una stessa tabella
 - Vincolo di tupla, cioè regole che devono essere rispettate all'interno di una singola tupla (Esempio: si potrebbe considerare un vincolo per cui il prezzo di vendita, deve essere sempre maggiore del prezzo di acquisto)
 - Vincolo di dominio, cioè restrizioni sul dominio di un attributo (Esempio, il dominio di "voto", va ristretto ai valori da 18 a 30)
 - Vincolo di chiave, consente al DBMS di utilizzare uno specifico attributo come identificatore di una n-upla
- Vincoli interrelazionali, riguardano regole che coinvolgono più relazioni.

Vincoli di tupla e dominio

Una possibile sintassi per esprimere i vincoli di tupla e dominio, è quella delle espressioni booleane.

Esempio:

Restringere il dominio dei voti degli esami: `(Voto ≥ 18) AND (Voto ≤ 30)`

Permettere la lode solo se il voto è 30: `(not(Lode="lode")) OR (Voto=30)` (o non c'è la lode, o il voto è pari a 30)

Oppure, se ho una tabella Pagamenti(Data,Importo,Ritenute,Netto), potrei imporre un vincolo del tipo `Netto=Importo-Ritenute`

O ancora, se io ho una tabella in cui compaiono gli attributi `alive` booleano, e `deathdate`, è impossibile avere `deathdate=null` e `alive=false`, pertanto:

((deathdate IS NOT NULL AND alive = True) OR (deathdate IS NULL))

Vincolo di chiave

Definizioni:

- Un insieme K di attributi di R , è detto superchiave di una relazione R , se per ogni coppia di tuple $t1$ e $t2$ in R , se $t1[K] = t2[K]$, allora $t1 = t2$

(In altre parole, K insieme di attributi è superchiave in una tabella, se in quella tabella gli attributi di K riescono ad identificare univocamente una tupla, non esistono due righe che abbiano gli stessi valori per tutti gli attributi di K)

Esempio: In una tabella **Studenti**(**Matricola, Nome, Cognome, DataNascita**), l'attributo **Matricola** è una superchiave, perché ogni studente ha un numero di matricola unico. Anche la combinazione (**Nome, Cognome, DataNascita**) potrebbe essere una superchiave, se assumiamo che non ci siano omonimi nati lo stesso giorno.

- K si dice chiave, se è una superchiave minimale, cioè non esiste alcun sottoinsieme $K' \subseteq K$ per cui K' rimane superchiave.

Esempio: Nella tabella **Studenti**, l'attributo **Matricola** è una **chiave**, perché è la superchiave più piccola possibile: rimuovendolo, non possiamo più distinguere univocamente gli studenti. Invece, l'insieme (**Matricola, Nome**) è una superchiave, ma **non è una chiave**, perché possiamo rimuovere "Nome" e avere comunque una superchiave minima (**Matricola**).

- **Superchiave** = Qualsiasi insieme che distingue le righe (anche con attributi extra).
- **Chiave** = La superchiave più piccola possibile (senza attributi superflui).

Scelta della chiave

Matricola	Cognome	Nome	Nascita	Corso
6328	Rossi	Dario	29/04/2000	Ing. Informatica
4766	Rossi	Luca	01/05/2001	Ing. Civile
4856	Neri	Luca	01/05/2001	Ing. Meccanica
5536	Neri	Luca	05/03/1999	Ing. Civile

In questa tabella, notiamo che l'insieme {Cognome,Corso}, è abbastanza per identificare un record della stessa tabella.

Tuttavia, possiamo dire che ciò è sempre vero? Assolutamente no, potrebbero esserci studenti di cognome uguale che seguono lo stesso corso. Possiamo dire che tale insieme è "casualmente" una chiave.

Le chiavi migliori per la relazione sopra, sono {Matricola} (Chiave surrogata) e {Cognome, Nome, Nascita} (Chiave naturale composta)

Notiamo che sugli attributi che formano una chiave, la presenza di valori nulli rappresenta un grosso problema, in quanto la presenza di un valore nullo in corrispondenza di un attributo che forma una chiave, impedirebbe la corretta identificazione della tupla, oltre che lo stabilirsi di riferimenti tra tuple di relazioni diverse.

Esempio:

Matricola	Cognome	Nome	Nascita	Corso
null	Rossi	Mario	null	Ing. Informatica
4766	Rossi	Luca	01/05/2001	Ing. Civile
4856	Neri	Luca	null	null
null	Neri	Luca	05/03/1999	Ing. Civile

In questa tabella (in cui abbiamo scelto come chiavi {Matricola} e {Cognome,Nome,Nascita}), ci sono alcuni problemi: per esempio, la prima tupla ha valori nulli sia su Matricola, che sulla Nascita, quindi nessuna delle due chiavi è utilizzabile. Tale

tupla, non è identificabile in alcun modo.

Anche le ultime due righe, anche se entrambe hanno una chiave completamente specificata (Matricola per la terza tupla, Cognome,Nome,Nascita per la quarta), risulta impossibile sapere se queste righe si riferiscono allo stesso studente o meno.

La soluzione, è impedire la presenza di valori nulli su almeno una delle chiavi (o sulla chiave, se ne è presente una sola).

Nella tabella in esempio, potrei imporre Matricola come attributo non opzionale, oppure l'insieme di Cognome, Nome, Nascita.

Esempio:

Data una tabella **movie(title,year,length)**, contenente i seguenti record:

("The batman",2023,145)
("The batman",NULL,145)
("The batman",2023,NULL)

Ho un dubbio, questi 3 record, rappresentano lo stesso film?

Se io definissi la chiave primaria come **UNIQUE(title,year,length)**, io ammetterei valori NULL, il che è evidentemente problematico.

E' quindi necessario, impedire la presenza di valori nulli sugli attributi che formano la chiave, con i vincoli:

NOT NULL(title)
NOT NULL(year)
NOT NULL(length)

Lesson notes

-- schema relazionale di una base di dati (nell'esempio la base di dati consiste di 4 relazioni/tabelle)u

-- l'asterisco denota un attributo opzionale (che può essere nullo)

-- in assenza di asterisco, l'attributo va considerato NOT NULL

movie(id, title, year, length)

person(id, firstname*, lastname*, givenname, birthdate*, deathdate*, alive(Y/N))

genre(name)

country(name, abbreviation)

-- per evitare ambiguità, per indicare un attributo si può usare la notazione puntata e prefissare il nome della tabella

genre.name

country.name

-- l'attributo alive può essere usato per distinguere il caso del valore null di un attributo dovuto a valore sconosciuto (ma esistente), dal caso di un attributo null dovuto a valore mancante

givenname deathdate alive

John 2025-03-05 N

Alice 2025-03-04 Y → violazione di un vincolo di integrità dei dati: non è possibile che una persona sia viva e abbia la data di decesso

Marta [NULL] N → Marta è deceduta, ma non conosciamo la data di decesso

Bob [NULL] Y → Bob non è deceduta, quindi il valore di deathdate è mancante

-- vincolo di integrità: quando deathdate non è nullo, il valore di alive deve essere No

```

check ((deathdate IS NOT NULL AND alive = True) OR (deathdate IS NULL))

-- consideriamo la relazione rating e i vincoli di integrità che potrebbero essere definiti sullo schema
rating(check_date, source, movie, scale, votes, score)

-- vincoli di integrità:
check(score>=0 AND score <= scale)
check(scale in (5, 10, 100))

-- vincoli di chiave:
--- superchiave (SK)
    → una qualsiasi combinazione di attributi che garantisce l'univocità dei valori nelle ennuple della relazione
    → gli attributi K della relazione R, sono superchiave se non esistono due tuple t1, t2 in R per le quali t1[K] = t2[K]

--- chiave (K)
    → una chiave è una superchiave minimale
    → K è un insieme di attributi superchiave di R. K è anche chiave di R se per un qualunque insieme S ⊆ K, possono esistere due tuple di R tali che t1[K-S] = t2[K-S]
    → in altri termini, K è chiave quando nessun attributo di K può essere eliminato senza perdere la proprietà di superchiave
    → In SQL le chiavi sono definite con il vincolo UNIQUE

--- chiave primaria (PK)
    → vincolo di entity integrity
    → una chiave primaria è una chiave sulla quale non sono possibili valori NULL
    → ogni relazione ha una e una sola chiave primaria
    → In SQL le chiavi primarie sono definite con il vincolo PRIMARY KEY

-- Esempi
movie(id, title, year, length, budget, plot)
id, title, year, length, budget, plot → SK
id → SK, K, PK
id, title → SK
id, year → SK
id, title, year → SK
title, year, length, budget, plot → SK
title, year, length → SK, K

-- queste tuple di movie, violano chiave title, year, length?
-- no perchè il valore null non è un valore
movie(title, year, length)
('The batman', 2023, 145)
('The batman', [NULL], 145)
('The batman', 2023, [NULL])

-- questi vincoli denotano che la terna title, year, length è una chiave primaria candidata
UNIQUE(title, year, length)
NOT NULL(title),
NOT NULL(year),
NOT NULL(length)

-- questi vincoli denotano che l'attributo id è una chiave primaria candidata
NOT NULL(id)
UNIQUE(id)

```

```
PRIMARY KEY(id)
```

```
country(iso3, name)
iso3 → SK, K, PK
iso3, name → SK
name → SK, K
```

```
-- vincoli su country
NOT NULL(iso3)
NOT NULL(name)
PRIMARY KEY(iso3)
```

```
cinema(name, city, address, phone)
name, city, address, phone → SK
name, city, address → SK
name, city → SK, K
name, address → SK
address → SK, K
```

```
-- vincoli su cinema
NOT NULL(name)
NOT NULL(city)
NOT NULL(address)
PRIMARY KEY(address)
```

Vincolo di integrità referenziale (o di chiave esterna)

Date due tabelle

```
movie(id, title, year, length, budget, plot)
rating(check_date, source, movie, scale, votes, score)
```

Il vincolo di integrità referenziale, interessa due relazioni R1 R2

- R1 - Relazione che "referenzia"

Nell'esempio, la relazione che referenzia è rating, poiché contiene un attributo (o insieme di attributi) detto foreign key (FK) (Nell'esempio, rating.movie), corrispondente ad UNA chiave di una relazione esterna.

- R2 - Relazione referenziata (movie)

Movie contiene l'attributo ID, chiave per la relazione stessa, che viene referenziato dall'attributo rating.movie.

Se io ho un rating con un dato valore di "movie", io sono sicuro che esiste un record nella tabella movie, con chiave primaria pari a quel valore, e che tale record sia univoco.

Garantisce che io non possa avere record di "rating" che puntino a film inesistenti.

Esempio:

Eseguendo il seguente comando [SQL](#)

```
insert into imdb.rating(check_date,source,movie,scale,score,votes) values ('2025-03-11','bdlab','004576',10,8,3,2345)
```

Il DBMS automaticamente controllerà, stabilito il vincolo di integrità referenziale sull'attributo movie, che esista uno e un solo record nella tabella movie, la cui chiave corrisponde a ' 004576 '.

In caso contrario, viene sollevato un errore SQL di violazione del vincolo di integrità referenziale

Il fatto che tale record sia univoco, possiamo dire che sia automatico, in quanto ci pensa già il vincolo di chiave nella tabella movie.

Il vero problema è verificare che effettivamente tale record nella tabella movie esista.

Anche nel caso eseguissi:

```
update rating set movie='03289' where movie='03244'
```

Tale operazione potrebbe comportare una violazione del vincolo di integrità referenziale, nel caso non esistesse un movie con id pari a `03289`.

L'operazione di eliminazione di un record

```
delete from movie where id 'XXXX'
```

Potrebbe potenzialmente generare problemi per l'integrità referenziale di rating, in caso esistessero record di rating in cui l'attributo `rating.movie='XXXX'` (Foreign Key)

In tal caso, di default, il DBMS decide di agire secondo la politica NO ACTION: se l'eliminazione di un record porta alla violazione del vincolo di integrità referenziale nella tabella rating, l'operazione non viene eseguita.

Possiamo anche indicare al DBMS di reagire in maniera diversa da NO ACTION, secondo politiche diverse, sempre garantendo di non violare il vincolo di integrità.

- CASCADE - propaga le modifiche della tabella referenziata, in quella che referenza.

Quando un record nella tabella `movie` viene eliminato o aggiornato, vengono aggiornati/eliminati anche tutti i record della tabella `rating`, la cui FK si riferisce a quello specifico record di `movie`.

- SET NULL - Quando un record della tabella `movie` viene aggiornato o modificato, i valori della chiave esterna dei record della tabella `rating` che si riferiscono a tale record, vengono impostati a `NULL`
- SET DEFAULT - alla chiave esterna della tabella secondaria viene assegnato il corrispondente valore di default al posto del valore modificato nella tabella principale;

Specifico quale politica adottare in risposta a diversi comandi DML, con la sintassi:

```
FOREIGN KEY {XXXX} REFERENCES table(attribute) ON UPDATE CASCADE ON DELETE NO ACTION
```

Vincoli in PostgreSQL

Vincolo di integrità referenziale:

```
CREATE TABLE 'books' (
    id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    author_id INTEGER,
    title varchar(255),
    published date,
    FOREIGN KEY(author_id) REFERENCES author(id)
);
```

Vincolo di tupla:

```
CREATE TABLE Item(
    Item_id INTEGER PRIMARY KEY,
    name VARCHAR(20),
    Item_price NUMERIC,
    Item_Selling_price NUMERIC CHECK(Item_Selling_price > Item_price )
);
```

Vincolo di dominio:

```
CREATE TABLE esame(
    voto NUMERIC CHECK (voto BETWEEN 18 AND 30)
);
```

Vincolo di chiave:

```
CREATE TABLE esame(
    id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
);
```

Esempio:

Date le tabelle

```
movie(id, title, year, length, budget, plot)
cinema(name, city, phone, address)
```

Un film viene proiettato in un cinema, in un determinato giorno e orario

```
projection(movie ,c_name, c_city, proj_date, hour)
```

In questa tabella, ho due Foreign Key:

- `movie` - si riferisce all'attributo movie.id
- `{c_name,c_city}` - FK composta, si riferisce alla chiave della relazione cinema

Generalmente, si evita di utilizzare chiavi esterne composte: introducendo per cinema una chiave surrogata **atomica**

```
cinema(id,name,city,phone,address)
```

La tabella `projection` si modifica con:

```
projection(movie , cinema_id , proj_date, hour)
```

Molto più comodo.

Algebra relazionale

E' un linguaggio procedurale basato su concetti di tipo algebrico, composto da un insieme di operatori, definiti su relazioni, e che producono ancora relazioni come risultati.

Definiamo gli operatori che compongono tale linguaggio:

Operatori unari

1. **Selezione (σ)**: Restituisce un sottoinsieme delle tuple che soddisfano una determinata condizione.

Sintassi: σ (Predicato) tabella

- Esempio: $\sigma(\text{salary} > 50000)$ Employee Equivalente SQL → `SELECT * FROM Employee WHERE salary > 50000`

Produce una nuova relazione, con grado uguale a quella su cui si effettua la selezione (stesso numero di attributi), ma la cui cardinalità potrebbe essere diversa.

Possiamo considerarla un'operazione di filtro.

Il Predicato "filtro", può essere articolato utilizzando gli operatori booleani. (AND,OR,NOT)

- Esempio: Trovare le pellicole prodotte fra il 2010 e il 2020 $\sigma(\text{year} \text{ BETWEEN } 2010 \text{ AND } 2020)$ movie

2. Proiezione (π): Estrae specifici attributi (colonne) di una relazione.

- Esempio: $\pi(\text{name}, \text{age})$ Employee Equivalente SQL → `SELECT name, age FROM Employee`

Produce una nuova relazione, la cui cardinalità è la stessa della relazione di partenza, ma grado strettamente minore.

Esempio:

Date le tabelle: `Movie(id,official_title,budget,year,length,plot)` e `Genre(movie,genre)`

Trovare le pellicole del 2010 che non sono thriller

R1 = $\pi(\text{id}) [\sigma(\text{year} = "2010") \text{ movie }]$

R2 = $\pi(\text{movie}) [\sigma(\text{genre} = "thriller") \text{ genre }]$

Result = R1-R2

3. Ridenominazione (ρ): Cambia il nome di una relazione o degli attributi di una relazione.

- Esempio: $\rho(\text{new_name} \leftarrow \text{old_name})$ table

Operatori binari

1. Unione (U): Restituisce tutte le tuple presenti in almeno una delle due relazioni.

- Esempio: `Employee U Manager`

Entrambe le tabelle devono avere stesso grado (numero di attributi), inoltre i loro attributi devono avere lo stesso dominio.

Per il concetto di insieme, l'operazione di unione non ammette duplicati.

2. Intersezione (\cap): Restituisce tutte le tuple che appartengono ad entrambe le relazioni.

Esempio: Trovare le pellicole che sono di genere Comedy e Romance

```
 $\pi(\text{movie}) [\sigma(\text{genre} = "comedy" \text{ AND } \text{genre} = "romance") \text{ genre}]$ 
```

Sbagliata! Il predicato viene valutato tupla per tupla.

E' impossibile che la stringa genre assuma contemporaneamente i due valori "comedy" e "romance".

Restituirà sempre un insieme vuoto.

```
R1 =  $\pi(\text{movie}) [\sigma(\text{genre} = "comedy") \text{ genre}]$ 
R2 =  $\pi(\text{movie}) [\sigma(\text{genre} = "romance") \text{ genre}]$ 
Risultato = R1  $\cap$  R2
```

In SQL, si realizza con l'operatore INTERSECT

Le due tabelle devono avere stesso grado, e tutti gli attributi lo stesso dominio

Query_1 INTERSECT Query_2

3. **Differenza (-):** Restituisce le tuple che appartengono alla prima relazione ma non alla seconda.

- Esempio: Employee – Manager

In SQL, si realizza con l'operatore EXCEPT

Le due tabelle devono avere stesso grado, e tutti gli attributi lo stesso dominio

Query_1 EXCEPT Query_2

4. **Prodotto cartesiano (x):** Combina ogni tupla della prima relazione con ogni tupla della seconda.

Esempio: Date due relazioni R1(A,B) e R2(C,D) le cui istanze sono:

A	B
A1	B1
A2	B2

C	D
C1	D1
C2	D2
C3	D3

Il prodotto cartesiano $R_1 \times R_2 = (A, B, C, D)$ produrrà le istanze:

A	B	C	D
A1	B1	C1	D1
A1	B1	C2	D2
A1	B1	C3	D3
A2	B2	C1	D1
A2	B2	C2	D2
A2	B2	C3	D3

In totale si hanno $2 \times 3 = 6$ righe.

5. **Join (⋈):** Correla dati in due relazioni in base a una condizione specifica.

Sintassi: Employee ⋈ (Employee.department_id = Department.id) Department

Esistono due tipi di join:

- Equi join (Join condizionale, correla i dati in due relazioni sulla base di una condizione booleana)
- Join naturale

Equi-Join

Date le tabelle person(id,given_name,birth_date,death_date) e crew(movie,person,role,character)

Operazione preliminare al join: prodotto cartesiano

Il prodotto cartesiano, non produce per forza valori "sensati", o meglio, non accoppia necessariamente tuple legate tra loro.

Possiamo però definire la Join come un "filtro" sul prodotto cartesiano, che mi accoppi le sole tuple che hanno qualcosa in comune, generalmente tuple in cui PK di una, e FK di un'altra, sono uguali.

Esempio:

`person(id, given_name, birth_date, death_date)`

id	given_name	birth_date	death_date
1	John	1980-05-10	NULL
2	Alice	1975-09-22	NULL
3	Bob	1960-02-15	2018-07-30

`crew(movie, person, role, character)`

movie	person	role	character
Matrix	1	Actor	Neo
Titanic	2	Actor	Rose
Inception	3	Actor	Cobb

Il **prodotto cartesiano** accoppia ogni riga della tabella `person` con ogni riga della tabella `crew`, indipendentemente da eventuali corrispondenze logiche.

La nuova tabella avrà tutte le colonne di entrambe le tabelle e $3 \times 3 = 9$ righe.

id	given_name	birth_date	death_date	movie	person	role	character
1	John	1980-05-10	NULL	Matrix	1	Actor	Neo
1	John	1980-05-10	NULL	Titanic	2	Actor	Rose
1	John	1980-05-10	NULL	Inception	3	Actor	Cobb
2	Alice	1975-09-22	NULL	Matrix	1	Actor	Neo
2	Alice	1975-09-22	NULL	Titanic	2	Actor	Rose
2	Alice	1975-09-22	NULL	Inception	3	Actor	Cobb
3	Bob	1960-02-15	2018-07-30	Matrix	1	Actor	Neo
3	Bob	1960-02-15	2018-07-30	Titanic	2	Actor	Rose
3	Bob	1960-02-15	2018-07-30	Inception	3	Actor	Cobb

Applichiamo un **filtro** per selezionare solo le tuple in cui la chiave primaria (`id` di `person`) coincide con la chiave esterna (`person` di `crew`):

`person ⚡ (person.id=crew.person) crew`

id	given_name	birth_date	death_date	movie	person	role	character
1	John	1980-05-10	NULL	Matrix	1	Actor	Neo
2	Alice	1975-09-22	NULL	Titanic	2	Actor	Rose
3	Bob	1960-02-15	2018-07-30	Inception	3	Actor	Cobb

Ora la tabella ha senso: ogni attore è associato al suo film e ruolo, eliminando le combinazioni insensate.

Esempio:

Date le tabelle:

`movie(id,title)`

`genre(movie,genre)`

`crew(person,movie)`

`person(id,given_name,birth_date)`

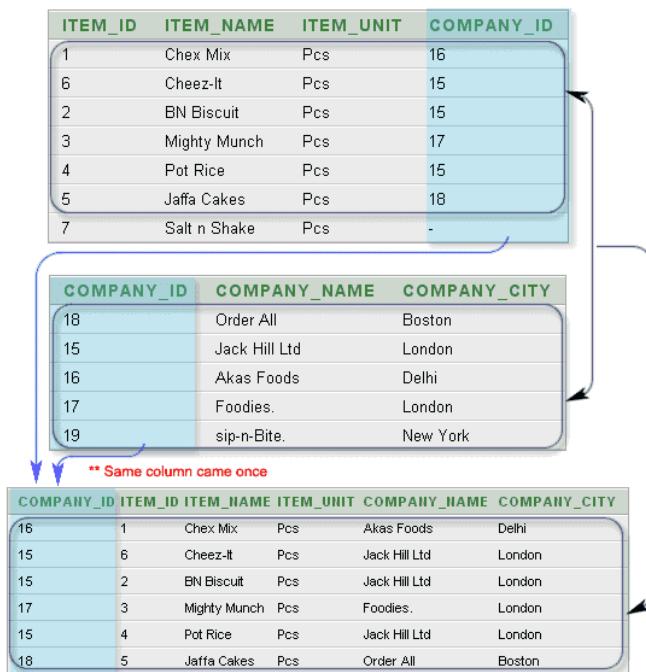
Restituire il nome delle persone nate dopo il 2000, che recitano in un film thriller.

Operiamo per gradi:

```
Person_2000 = ⚡(birth_date>2001-01-01) person
movie_thriller = ⚡(genre="thriller) genre
person_2000_join_crew = person_2000 ⚡ (id=person) crew
final_step = ⚡(person_2000_join_crew ⚡ (person_2000_join_crew.movie = movie_thriller.movie) movie_thriller
Risultato = ⚡(id,given_name) final_step
```

Natural Join

E' un particolare tipo di equi-join, che associa automaticamente tuple di relazioni diverse, in cui ci sono attributi di stesso nome, se i loro valori sono uguali.



Operatore divisione

La **divisione** è un operatore binario, che produce una tabella che contiene tutte le tuple $\langle x \rangle$ della prima relazione A tali che per ogni tupla $\langle y \rangle$ della seconda relazione B ci sia una tupla $\langle x, y \rangle$ nella prima relazione A.

$$A/B = \{ \langle x \rangle \mid \forall \langle y \rangle \in B, \langle x, y \rangle \in A \}$$

Un esempio pratico

Ho una relazione A di impiegati e una relazione B degli uffici di un'azienda

A

impiegato	ufficio
Rossi	marketing
Rossi	vendite
Bianchi	marketing

B

ufficio
marketing
vendite

WWW.ANDREADMININI.COM

Per trovare gli impiegati che lavorano in tutti i reparti si ricorre alla divisione A/B.

A/B

Nella tabella A c'è solo un valore <impiegato> tale che ogni <impiegato,ufficio> è compreso in A.

A

impiegato	ufficio
Rossi	marketing
Rossi	vendite
Bianchi	marketing

B

ufficio
marketing
vendite

WWW.ANDREADMININI.COM

Il risultato è una tabella contenente l'impiegato che lavora in entrambi i reparti.

In questo caso è l'impiegato Rossi.

impiegato
Rossi

WWW.ANDREADMININI.COM

ATTENZIONE! La divisione avviene confrontando valori di attributi con nome uguale!

Esempio:

Date le tabelle:

```
movie(id,title, year)
genre(movie,genre)
```

Trovare titolo e anno dei film che sono thriller, drama e action

```
G= σ(genre IN ("thriller","drama","action")) genre
G(movie,genre)
```

Rinomino movie in G

```
H = p(movie→id) G
H(id,genre)
```

J = movie*H (* è la natural join)

```
J(id,title,year,genre)
```

Risultato = J / π(genre)

"Trova gli elementi della prima tabella che sono associati a tutti gli elementi della seconda tabella."

Fai finta che della prima tabella, per ogni riga, consideri la tupla senza l'attributo il cui nome è sia nella prima che nella seconda tabella. Unendo quella tupla, a tutti i valori della seconda tabella, devi trovare una tupla che fa parte della prima tabella.

Viste

Le viste in Postgres, sono un modo per creare alias di altre query in un DB.

Creando una vista, ogni volta che sarà richiamato il suo nome, esso verrà tradotto in una query, eseguita in tempo reale.

Rappresentano quindi un solo aiuto semantico, per ridurre la complessità di query lunghe, ma non aumentano le prestazioni, perché la creazione di un alias non crea una nuova tabella, ma un solo modo alternativo di richiamare una query, che verrà eseguita ogni volta che viene richiamata.

One thing you'll notice is that views can improve the user experience, but they won't really ever improve performance because they don't actually run the query, they just alias it. If you want something that runs the query, you'll need a materialized view. If you want something that improves the performance, you'll need a materialized view, which runs the query and stores the result.

```
CREATE VIEW nome_vista AS (Query)
```

CTE(Common Table Expressions)

Permette di semplificare query molto lunghe, creando alias che hanno scope limitato alla singola query.

Mentre una vista crea un alias persistente, una CTE esiste solo nella query in cui essa è definita.

```
WITH regional_sales AS (
    SELECT region, SUM(amount) AS total_sales
    FROM orders
    GROUP BY region
), top_regions AS (
    SELECT region
    FROM regional_sales
    WHERE total_sales > (SELECT SUM(total_sales)/10 FROM regional_sales)
)
SELECT region,
       product,
       SUM(quantity) AS product_units,
       SUM(amount) AS product_sales
```

```
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

Subquery (o query nidificate/innestate):

Tramite le subquery, puoi riferire una query ad un risultato di un'altra query:

Esempio:

```
SELECT *
FROM Clienti
WHERE id IN (SELECT idCliente
              FROM Incassi
              WHERE importo > 1000)
```

Selezione tutti i risultati della subquery specificata.

IN:

Permette di selezionare i risultati restituiti dalla subquery.

ANY/SOME:

Si traduce in "almeno uno":

```
SELECT *
FROM Incassi
WHERE importo > SOME (SELECT stipendio
                      FROM Commesse
                      WHERE tipologia = 'commessaSemplice')
```

Per esempio, questa query seleziona le tuple in cui il relativo importo è maggiore di almeno un risultato della subquery.

ALL:

```
SELECT *
FROM Incassi
WHERE importo > ALL (SELECT stipendio
                      FROM Commesse
                      WHERE tipologia = 'commessaSemplice')
```

Questa query seleziona le tuple in cui il relativo importo è maggiore di **tutti** i risultati della subquery.

Outer Join (Su relazioni 1 a N)

Date due tabelle, **IN RELAZIONE 1 A N**:

PK	A1	ID (FK)
1	A	1

ID	B1
1	E

PK	A1	ID (FK)
2	B	1
3	C	2
4	D	NULL

ID	B1
2	F
3	G
4	H

Dalle parti in **giallo**, capisco che la relazione è una **1 a N**, data la corrispondenza della PK della prima riga di T2 in più tuple di T1.

Dalle parti in **rosso**, capisco che la relazione è **opzionale** dalla parte di **T1** (Presenza di campo con FK a **NULL**), e anche dalla parte di **T2** (Ci sono **FK** che **non sono in relazione** con alcuna entità di T1).

Se eseguissi un'operazione di **INNER JOIN** tra le due, visualizzerei:

```
SELECT * FROM T1 INNER JOIN T2 ON (T1.ID=T2.ID);
```

Output di entrambe le tabelle, perché l'operatore ***** visualizza tutti i campi di entrambe le tabelle:

PK	A1	ID (FK)	ID	B1
1	A	1	1	E
2	B	1	1	E
3	C	2	2	F

Left Outer Join:

Permette di tenere, oltre ai valori correlati tra le due tabelle, anche quelli non correlati dalla tabella di sinistra.

Permette di visualizzare i soli valori della tabella associata alla 1 o alla N, con o senza i valori in comune alle tabelle: (**Differenza tra tabelle unita eventualmente all'intersezione tra le tabelle**).

ID	B1
1	E
2	F
3	G
4	H

PK	A1	ID (FK)
1	A	1
2	3	1
3	C	2
4	D	NULL

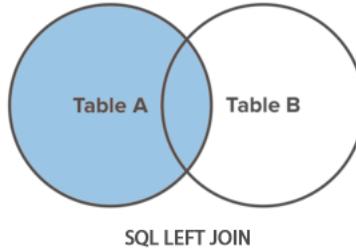
```
SELECT * FROM T2 LEFT JOIN T1 ON (T2.ID=T1.ID)
```

In Algebra relazionale: $\sigma(T1 =\!\!> T2)$

Mostra di entrambe le tabelle, i valori che solo nella tabella a **SINISTRA** (Nel modello E/R) di T2, (quindi i valori di T1), con anche i valori in comune tra le due tabelle.

Output:

PK	A1	ID (FK)	ID	B1
1	A	1	1	E
2	3	1	1	E
3	C	2	2	F
4	D	NULL	NULL	NULL



Se volessi visualizzare i valori che appartengono solo alla **tabella di sinistra**, cioè i valori che non hanno relazioni con l'altra tabella (Quindi con **FK NULL**):

Operazione di **differenza**:



```
SELECT [list] FROM
[Table A] A
LEFT JOIN
[Table B] B
ON A.value = B.value
WHERE B.value IS NULL
```

```
SELECT * FROM T2 LEFT JOIN T1 ON (T2.ID=T1.ID) WHERE T1.ID IS NULL ;
```

In Algebra relazionale: $\sigma(T1.ID \text{ IS NULL}) (T1 =\!\!> T2)$

Output:

PK	A1	ID (FK)	ID	B1
4	D	NULL	NULL	NULL

La differenza tra Left Outer Join senza intersezione ed EXCEPT, è che EXCEPT restituisce LE TUPLE che compaiono in una tabella, ma non in un'altra, la Left Outer Join senza intersezione, restituisce invece le tuple di una tabella, che non hanno corrispondenza (tramite controllo specifico) in un'altra tabella.

```
SELECT * FROM T2 LEFT JOIN T1 ON T2.ID = T1.ID WHERE T1.ID IS NULL
```

NON è equivalente a

```
SELECT * FROM T2 WHERE T2.ID IS NULL .
```

Questa query restituisce:

- Solo le righe della **tabella T2**
- In cui il **valore di T2.ID è NULL**

Quindi **filtra righe con NULL esplicito nel campo ID in T2**.

Questa è completamente diversa:

- Fa una **LEFT JOIN** da **T2** a **T1** sul campo **ID** in **T2**
- Restituisce **tutte le righe di T2**, aggiungendo colonne di **T1** dove c'è match
- Poi filtra **le righe in cui il JOIN ha fallito**, cioè **nessuna riga di T1 ha lo stesso ID**

Quindi **seleziona righe di T2 che non hanno una corrispondenza in T1 sul campo ID**

(anche se T2.ID non è NULL!)

Tabella T2

ID	Nome
1	Luca
2	Giulia
NULL	Mario
5	Anna

Tabella T1

ID	Prodotto
1	A
2	B
3	C

```
SELECT * FROM T2 WHERE T2.ID IS NULL;
```

Risultato:

ID	Nome
NULL	Mario

Restituisce **solo** la riga con **NULL** esplicito.

```
SELECT T2.*  
FROM T2  
LEFT JOIN T1 ON T2.ID = T1.ID  
WHERE T1.ID IS NULL;
```

Risultato:

ID	Nome
NULL	Mario
5	Anna

Restituisce le righe di T2 che **non hanno match in T1**:

- **NULL non può essere confrontato con nulla** ⇒ niente match
- **5 non esiste in T1** ⇒ niente match

Query	Risultato
SELECT * FROM T2 WHERE T2.ID IS NULL	Solo righe dove T2.ID è proprio NULL
SELECT * FROM T2 LEFT JOIN T1 ON ... WHERE T1.ID IS NULL	Tutte le righe di T2 senza match in T1 (inclusi NULL e non trovati)

Due usi molto diversi:

- Il primo cerca **nulli**
- Il secondo cerca **mancanza di corrispondenza**

Right join:

Identica nel funzionamento a Left Outer Join, in algebra relazionale l'uguale va messo a destra.

$\sigma(T2.ID \text{ IS NULL}) (T2 \bowtie T1) , \sigma(T2 \bowtie T1)$

Full Join:

Permette di visualizzare tutti i campi in comune e non tra le due tabelle, o solo i campi non in comune tra le due tabelle:



SELECT [list] FROM
[Table A] A
FULL OUTER JOIN
[Table B] B
ON A.Value = B.Value

SELECT * FROM T1 FULL OUTER JOIN T2 ON (T1.ID=T2.ID)

In Algebra relazionale: $\sigma(T2 =\bowtie T1) \cup \sigma(T2 \bowtie T1)$

Output:

PK	A1	ID (FK)	ID	B1
1	A	1	1	E
2	3	1	1	E
3	C	2	2	F
4	D	NULL	NULL	NULL
1	A	1	1	E
2	3	1	1	E
3	C	2	2	F
NULL	NULL	NULL	3	G
NULL	NULL	NULL	4	H

Le Prime 4 righe, sono risultato di una Left Join, contando anche i valori in comune tra le tabelle.

Le ultime righe, sono risultato di una Right Join, contando anche i valori in comune tra le tabelle.

Vediamo infatti che i valori in comune sono ripetuti nell'output.

Per visualizzare i valori di entrambe le tabelle, senza duplicati, applico l'operazione di differenza a una delle due tabelle:

SELECT * FROM T1 LEFT JOIN T2 ON (T1.ID= T2.ID) WHERE T2.ID IS NULL UNION SELECT * FROM T1 RIGHT JOIN T2 ON (T1.ID= T2.ID);

In Algebra relazionale: $\sigma(T2.ID \text{ IS NULL}) (T2 =\bowtie T1) \cup \sigma(T2 \bowtie T1)$

Oppure $\sigma(T2 =\bowtie T1) \cup \sigma(T1.ID \text{ IS NULL}) (T2 \bowtie T1)$

PK	A1	ID (FK)	ID	B1
1	A	1	1	E
2	3	1	1	E
3	C	2	2	F
4	D	NULL	NULL	NULL
NULL	NULL	NULL	3	G
NULL	NULL	NULL	4	H



Per visualizzare solo i valori non in comune tra le due tabelle, eseguo un operazione di unione delle due differenze tra tabelle:

```
SELECT * FROM T1 LEFT JOIN T2 ON (T1.ID=T2.ID) WHERE T2.ID IS NULL UNION SELECT * FROM T1 RIGHT JOIN T2 ON (T1.ID=T2.ID) IS NULL;
```

In Algebra relazionale:

$\sigma(T2.ID \text{ IS NULL}) (T2 =\text{m=} T1) \cup \sigma(T1.ID \text{ IS NULL}) (T2 \text{ m= } T1)$

Oppure: $\sigma(T2.ID \text{ IS NULL}, T1.ID \text{ IS NULL}) (T2 =\text{m=} T1)$

Output:

PK	A1	ID (FK)	ID	B1
4	D	NULL	NULL	NULL
NULL	NULL	NULL	3	G
NULL	NULL	NULL	4	H

Funzioni di aggregazione:

Le funzioni di aggregazione permettono di eseguire operazioni su **gruppi di dati**, su più valori, e su più righe, restituendo un singolo valore)

La funzione COUNT():

La funzione **COUNT()**, permette di restituire il **numero di elementi contenuto in una tabella**.

Può ricevere in **input** un valore di **qualsiasi tipo** (string,int,float,date), e ignora i campi **NULL**.

Esempio:

Data la seguente tabella:

ID	Nome	Peso	Altezza
1	Mario	70	175

ID	Nome	Peso	Altezza
2	Mario	NULL	180
3	Marco	60	190
4	Maria	80	175
5	Lucio	50	NULL

SELECT COUNT(ID) AS ConteggioTuple FROM Tabella;

Output: 5

Questa query è equivalente a:

SELECT COUNT(*) AS ConteggioTuple FROM Tabella;

SELECT COUNT(Nome) AS ConteggioNomi FROM Tabella;

Output: 5

SELECT COUNT(DISTINCT Nome) AS ConteggioNomiNonDuplicati FROM Tabella;

Output: 4

SELECT COUNT(Peso) FROM Tabella;

Output: 4 (Non vengono conteggiati i valori NULL.

SELECT COUNT(DISTINCT Altezza) FROM Tabella;

Output: 3

(Non vengono contati i valori NULL, non viene ripetuto il valore 175)

La funzione MIN() e MAX():

Le funzioni **MIN()** e **MAX()**, restituiscono i valori **numerici** o **alfanumerici** maggiori.

Possono ricevere in input dati di tipo **numerico** o **stringhe**, nel caso fossero passati attributi di tipo numerico, viene restituito il valore **numerico** minimo/massimo, nel caso di stringhe, viene restituita la stringa **lessicograficamente minore**.

Esempio:

Data la seguente tabella:

ID	Nome	Peso	Altezza
1	Mario	70	175
2	Mario	NULL	180
3	Marco	60	190
4	Maria	80	175
5	Lucio	50	NULL

SELECT MIN(Peso) AS PesoMinimo FROM Tabella;

Output: PesoMinimo: 50.

SELECT MIN(Peso) AS PesoMinimo, MAX(Peso) AS PesoMassimo FROM Tabella;

Output: PesoMinimo:50 PesoMassimo:80.

SELECT MIN(Nome) AS NomeMinimo FROM Tabella;

Output:

Lucio

SELECT MIN(Nome) AS NomeMinimo FROM Tabella WHERE ID IN(2,3,5);;

Output (Tra le righe con id 2, 3 e 5):

Lucio

SELECT MAX(Nome) AS NomeMassimo FROM Tabella WHERE ID IN(2,3,5);;

Output (Tra le righe con id 2, 3 e 5):

Mario

La funzione AVG():

La funzione **AVG()**, permette di restituire la media di campi numerici passati come input.

Può ricevere quindi in input SOLO nomi di attributi di tipo numerico, restituisce un numero di tipo **FLOAT**.

Esempio:

Data la seguente tabella:

ID	Nome	Peso	Altezza
1	Mario	70	175
2	Mario	NULL	180
3	Marco	60	190
4	Maria	80	175
5	Lucio	50	NULL

SELECT AVG(Peso) FROM Tabella;

Output:

65 (260/4)

SELECT AVG(Altezza) AS MediaAltezza FROM Tabella;

Output:

180

SELECT AVG(DISTINCT Altezza) as MediaDistinct from Tabella;

Output:

181.6 periodico (Media tra 180,190 e 175 preso una sola volta/3)

La funzione SUM():

La funzione **SUM()**, riceve in **input** solo nomi di attributi di tipo **numerico**, e restituisce la **somma** dei valori contenuti all'interno di quella tabella.

Esempio:

Data la seguente tabella:

ID	Nome	Peso	Altezza
1	Mario	70	175
2	Mario	NULL	180

ID	Nome	Peso	Altezza
3	Marco	60	190
4	Maria	80	175
5	Lucio	50	NULL

```
SELECT SUM(Peso) as SommaDeiPesi FROM Tabella WHERE Nome LIKE "M%";
```

Output:

Sommadeipesi: 210 (70 + 60 + 80, somma dei pesi delle tuple con Nome che inizia con 'M')

```
SELECT SUM(DISTINCT Altezza) as SommaAltezzaDistinct FROM Tabella;
```

Output:

SommaAltezzaDistinct: 175+190+180.

La funzione STDDEV():

La funzione **STDDEV()**, restituisce la **deviazione standard** di valori contenuti in una colonna di tipo **numerico** passata come input.

Restituisce un valore di tipo **numerico**.

GROUP BY:

Permette di unire dati i cui valori di una colonna sono uguali.

Ipotizzando di lavorare su una tabella **zone_città**, con questa struttura e dati:

Codice	Quartiere	numeroAbitanti	estensione	zona
1	Porta Mortara	10000	2000	SUD
2	Bicocca	20000	3000	SUD
3	S.Martino	15000	1500	EST
4	Sacro Cuore	30000	3500	CENTRO
5	S.Rita	20000	2500	OVEST
6	S.Andrea	10000	1500	NORD
7	S.Rocco	15000	1200	NORD

Utilizzando la keyword **GROUP BY**, unisco i valori delle tuple con valore relativo a "zona" uguale:

```
SELECT ZONA, SUM(estensione) FROM zone_città GROUP BY ZONA;
```

Come se fosse la somma delle estensioni, ma delle singole zone (come fosse un distinct, **infatti se uso la GROUP BY senza HAVING, l'output è uguale a DISTINCT.**).

Output:

Zona	Somma
SUD	5000
EST	1500
CENTRO	3500
OVEST	2500
NORD	3700

Eseguendo la stessa query con un **ORDER BY**, verrà mostrato lo stesso output, ma in maniera ordinata:

```
SELECT ZONA, SUM(estensione) FROM zone_città GROUP BY Zona ORDER BY Zona
```

Zona	Somma
SUD	5000
EST	2500
CENTRO	3500
OVEST	2500
NORD	3300

"Per ogni zona, visualizza il numero massimo di abitanti"

Query: `SELECT zona, MAX(numeroAbitanti) as NumeroMassimoAbitanti FROM zone_città GROUP BY Zona`

Output:

Zona	NumeroMassimoAbitanti
SUD	20000 (Tra le zone "SUD", il massimo è 20000)
EST	15000
CENTRO	30000
OVEST	20000
NORD	15000

"Per ogni zona, visualizza il numero di quartieri con estensione ≥ 1000 "

Query: `SELECT zona, COUNT(Quartiere) FROM zone_città WHERE estensione >= 1000 GROUP BY zona`

Output:

"Per ogni zona, quanti quartieri hanno estensione ≥ 1000 ?"

Zona	NumeroQuartieri
SUD	2
EST	1
CENTRO	1
OVEST	1
NORD	1

Quando uso una GROUP BY, devo raggruppare necessariamente per tutte le colonne della SELECT che non siano funzioni aggregate.

HAVING (Utilizzo funzioni di aggregazione come condizione di selezione):

Per utilizzare una funzione di aggregazione per filtrare determinati risultati, non posso inserirla dopo la clausola **WHERE**, ma devo utilizzare la keyword **HAVING**:

RICORDA! Nella HAVING posso usare solo colonne presenti nel GROUP BY e funzioni di aggregazione.

RICORDA! La keyword **HAVING**, deve essere sempre preceduta da una **GROUP BY**.

Esempio:

"Per ogni zona, visualizza quelle con numero di quartieri > 1".

Query: `SELECT zona,COUNT(numeroQuartieri) FROM zone_città GROUP BY zona HAVING COUNT(numeroQuartieri)>1`

Output:

zona	numeroQuartieri
SUD	2
NORD	2

Ricorda! L'Ordine per formulare le query è: **WHERE, GROUP BY, HAVING**.

Tutte le seguenti query, che non seguono questo ordine, sono **SBAGLIATE!**

MariaDB [INGENITO_Dipendenti]> `SELECT Citta,AVG(Salario),AVG(Eta) FROM Dipendenti GROUP BY Citta HAVING Count(*)>=2 AND Eta BETWEEN 20 AND 40;`
ERROR 1054 (42S22): Unknown column 'Eta' in 'having clause'

MariaDB [INGENITO_Dipendenti]> `SELECT Citta,AVG(Salario),AVG(Eta) FROM Dipendenti GROUP BY Citta HAVING Count(*)>=2 WHERE Eta BETWEEN 20 AND 40;`
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near 'WHERE Eta BETWEEN 20 AND 40' at line 1

MariaDB [INGENITO_Dipendenti]> `SELECT Citta,AVG(Salario),AVG(Eta) FROM Dipendenti GROUP BY Citta HAVING Count(*)>=2 Eta BETWEEN 20 AND 40;`
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near 'Eta BETWEEN 20 AND 40' at line 1

MariaDB [INGENITO_Dipendenti]> `SELECT Citta,AVG(Salario),AVG(Eta) FROM Dipendenti GROUP BY Citta WHERE eta BETWEEN 20 AND 40 HAVING Count(*)>=2;`
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near 'WHERE eta BETWEEN 20 AND 40 HAVING Count(*)>=2' at line 1

MariaDB [INGENITO_Dipendenti]> `SELECT Citta,AVG(Salario),AVG(Eta) FROM Dipendenti GROUP BY Citta WHERE eta BETWEEN 20 AND 40 HAVING Count(Citta)>=2;`
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near 'WHERE eta BETWEEN 20 AND 40 HAVING Count(Citta)>=2' at line 1

MariaDB [INGENITO_Dipendenti]> `SELECT Citta,AVG(Salario),AVG(Eta) FROM Dipendenti GROUP BY Citta HAVING Count(*)>=2 Eta BETWEEN 20 AND 40;`

```
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near 'Eta BETWEEN 20 AND 40' at line 1
```

```
MariaDB [INGENITO_Dipendenti]> SELECT Citta,AVG(Salario),AVG(Eta) FROM Dipendenti GROUP BY Citta HAVING Count(*)>=2 WHERE Eta BETWEEN 20 AND 40;
```

```
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near 'WHERE Eta BETWEEN 20 AND 40' at line 1
```

```
MariaDB [INGENITO_Dipendenti]> SELECT Citta,AVG(Salario),AVG(Eta) FROM Dipendenti HAVING Count(*)>=2 WHERE Eta BETWEEN 20 AND 40 GROUP BY Citta;
```

```
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near 'WHERE Eta BETWEEN 20 AND 40 GROUP BY Citta' at line 1
```

Rappresentazione di gerarchie

Ipotizziamo di voler rappresentare una struttura gerarchica del tipo:

```
Thriller
- Noir
  - Poliziesco
  - Spionaggio
  - Cronaca nera
- Splatter
```

Possiamo rappresentare ogni genere con un nodo, ognuno contenente un riferimento al nodo padre.

```
create table imdb.genre_hierarchy (
    genre_name varchar primary key,
    genre_parent varchar
);

alter table imdb.genre_hierarchy add constraint parent_fk foreign key (genre_parent) references imdb.genre_hierarchy
(genre_name);
```

Tramite il vincolo di primary key, impediamo l'inserimento di nodi duplicati, mentre tramite il vincolo di integrità referenziale (chiave esterna) su genre_parent, imponiamo l'inserimento di soli valori esistenti, come padre di un nodo.

Ora possiamo inserire i valori.

```
insert into imdb.genre_hierarchy values ('Thriller',null) //nodo radice, non ha padre
insert into imdb.genre_hierarchy values ('Noir','Thriller')
insert into imdb.genre_hierarchy values ('Splatter','Thriller')
insert into imdb.genre_hierarchy values ('Poliziesco','Noir')
insert into imdb.genre_hierarchy values ('Spionaggio','Poliziesco')
insert into imdb.genre_hierarchy values ('Cronaca nera','Poliziesco')
```

Ora possiamo chiederci:

"Restituire i sopra-generi di Poliziesco"

Cioè vogliamo Poliziesco → Noir → Thriller

```

WITH RECURSIVE search_parent(the_genre,target_genre) AS (
    SELECT genre_name,genre_parent FROM imdb.genre_hierarchy WHERE genre_name='Poliziesco'
    UNION ALL
    SELECT sp.the_genre, gh.genre_parent FROM search_parent sp INNER JOIN imdb.genre_hierarchy gh ON sp.target_
    genre = gh.genre_name
)
SELECT * FROM search_parent;

```

Questa query funziona così:

Il caso base, seleziona la riga, composta da genre_name e genre_parent, il cui genere è "poliziesco".

the_genre	target_genre
Poliziesco	Noir

Presà questa riga, la unisce ricorsivamente, con la riga il cui genre_name è uguale al suo genre_parent.

the_genre	target_genre
Poliziesco	Thriller

Continua ad effettuare join ricorsivamente, finchè non si arriva a "joinare" con un record la cui FK è NULL. L'inner join fallirà, e la query termina.

the_genre	target_genre
Poliziesco	Noir
Poliziesco	Thriller
Poliziesco	NULL

Versione con introduzione del parametro "distance"

Restituire i primi 2 sopra-generi del genere "poliziesco"

```

WITH RECURSIVE search_parent(the_genre,target_genre,distance) AS (
    SELECT genre_name,genre_parent,1 FROM imdb.genre_hierarchy WHERE genre_name='Poliziesco'
    UNION
    SELECT sp.the_genre, gh.genre_parent,distance+1 FROM search_parent sp INNER JOIN imdb.genre_hierarchy gh ON
    sp.target_genre = gh.genre_name WHERE distance<1
)
SELECT * FROM search_parent;

```

La condizione del WHERE, è distance<1, perchè è considerata una POST-CONDIZIONE, nel senso che prima la select stamperà distance+1, poi controllerà la clausola del where

Trigger e asserzioni

Contenuto di una base di dati:

- Tabelle
- Viste
- Procedure

- Trigger
- Asserzioni

I Trigger (Inneschi) permettono di definire un comportamento automatico della base di dati, a fronte di uno specifico evento sui dati.

In particolare, consentono l'esecuzione di una procedura, in risposta a eventi di INSERT, UPDATE o DELETE.

Possono essere eseguiti:

- **BEFORE** (prima dell'operazione),
- **AFTER** (dopo l'operazione),
- oppure (in alcuni DBMS) **INSTEAD OF** (al posto dell'operazione, utile soprattutto per viste).

Sintassi:

```
CREATE TRIGGER <nome_trigger> {BEFORE | AFTER} <evento> ON <nome_tabella> [FOR EACH] {ROW | STATEMENT} ] EXECUTE PROCEDURE <nome_funzione>
```

La procedura chiamata, deve ricevere 0 parametri e restituire un trigger.

Le opzioni FOR EACH ROW e FOR EACH STATEMENT permettono di definire se, a seguito dell'operazione di innesco, il trigger va attivato singolarmente per ogni n-upla coinvolta, oppure una volta sola per l'intera istruzione.

Esempio: Voglio tenere traccia per ogni film, di alcune informazioni statistiche, per esempio il numero di attori per ogni film.

Se queste informazioni sono poco richieste, posso risolvere tale necessità utilizzando una query.

Se però tali informazioni sono richieste spesso, l'esecuzione di una query, che potrebbe interessare un numero molto grande di istanze, potrebbe rappresentare un grande problema di performance.

Creo allora una tabella, che contiene per ogni film, il numero di attori, registri ecc... all'interno del suo cast.

```
create table movie_counts (
    movie varchar references movie(id),
    p_role varchar,
    m_count integer,
    primary key(movie,p_role)
)
```

Ora, popoliamo la tabella, in modo che ogni film, abbia 4 record, ognuno con il numero di producer, actor, director, writer all'interno del proprio cast.

```
with roles as (select distinct p_role from imdb.crew)
with movie_roles as (select movie.id, p_role from imdb.movie,roles) //prodotto cartesiano, lego ogni film a tutti e 4 i ruoli

select movie.id,p_role,count(person) from movie_roles left join imdb.crew ON movie_roles.id=imdb.crew.movie and movie_roles.p_role=imdb.crew.p_role GROUP BY movie_roles.id,imdb.crew.p_role ORDER BY movie_roles.id
```

Questa tabella, deve rimanere costantemente aggiornata, per garantire la consistenza dei dati.

Ogni volta che un record viene inserito nella tabella crew, devo aggiornare la tabella, identificando il record interessato in base al film e al ruolo.

```
create trigger update_counts before insert on imdb.crew for each row execute procedure do_count_increment();

create procedure do_count_increment()
```

```

returns trigger as $$

begin
    update movie_counts set m_count = m_count+1 where movie=new.movie and p_role=new.p_role;
end;

$$language plpgsql

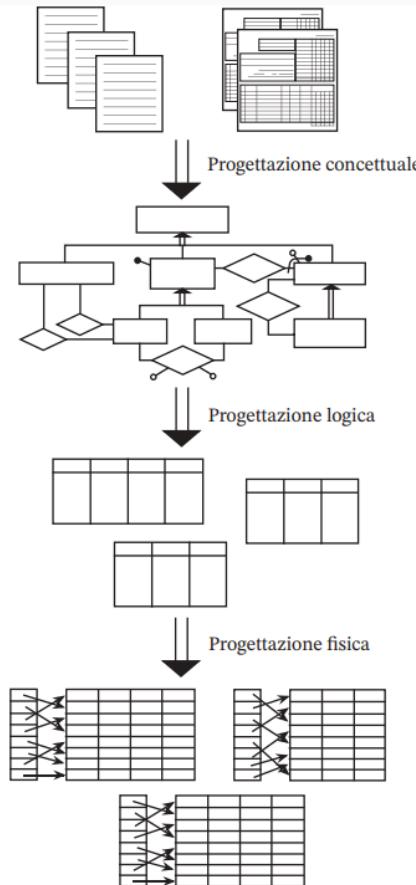
```

Definisco un trigger, associato alla tabella crew, che si attiva prima di effettuare un insert su quella tabella.

La keyword NEW nella procedura, si riferisce al nuovo record che sta per essere inserito.

Progettazione di una base di dati

La metodologia di progettazione di una base di dati più comunemente utilizzata, consiste in 3 fasi che permettono di separare in maniera netta "cosa" rappresentare, dal "come" farlo.



1. Progettazione concettuale - Si produce uno schema concettuale, che permetta di inquadrare formalmente le informazioni da rappresentare, senza preoccuparsi delle modalità con cui esse verranno codificate all'interno del sistema informatico.
Lo strumento principale utilizzato in questa fase è il modello entità relazione, o sue varianti (Esempio: UML Diagram).

Esempio:

Sistema universitario semplice:

- **Entità:** `Studente(matricola, nome, cognome, data_nascita)`,

`Corso(codice, nome, CFU)`

- **Relazione:**

`Iscritto(Studente, Corso, data_iscrizione)`

(cardinalità: molti-a-molti)

2. Progettazione logica - Traduzione dello schema concettuale, in termini del modello di rappresentazione dei dati adottato dal DBMS a disposizione.

Il prodotto di questa fase viene detto schema logico, e fa riferimento ad un modello logico della base di dati (nel nostro caso, verranno prodotte relazioni, che fanno riferimento al modello relazionale della base di dati).

Le entità diventano relazioni, con attributi e chiave primaria

Le relazioni molti a molti vengono rappresentate con una tabella separata con chiavi esterne

Le relazioni 1:N e 1:1 mediante utilizzo di chiave esterna

Le entità deboli come chiave primaria composta+ chiave esterna

Esempio:

Tabelle risultanti:

- `Studente(matricola PK, nome, cognome, data_nascita)`
- `Corso(codice PK, nome, CFU)`
- `Iscritto(matricola FK → Studente, codice FK → Corso, data_iscrizione, PRIMARY KEY(matricola, codice))`

3. Progettazione fisica - Lo schema logico viene ottimizzato per l'implementazione in uno specifico DBMS.

Vengono definiti i tipi di dato concreti da utilizzare, i vincoli, i privilegi di accesso alle operazioni SQL.

Infine, vengono definiti i comandi da eseguire per l'implementazione della base di dati.

Esempio:

```
CREATE TABLE Studente (
    matricola CHAR(10) PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    cognome VARCHAR(100) NOT NULL,
    data_nascita DATE NOT NULL
);

CREATE TABLE Corso (
    codice CHAR(5) PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    CFU SMALLINT CHECK (CFU > 0)
);

CREATE TABLE Iscritto (
    matricola CHAR(10),
    codice CHAR(5),
    data_iscrizione DATE NOT NULL,
    PRIMARY KEY (matricola, codice),
    FOREIGN KEY (matricola) REFERENCES Studente(matricola) ON DELETE CASCADE,
    FOREIGN KEY (codice) REFERENCES Corso(codice)
);
```

```
-- Indice per ottimizzare le query sui corsi più frequentati
CREATE INDEX idx_iscritto_codice ON Iscritto(codice);
```

Progettazione concettuale - Modello ER

Modello concettuale di dati, che fornisce una serie di costrutti atti a descrivere la realtà di interesse in maniera facile e indipendente dall'organizzazione dei dati nel calcolatore.

Costrutti principali del modello:

Costrutti	Rappresentazione grafica
Entità	Rettangolo
Relazione	Rombo
Attributo semplice	Linea con un nodo
Attributo composto	Linea con due nodi
Cardinalità di relazione	(m_1, M_1) e (m_2, M_2)
Cardinalità di attributo	(m, M)
Identificatore interno	Rettangolo con un punto nero
Identificatore esterno	Rettangolo con rombo e punto nero
Generalizzazione	Rettangolo con rombo
Sottoinsieme	Due rettangoli collegati da un'arrows

Entità

Rappresentano classi di oggetti (fatti, cose, persone), con proprietà comuni ed esistenza "autonoma"

Esempio: Entità "Città", occorrenze di tale entità potrebbero essere "Milano", "Roma", "Palermo".

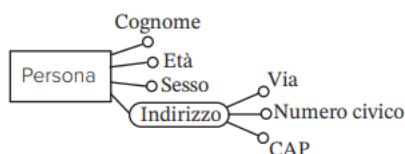
Relazioni

Legami logici tra entità. Nel modello ER, ogni relazione (inteso come legame) è un rombo con all'interno un nome.

Per non assegnare un verso alla relazione, è buona pratica assegnare un sostantivo al nome della relazione, piuttosto che un verbo

Esempio: con le entità Azienda e Operaio, la relazione "Lavora in" imporrebbre un verso da Operaio ad Azienda. Un nome alternativo e più corretto potrebbe essere "Sede di lavoro".

Attributi



Descrivono proprietà di entità o relazioni.

Possono essere atomici (valori singoli, esempio "Cognome") o composti (Residenza, composto da Via, CAP, Civico).

Cardinalità delle relazioni

Vengono specificate per ciascuna partecipazione di un'entità ad una relazione, e descrivono il numero minimo e massimo di occorrenze di entità che possono partecipare ad una relazione.



Ad un impiegato, può essere assegnato un numero da 1 a 5 incarichi.

Ogni incarico, può essere assegnato ad un numero da 0 a 50 impiegati.

La cardinalità minima deve essere minore o uguale della cardinalità massima.

Nella maggior parte dei casi, è sufficiente usare tre valori: 0,1 ed il simbolo N.

Se la cardinalità minima è 0, si dice che la relazione è opzionale, altrimenti si dice obbligatoria.

Per identificare la cardinalità di una relazione, il modo corretto di ragionare è vincolare un'istanza di un'entità, e chiedersi "questa specifica istanza, a quante istanze dell'altra relazione può essere associata?"

Cardinalità degli attributi

Descrivono il numero minimo e massimo di valori dell'attributo associati ad ogni occorrenza di entità o relazione.

Nella maggior parte dei casi è (1,1), in tal caso può essere omessa.

Identificatori delle entità

Attributi (o insieme di attributi) che permettono di identificare univocamente le occorrenze delle entità.

Un identificatore è **interno**, se è scelto tra gli attributi della tabella stessa.

Esempio: Tabella Automobile(Modello,Targa,Colore), l'attributo Targa è abbastanza per identificare univocamente un veicolo.



Identificatore interno, atomico e composto

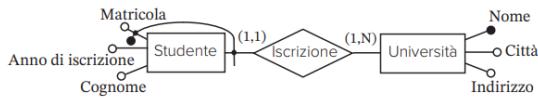
Entità debole

Alcune volte, gli attributi di un'entità non sono sufficienti ad identificare univocamente le sue occorrenze.

Esempio: Tabella Studente(Matricola,Anno_Iscrizione,Cognome), ipotizzando che tale tabella contenga studenti provenienti da diverse università, allora la matricola non sarebbe più in grado di identificare univocamente uno studente, perché in università diverse, potrebbero essere presenti studenti con stessa matricola.

Per identificare uno studente, oltre alla sua matricola, è necessario sapere in che università studia. A questo punto, possiamo stabilire un identificatore **esterno**, composto dalla matricola dello studente, e dall'identificatore dell'università.

E' necessario che lo studente partecipi ad una relazione con l'università ESCLUSIVAMENTE con cardinalità (1,1), altrimenti, se lo studente potesse essere iscritto a più università contemporaneamente, dovremmo considerare il caso in cui esso abbia stessa matricola per entrambi gli atenei.



Studente(Matricola, NomeUniversità, Cognome, AnnoIscrizionne)
Università(Nome, Città, Indirizzo)

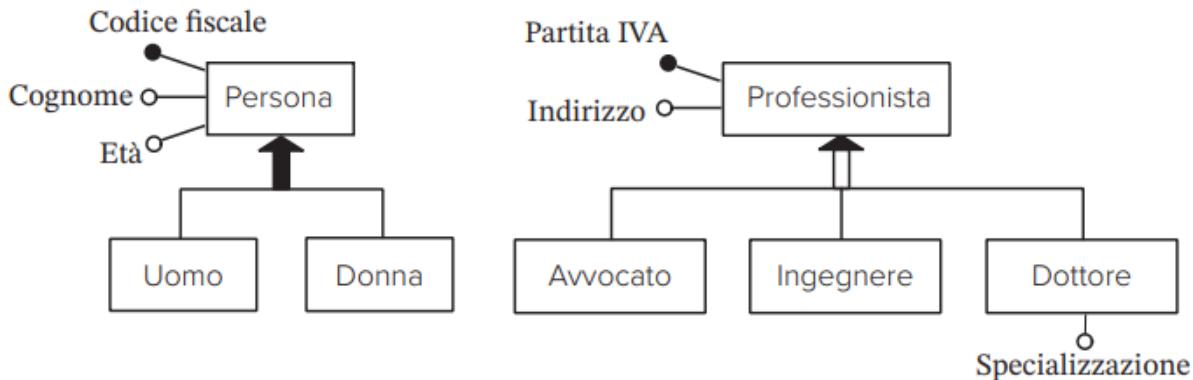
Con tale sintassi, indichiamo che lo studente è univocamente identificabile tramite la sua matrícula, solo all'interno del contesto di una singola università.

In questo caso, l'entità Studente si dice entità debole, poiché univoca solo se associata ad un'altra entità (università).

Generalizzazioni (Relazioni is-a)

Rappresentano gerarchie in cui compaiono entità "padri" (più generali) ed entità "figlie" (più specifiche, specializzazioni delle entità padri).

- Ogni occorrenza di un'entità figlia, è anche un'occorrenza dell'entità genitore (un uomo, è anche una persona)
- Ogni proprietà (attributi, relazioni...) dell'entità padre, è ereditato anche dalle entità figlie (Sia uomini che donne avranno CF, Cognome, Età).



Con tale sintassi, congiungiamo entità padri e figlie.

- Una generalizzazione è totale, se ogni occorrenza dell'entità genitore è un'occorrenza di almeno una delle entità figlie, altrimenti è parziale.

Nel modello E/R, la freccia piena indica una generalizzazione totale, vuota indica una generalizzazione parziale.

Nel caso di Uomo e Donna, ogni persona è collocabile in una delle due categorie, non ci sono persone che rimangono "fuori", la relazione è totale.

Nel caso di Professionista, la relazione è parziale (Ci sono altre professioni oltre ad Avvocato, Ingegnere, Dottore)

- Una generalizzazione è esclusiva, se ogni occorrenza dell'entità genitore è collocabile in una e una sola specializzazione (entità figlia), altrimenti è sovrapposta.

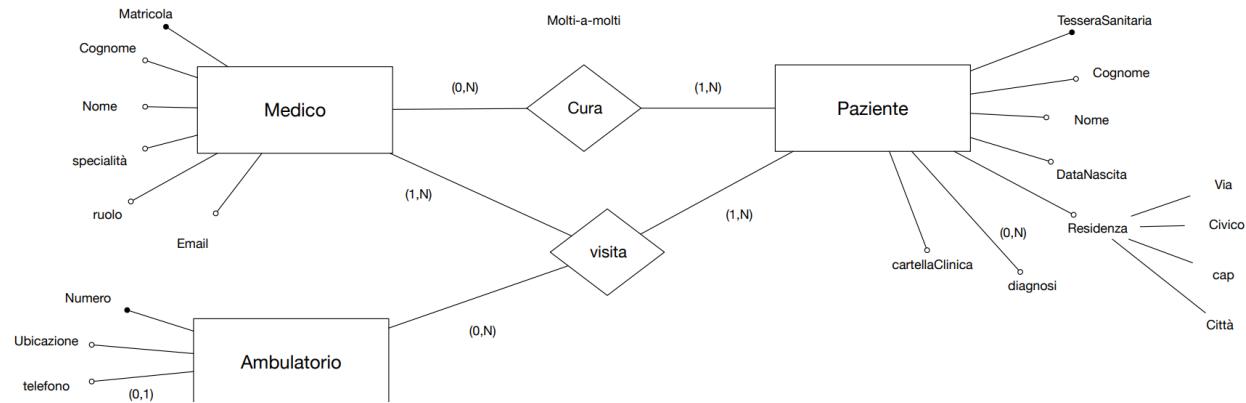
Nel caso di Uomo e Donna, non ci sono persone che possono essere contemporaneamente Uomini e Donne, la relazione è esclusiva.

Nel caso di Professionista, la relazione è esclusiva. Un professionista può svolgere due professioni contemporaneamente.

Progettazione logica

Traduzione del modello ER alla definizione delle tabelle.

Si modelli un diagramma ER che descrive una realtà ospedaliera in cui i medici curano i pazienti.
I pazienti sono visitati dai medici in ambulatorio



Partendo dal modello ER sopra, ogni entità diventa una tabella separata.

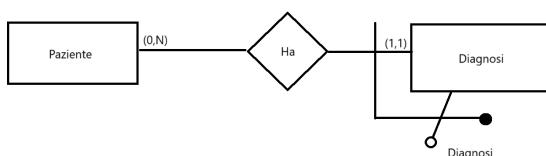
Tabelle: Medico, Paziente, Ambulatorio.

Ad ognuna di queste tabelle, iniziamo ad aggiungere gli attributi che compaiono nel modello ER, sottolineando la Primary Key e ponendo un asterisco vicino agli attributi opzionali (cardinalità minima pari a 0) per convenzione.

Gli attributi **composti**, possono essere trattati in due modi:

- Scomporli in attributi atomici, e rappresentarli singolarmente.
- Concatenarli e memorizzarli in un unico attributo.

La scelta della metodologia più adatta dipende dalla realtà di studio. Se pensiamo che possano essere realizzate query che si riferiscono ai singoli componenti dell'attributo atomico, allora potrebbe essere utile scomporli, altrimenti la soluzione tramite concatenazione può essere utilizzata per semplificare la struttura.



Nel modello, compare anche un attributo multiplo (Diagnosi), che potrebbe contenere 0 o N valori.

In questo specifico caso, l'attributo Diagnosi diventa un'entità debole, in relazione 1:N con la tabella di partenza (Paziente)

Il fatto che l'entità sia debole, vuol dire che la Primary Key di Diagnosi, è la combinazione di Diagnosi e della Chiave Primaria di Paziente.

All'interno della tabella Diagnosi quindi, avremo gli attributi Diagnosi, TesseraSanitaria (FK), e la chiave primaria sarà la combinazione dei due.

Regola generale: Gli attributi multipli, diventano tabelle a se, con un solo attributo, pari al nome dell'attributo. Le cardinalità sono (1,1) dal lato della nuova tabella, (0,N) o (1,N) dalla parte dell'entità principale, in base a se il possedimento dell'attributo era obbligatorio o no

Tabelle:

Medico(Matricola ,Cognome,Nome,Specialità,ruolo,email)

Ambulatorio(Numero ,Ubicazione,Telefono*)

Paziente(TesseraSanitaria,Cognome,Nome,DataNascita,Residenza,Diagnosi, cartellaClinica) FK Paziente.Diagnosi → Diagnosi.Diagnosi

Diagnosi(Diagnosi,TesseraSanitaria) FK Diagnosi.TesseraSanitaria → Paziente.TesseraSanitaria)

Passiamo ora a trattare le relazioni Cura (N:N) e Visita (Ternaria).

La relazione Cura N:N, viene rappresentata come una tabella a se, in cui ogni record conterrà l'associazione delle FK delle due tabelle interessate

Cura(Medico,Paziente) FK Cura.Medico → Medico.Matricola, Cura.Paziente → Paziente.TesseraSanitaria

La chiave primaria sarà data dall'unione di Medico e Paziente.

Visita, è una N:N tra 3 entità: Anch'essa diventa una tabella a se

Visita(Medico,Paziente,Ambulatorio) FK Visita.Medico → Medico.Matricola, Visita.Paziente → Paziente.TesseraSanitaria, Visita.Ambulatorio → Ambulatorio.Numero

La chiave primaria sarà data dall'unione di Medico, Paziente e Ambulatorio

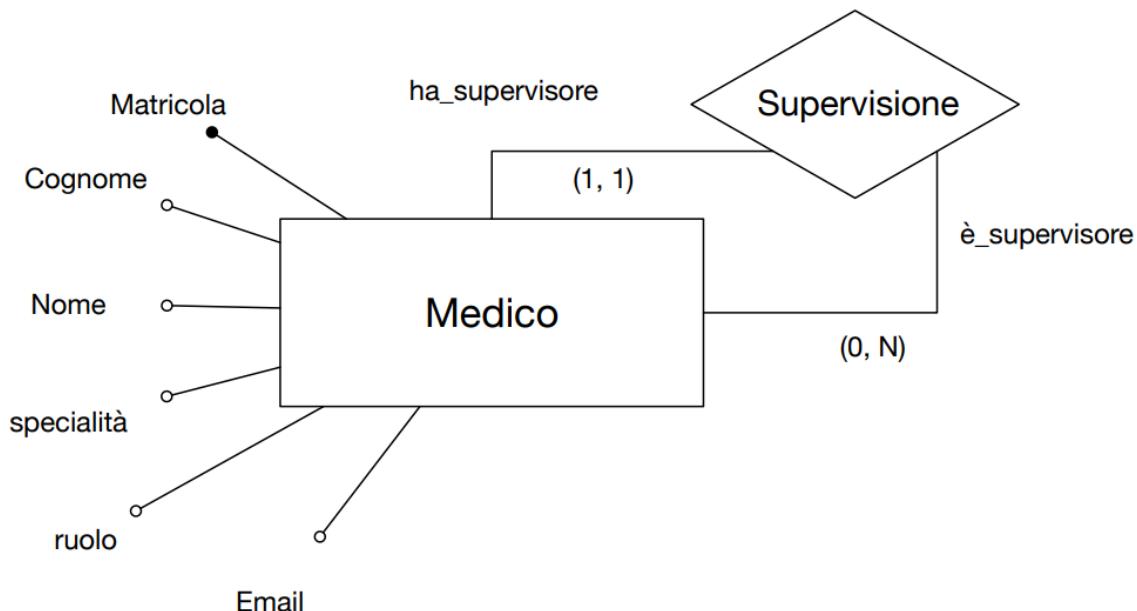


Tabella: Medico(Matricola ,Cognome,Nome,Specialità,Ruolo,Email,Supervisore) FK Medico.Supervisore → Medico.Matricola

Un medico ha uno e un solo supervisore, pertanto la FK è NOT NULL.

Traduzioni di gerarchie (Relazioni is-a)



3 metodologie di mapping

- Rappresentazione tramite tabelle separate

Tengo una tabella per la superclasse, e tante tabelle quante sono le sottoclassi, collegate alla superclasse mediante una chiave esterna

Esempio: Date le tabelle

Entità generale: Persona (codice_fiscale, nome, cognome)
Sottoclassi: Studente (matricola, corso_di_studio)
Docente (matricola_docente, dipartimento)

La traduzione produce:

```
Persona(codice_fiscale PK, nome, cognome)
Studente(codice_fiscale PK FK → Persona, matricola, corso_di_studio)
Docente(codice_fiscale PK FK → Persona, matricola_docente, dipartimento)
```

- Collazzo del padre

Rappresento le sole sottoclassi, duplicando in esse gli attributi della superclasse

```
Studente(codice_fiscale PK, nome, cognome, matricola, corso_di_studio)
Docente(codice_fiscale PK, nome, cognome, matricola_docente, dipartimento)
```

Questa metodologia non è compatibile con relazioni is-a parziali. Se ci sono entità che non appartengono a nessuno dei figli, con questa trasformazione non riuscirei a rappresentarle

- Collazzo dei figli

Rappresento la gerarchia mediante un'unica tabella, che contiene tutti gli attributi della superclasse e delle sottoclassi.

Aggiungo un attributo discriminante, che identifica il tipo di entità all'interno della gerarchia

```
Persona(
codice_fiscale PK,
nome,
cognome,
ruolo CHECK (ruolo IN ('Studente', 'Docente')),
matricola, -- solo per Studente, quindi NULLABLE
corso_di_studio, -- solo per Studente, quindi NULLABL
matricola_docente, -- solo per Docente, quindi NULLABL
```

dipartimento

-- solo per Docente, quindi NULLABL

)

Traduzioni di relazioni

Tipologia	Concetto iniziale	Risultati possibili
Associazione binaria molti a molti		$E_1(A_{E11}, A_{E12})$ $E_2(A_{E21}, A_{E22})$ $R(A_{R1}, A_{R2}, A_1, A_2)$
Associazione ternaria molti a molti		$E_1(A_{E11}, A_{E12})$ $E_2(A_{E21}, A_{E22})$ $R(A_{R1}, A_{R2}, A_1, A_2)$
Associazione uno a molti con partecipazione obbligatoria		$E_1(A_{E11}, A_{E12}, A_1)$ $E_2(A_{E21}, A_{E22})$
Associazione uno a molti con partecipazione opzionale		$E_1(A_{E11}, A_{E12})$ $E_2(A_{E21}, A_{E22})$ Oppure: $E_1(A_{E11}, A_{E12}, A_1)$ $E_2(A_{E21}, A_{E22})$
Associazione con identificatore esterno		$E_1(A_{E11}, A_{E12}, A_1)$ $E_2(A_{E21}, A_{E22})$

Tipologia	Concetto iniziale	Risultati possibili
Associazione uno a uno con partecipazione obbligatoria per entrambe le entità		$E_1(A_{E11}, A_{E12}, A_1)$ $E_2(A_{E21}, A_{E22}, A_2)$ Oppure: $E_1(A_{E11}, A_{E12}, A_1)$ $E_2(A_{E21}, A_{E22})$
Associazione uno a uno con partecipazione opzionale per un'entità		$E_1(A_{E11}, A_{E12}, A_1)$ $E_2(A_{E21}, A_{E22})$
Associazione uno a uno con partecipazione opzionale per entrambe le entità		$E_1(A_{E11}, A_{E12})$ $E_2(A_{E21}, A_{E22})$ Oppure: $E_1(A_{E11}, A_{E12}, A_1)$ $E_2(A_{E21}, A_{E22})$ Oppure: $E_1(A_{E11}, A_{E12})$ $E_2(A_{E21}, A_{E22}, A_2)$

Reverse Engineering

Partendo dal modello Logico, costruire il modello concettuale.

Passi:

1. Ricostruzione tabelle naturali:

Riconosco le tabelle in cui compare un ID o Primary Key non composta da FK: In tal caso, la tabella diventa un'entità propria.

2. Provo a capire le tabelle con chiavi esterne:

- Entità deboli:** Hanno come chiave un identificatore esterno oppure composto da un identificatore esterno. Cercare PK che siano composte da FK.

La Cardinalità associata ad entità deboli è sempre (1,1).

In origine le entità deboli potrebbero essere state un attributo multivale.

- Molti a molti (associazione):** Chiave primaria composta in cui gli attributi che la formano, sono delle FK che puntano a tabelle diverse.

Riconoscere presenza di attributi associati alla relazione.

Se una associazione ha un attributo in più nella chiave, probabilmente è stato inserito per un vincolo extra-schema, va inserito come attributo dell'associazione.

Esempio:

```
CREATE TABLE certificate_request (
    r_date DATE NOT NULL,
    person INTEGER NOT NULL REFERENCES person(id),
    certificate INTEGER NOT NULL REFERENCES certificate(id),
    PRIMARY KEY (r_date, person, certificate)
);
```

In questa tabella, ho una PK composta da due FK: suggerisce una relazione N:N tra person e certificate. La presenza di r_date, indica che una persona può richiedere un certificato una sola volta al giorno. Va disegnato come attributo della relazione `certificate_request`.

Senza informazioni aggiuntive ho (0,N) (0,N).

Per le associazioni N:N, non ho modo di riconoscere la cardinalità minima, mettere sempre (0,N).

- **Associazione ricorsiva (N:N):** 2 FK che si riferiscono stessa tabella.

Esempio:

Immagina una tabella `Persona(CF, Nome, Cognome)` e vuoi modellare che **una persona può supervisionare un'altra persona**.

Serve quindi una relazione tra persone:

`Supervisione(Supervisore, Supervisionato)`

- `Supervisore` è una FK a `Persona.CF`
- `Supervisionato` è una FK a `Persona.CF`
- Entrambe le colonne fanno riferimento alla **stessa tabella**, ma indicano **due ruoli diversi**

- **Molti a uno:** Cerchiamo nella tabella una FK che non sia parte della PK.

La presenza di una FK in una tabella, potrebbe suggerire una relazione 1:1 oppure 1:N.

Se la FK è specificata con vincolo UNIQUE, è una 1:1, altrimenti si specifica la relazione più generica possibile, cioè 1:N.

Nel reverse engineering perdo delle informazioni.
(entità deboli caso particolare in cui FK è nella PK)

Quando trovo una FK in una tabella, se è specificata con vincolo NOT NULL, mi suggerisce cardinalità minima 1. Se specificata con vincolo UNIQUE, cardinalità massima è 1. Se ci sono entrambi, cardinalità 1,1

Caso	Interpretazione	Cardinalità minima	Cardinalità massima
FK senza vincoli	Relazione 1:N (generica)	0	N
FK con NOT NULL	Cardinalità minima 1	1	N
FK con UNIQUE	Cardinalità massima 1 → relazione 1:1 (potenzialmente)	0	1
FK con NOT NULL + UNIQUE	Relazione 1:1 obbligatoria	1	1

- **Gerarchia:**

Posso riconoscerle solo in caso di rappresentazione tramite tabella padre + tabelle figlie

```
CREATE TABLE person (
    id INT PRIMARY KEY,
    name VARCHAR
);
```

```
CREATE TABLE student (
    id INT PRIMARY KEY REFERENCES person(id),
    major VARCHAR
);
```

```
CREATE TABLE professor (
    id INT PRIMARY KEY REFERENCES person(id),
    department VARCHAR
);
```

In questo caso, riconosco la presenza di PK che sono anche FK alla tabella padre.

Qui la gerarchia è sovrapposta (non c'è nessun vincolo che impedisca ad una stessa entità person di essere associata a student che professor), e parziale (non c'è nessun vincolo che imponga ad una persona di essere associata per forza ad una classe figlia).

Oppure, presenza di attributo discriminante

```
CREATE TABLE person (
    id INT PRIMARY KEY,
    name VARCHAR,
    category VARCHAR CHECK (category IN ('Student', 'Professor'))
);
```

E' una gerarchia totale (category non è NULL, una person deve essere necessariamente uno studente o un professore) e disgiunta (una persona non può essere contemporaneamente studente o professore, a category posso associare un solo valore alla volta).

Normalizzazione

Le forme normali, sono proprietà che "certificano" la qualità dello schema di una base di dati relazionale, e sono soddisfatte quando non ci sono anomalie.

Quando una relazione non soddisfa una forma normale, allora presenta ridondanze, e si presta a comportamenti poco desiderabili durante le operazioni di aggiornamento.

Esempio:

Impiegato	Stipendio	Progetto	Bilancio	Funzione
Rossi	20 000	Marte	2000	Tecnico
Verdi	35 000	Giove	15 000	Progettista
Verdi	35 000	Venere	15 000	Progettista
Neri	55 000	Venere	15 000	Direttore
Neri	55 000	Giove	15 000	Consulente
Neri	55 000	Marte	2000	Consulente
Mori	48 000	Marte	2000	Direttore
Mori	48 000	Venere	15 000	Progettista
Bianchi	48 000	Venere	15 000	Progettista
Bianchi	48 000	Giove	15 000	Direttore

- Lo stipendio di ciascun impiegato, è unico, ed è funzione del solo impiegato, indipendentemente dai progetti a cui partecipa.
- Il bilancio di ciascun progetto, è unico e dipende solo dal progetto, indipendentemente dagli impiegati che vi partecipano.

Notiamo che, nella seguente base di dati, il valore dello stipendio è ridondato ogni volta che compare un impiegato, allo stesso tempo il bilancio compare ad ogni ricorrenza del progetto.

Anomalia di aggiornamento

Oltre alla ridondanza, è da considerare il fatto che, in caso di aggiornamento dello stipendio per esempio, dovrei modificare tutte le tuple relative ad uno specifico impiegato.

Anomalia di cancellazione

La chiave della relazione sopra, è l'insieme (Impiegato, Progetto).

Se un impiegato interrompe la partecipazione a tutti i progetti, ma senza lasciare l'azienda, allora tutte le tuple relative a quell'impiegato vengono eliminate, e l'unico modo per continuare a tener traccia di quell'impiegato è ammettere un record che presenti valore nullo su parte della chiave (Progetto,Bilancio,Funzione), il che NON è ammissibile

Anomalia di inserimento

Per lo stesso problema prima esaminato, se si hanno informazioni di un nuovo impiegato, che non partecipa ad alcun progetto, sarà impossibile inserirle finché ad esso non verrà assegnato un progetto senza ammettere valori nulli su parte della chiave (inammissibile)

Dipendenze funzionali

Per lo studio sistematico delle anomalie introdotte, utilizziamo come strumento di lavoro la dipendenza funzionale.

Introduciamo con un esempio:

Nella relazione Impiegato-Progetto, abbiamo notato come lo stipendio di ciascun impiegato sia unico.

Cioè, in ogni tupla in cui è presente un determinato impiegato, il suo stipendio rimane sempre lo stesso.

Possiamo dire che il valore dell'attributo Impiegato determina il valore dell'attributo stipendio.

Stesso ragionamento, può esser fatto con Progetto e bilancio.

Formalizziamo questo concetto:

Data una relazione r su uno schema $R(X)$, siano Y, Z due sottoinsiemi di attributi non vuoti di X , c'è una dipendenza funzionale tra Y e Z se per ogni coppia di tuple aventi gli stessi valori sugli attributi Y , esse avranno stesso valore anche sugli attributi Z .

Sulla relazione in esempio, scriviamo le seguenti dipendenze funzionali:

Impiegato \rightarrow Stipendio

Progetto \rightarrow Bilancio

Osservazione: Nella relazione in esame, potremmo anche scrivere una dipendenza funzionale del tipo Impiegato, Progetto \rightarrow Progetto, a indicare che se due tuple hanno stesso valore per Impiegato e Progetto, allora avranno stesso valore sull'attributo Progetto.

Questa sopra è detta dipendenza banale, ovvio che se due tuple hanno valore su "Progetto" uguale, allora avranno stesso valore su Progetto (grazie ar cazzo)...

Faremo riferimento a sole dipendenze non banali, cioè dipendenze del tipo $Y \rightarrow Z$, in cui Z non compare tra gli attributi di Y .

Osservazione: Esiste un legame tra dipendenze funzionali e il vincolo di chiave. Infatti potremmo dire che se Y è una chiave, allora esiste una dipendenza tra Y e qualsiasi altro attributo della tabella (Y è univoca, quindi se due tuple hanno valore di Y uguale, avranno i valori di tutti gli attributi uguali, cioè sono la stessa tupla).

Forma normale di Boyce e Codd (BCNF)

Una relazione r è in forma normale di Boyce e Codd, se per ogni sua dipendenza funzionale non banale $A \rightarrow B$, A contiene una chiave (superchiave) di r .

Sulla relazione in esempio, scriviamo le seguenti dipendenze funzionali:

Impiegato \rightarrow Stipendio

Progetto \rightarrow Bilancio

Impiegato Progetto \rightarrow Funzione

Questa relazione non soddisfa la BCNF, attraverso un processo di normalizzazione, la scomponiamo in diverse tabelle, una per ogni dipendenza individuata, in cui per ognuna la chiave è il primo membro della dipendenza stessa.

Impiegato	Stipendio	Progetto	Bilancio
Rossi	20 000	Marte	2000
Verdi	35 000	Giove	15 000
Neri	55 000	Venere	15 000
Mori	48 000		
Bianchi	48 000		

Impiegato	Progetto	Funzione
Rossi	Marte	Tecnico
Verdi	Giove	Progettista
Verdi	Venere	Progettista
Neri	Venere	Direttore
Neri	Giove	Consulente
Neri	Marte	Consulente
Mori	Marte	Direttore
Mori	Venere	Progettista
Bianchi	Venere	Progettista
Bianchi	Giove	Direttore

Proprietà di una buona scomposizione

- Assenza di perdita - se ricostruiamo una relazione dalle sue parti decomposte, non dobbiamo generare n-uple non inizialmente presenti

Impiegato	Progetto	Sede
Rossi	Marte	Roma
Verdi	Giove	Milano
Verdi	Venere	Milano
Neri	Saturno	Milano
Neri	Venere	Milano

Data la seguente relazione, soddisfa le dipendenze funzionali:

Impiegato → Sede

Progetto → Sede

Una scomposizione immediata potrebbe essere:

Impiegato	Sede	Progetto	Sede
Rossi	Roma	Marte	Roma
Verdi	Milano	Giove	Milano
Neri	Milano	Saturno	Milano
		Venere	Milano

Questa soluzione però ha un problema: non permette di ricostruire la relazione originale.

Provando infatti ad agire con un Join sull'attributo Sede, produrrei questa tabella

Impiegato	Progetto	Sede
Rossi	Marte	Roma
Verdi	Giove	Milano
Verdi	Venere	Milano
Neri	Saturno	Milano
Neri	Venere	Milano
Verdi	Saturno	Milano
Neri	Giove	Milano

Verdi con Sede a Milano, sarebbe associato sia ai progetti Giove, Venere, Saturno, quando nella tabella originale egli lavorava solo per il progetto Giove e Venere.

Diciamo che r si decompone senza perdita in X_1 e X_2 se il join delle due relazioni è uguale a r stessa.

Per garantire la decomposizione senza perdita, l'insieme degli attributi comuni alle due relazioni, deve essere chiave per almeno una delle relazioni decomposte.

In questo caso, l'attributo comune Sede, non è chiave per nessuna delle due tabelle.

- Conservazione delle dipendenze - Ogni dipendenza funzionale presente nella tabella di partenza, deve essere mantenuta anche nelle relazioni che la decompongono.

Impiegato	Progetto	Sede
Rossi	Marte	Roma
Verdi	Giove	Milano
Verdi	Venere	Milano
Neri	Saturno	Milano
Neri	Venere	Milano
Verdi	Saturno	Milano
Neri	Giove	Milano

Ipotizziamo di scomporre la tabella originale nelle due relazioni:

Impiegato	Sede	Impiegato	Progetto
Rossi	Roma	Rossi	Marte
Verdi	Milano	Verdi	Giove
Neri	Milano	Verdi	Venere
		Neri	Saturno
		Neri	Venere

Impiegato, attributo in comune tra le due, è chiave della prima, pertanto la scomposizione è garantita senza perdite.

In queste due tabelle, la prima ha dipendenza Impiegato → Sede, la seconda non ne ha.

Supponiamo di voler inserire una nuova tupla "Neri,Marte,Milano": Nella tabella iniziale, questa tupla violerebbe la dipendenza Progetto → Sede, perché due tuple del progetto Marte, avrebbero due sedi diverse.

Nella scomposizione invece, io dovrei mettere nella prima tabella "Neri,Milano", il che non viola nessuna dipendenza, e nella seconda tabella "Neri,Marte", anch'essa non viola alcuna dipendenza.

Per far sì che le dipendenze vengano mantenute anche nelle scomposizioni, devo fare in modo che tutti gli attributi che fan parte di una dipendenza funzionale, si trovino in una stessa tabella.

Terza forma normale

Per ogni dipendenza funzionale non banale $A \rightarrow B$, è vera almeno una delle due

- A contiene una chiave K di r (A è superchiave)
- B appartiene ad almeno una chiave di r .

Seconda forma normale

Se una relazione ha chiave primaria atomica, è sicuramente in 2NF.

Se una relazione ha una chiave primaria composta invece, essa è in 2NF se:

- Tutti gli attributi non chiave, dipendono da tutta la chiave, e non solo da una parte.

Tabella: **Iscrizioni**

CorsolD	StudentelD	NomeStudente	NomeCorso
C1	S1	Anna	Matematica
C1	S2	Luca	Matematica
C2	S1	Anna	Fisica

Chiave primaria: (CorsolD, StudentelD)

Dipendenze funzionali:

- **StudentelD** → NomeStudente (dipendenza **parziale**)

- **CorsolD** → NomeCorso (dipendenza **parziale**)

Non è in 2NF perché alcuni attributi (NomeStudente e NomeCorso) dipendono **solo da una parte della chiave**.

Studente(**StudentID**,NomeStudente)

Corso(**CorsolD**,NomeCorso)

Iscriversi(**StudentID,CorsolD**)

StudentID → NomeStudente

CorsolD → NomeCorso

E' in seconda forma normale.

Prima forma normale

Ogni attributo della relazione, è un attributo semplice.

Progettazione fisica

Ci occupiamo di come i dati vengono memorizzati su memoria secondaria (Dischi meccanici, Dischi stato solido), allo scopo di garantire la persistenza.

In memoria secondaria, i dati sono memorizzati in file (divisi in blocchi di dimensione B), ognuno contenente una sequenza di record (di dimensione R).

Ogni record è una collezione di valori di dati (campi): i dati di tipo BLOB e CLOB vengono memorizzati a parte, e il record contiene solo un puntatore al dato.

Il DBMS, per poter lavorare sui dati, necessita di un processo di caricamento di tali dati in memoria centrale.

Nella memoria centrale, il DBMS ha a disposizione un buffer, organizzato in pagine di dimensione pari a X blocchi.

In base alla dimensione dei blocchi, e a quella dei record, un blocco generalmente memorizza più di un record (decine, centinaia).

Quando dobbiamo lavorare coi dati, il DBMS sposta dei blocchi dalla memoria secondaria ai buffer, secondo specifiche politiche di allocazione e deallocazione (Valgono i concetti visti nel corso di Sistemi Operativi, LRU, Pseudo-LRU, ecc...)

Fattore di blocco

Indica il numero di record contenuti in un blocco di memoria.

Utile a calcolare lo spazio su disco necessario per memorizzare un insieme di record.

$$bfr = \lfloor \frac{B}{R} \rfloor$$

- B è la dimensione del blocco
- R è la dimensione media del record

Vi è un elemento di variabilità in questo rapporto, che rende il fattore di blocco non sempre determinabile con precisione: ogni record può avere lunghezza fissa, oppure variabile.

Esempio:

```
CREATE TABLE clienti_fissi (
    id      INTEGER,      -- 4 byte
    nome   CHAR(30),     -- 30 byte (fisso, spazi riempiti se vuoto)
    cognome CHAR(30),    -- 30 byte
    eta    SMALLINT      -- 2 byte
);
```

La tabella sopra, ha record a dimensione fissa.

Dalla documentazione di Postgresql, conosciamo la dimensione dei diversi data types:

- `INTEGER` = 4 byte
- `CHAR(30)` = 30 byte (fisso)
- `CHAR(30)` = 30 byte
- `SMALLINT` = 2 byte

Total:

$$R=4+30+30+2=66 \text{ byte}$$

Nota: PostgreSQL può aggiungere qualche byte di **overhead per riga** (tipicamente 23–27 byte per intestazioni e allineamenti). Ma per il calcolo **semplificato del BFR**, consideriamo solo i dati grezzi.

Esempio:

```
CREATE TABLE articoli_var (
    id      SERIAL,      -- 4 byte (intero autoincrementale)
    titolo  VARCHAR(100), -- da 1 a 100 caratteri
    contenuto TEXT,      -- da pochi a migliaia di caratteri
    autore  VARCHAR(50)  -- da 1 a 50 caratteri
);
```

Non possiamo conoscere la dimensione di una stringa variabile, così come non possiamo conoscere la dimensione di un dato BLOB, e altri.

Dobbiamo stimare la dimensione di un record.

Ogni carattere in PostgreSQL in UTF-8 può valere **1 byte (ASCII)**, ma per sicurezza si considera **2 byte medi** per supporto Unicode.

Quindi:

- `id` = 4 byte
- `titolo` = $60 \times 2 = 120$ byte
- `contenuto` = $2000 \times 2 = 4000$ byte
- `autore` = $30 \times 2 = 60$ byte

Total stimato:

$$R \approx 4 + 120 + 4000 + 60 = 4184 \text{ byte}$$

Strutture primarie

Come sono disposti i record nei files su disco.

- Strutture sequenziali - i file sono costituiti da blocchi logicamente consecutivi, e i record sono inseriti in essi sequenzialmente

Criteri di memorizzazione:

- Non ordinata - Record memorizzati in ordine di inserimento (Caso migliore: il record da cercare è il primo, Caso peggiore: il record da cercare è l'ultimo, Caso medio: il record da cercare sta in mezzo.)
- Ordinata - Record memorizzati ordinando sul valore di un campo chiave: Inserimento e cancellazione lenti, ricerca veloce.
- Memorizzazione sequenziale ad array - Possibile solo con record a lunghezza fissa, ogni record ha un indice I che determina la sua posizione nel file.

L'identificazione avviene tramite indice in tempo costante.

- Strutture ad accesso calcolato - La posizione di una record nel file, è determinata tramite una funzione di hash calcolata sul valore di un campo chiave.

Applicabile solo con record a lunghezza fissa. Ricerca efficiente.

Indici

Un indice è una struttura d'accesso secondaria progettata per rendere più efficiente la ricerca dei dati all'interno di un file, affiancandosi alla struttura primaria di memorizzazione

Funziona come l'indice analitico di un libro: elenca determinati valori (le chiavi) e punta direttamente ai dati corrispondenti.

Ogni voce di un indice contiene:

- Una **chiave** (il valore da cercare)
- Un **puntatore** al **blocco** o **record** nel file dati

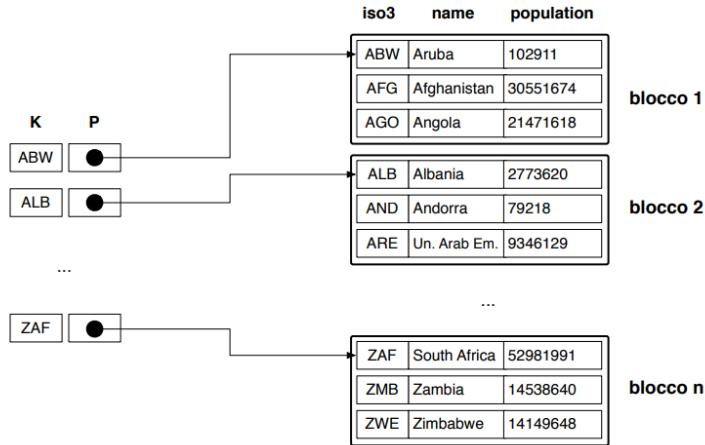
Indici primari

Definiti su file in cui i record sono ordinati rispetto ad un attributo chiave (detto campo di ordinamento).

E' a sua volta un file ordinato, costruito al fine di indirizzare i blocchi di un altro file ordinato. L'indicizzazione avviene associando ad ogni blocco del file, due informazioni:

- Chiave del primo record del blocco
- Puntatore al blocco di memoria stesso

Una voce i dell'indice, è perciò della forma $\langle K, P \rangle$ (Key,Pointer), e rappresenta un intero blocco di memoria: si dice che l'indice primario è sparso, cioè contiene una voce per ciascun blocco del file dati, e non per ogni record.



Indici secondari

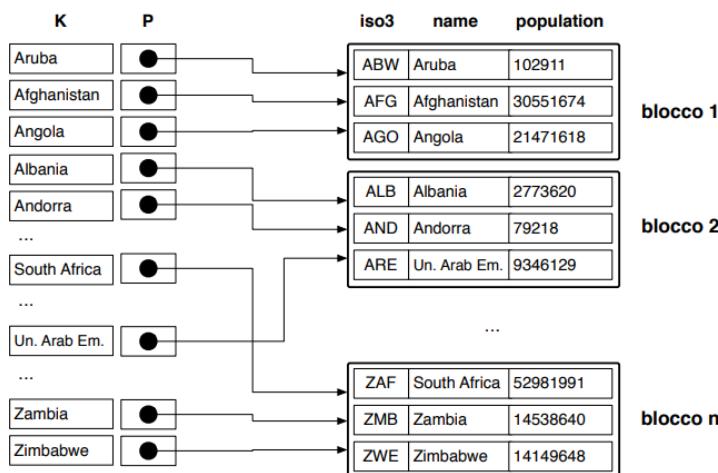
Un indice secondario è una struttura d'accesso che può essere costruita su **qualsiasi attributo, anche se non chiave**, anche se il file non è ordinato rispetto a tale attributo. È quindi applicabile a file **non ordinati, ordinati su altri campi o memorizzati con hash**.

Se il campo di indicizzazione è chiave, allora l'indice secondario conterrà una voce per ogni chiave, associata ad un puntatore ad un singolo record.

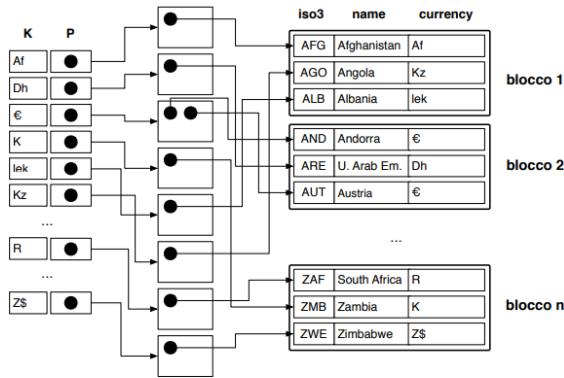
Se invece il campo di indicizzazione non è chiave, vi potrebbero essere più record con lo stesso valore indicizzato. In tal caso, ogni voce dell'indice punterà ad un insieme di record.

In entrambi casi, un indice secondario è sempre denso.

Poiché l'indice è ordinato rispetto all'attributo indicizzato, è possibile effettuare **ricerche binarie sull'indice stesso**, anche se il file non lo consente direttamente. Si riduce così drasticamente il numero di accessi rispetto a una ricerca lineare.



Indici secondari su campi non chiave



Esercizi in classe:

Indici primari:

- Record di dimensione fissa R = 100 byte
- File con r = 30.000 record e blocchi B = 1024 byte
- Quante voci conterrà l'indice?

Ogni blocco è grosso 1024 byte, ogni record è grande 100 byte, in ogni blocco ci stanno $1024/100 = 10$ record.

Per memorizzare 30000 record, ho bisogno di 3000 blocchi.

Un indice primario per blocco = 3000 indici.

Sicurezza

Garantire la sicurezza di una base di dati, vuol dire raggiungere i seguenti obiettivi:

- Segretezza - Proteggere i dati da letture non autorizzate
- Integrità - Proteggere i dati da modifiche o cancellazioni non autorizzate
- Disponibilità - Garantire che non si verifichino casi in cui utenti legittimi non possano accedere ai dati.

Tecniche per il raggiungimento di tali obiettivi:

- Autenticazione - Verifica dell'identità dell'utente
- Controllo dell'accesso - Per ogni richiesta di accesso ai dati, verificare che l'utente sia autorizzato.
- Crittografia - Cifratura dei dati, per far sì che possano essere decifrati solo da utenti autorizzati.

Ci concentreremo in particolare sulla seconda tecnica: le altre due sono da considerarsi extra-DBMS.

Controllo accesso

Vogliamo limitare e controllare le operazioni che ogni agente (utenti o programmi in esecuzione, inclusi gli utenti di default) può effettuare, mediante il meccanismo dei privilegi.

Il meccanismo che ha il compito di controllare se l'utente è autorizzato a compiere l'accesso ai dati, è il **Reference Monitor**.

Politiche per l'amministrazione della sicurezza

Chi concede e revoca i diritti?

- Politica centralizzata: unico soggetto detto DBA controlla l'intera base di dati.
(L'utente postgres)
- Politica decentralizzata: più soggetti si occupano di porzioni diverse del DB.
- Ownership: i permessi su di un oggetto sono gestiti dall'utente che l'ha creato.

Politiche per il controllo dell'accesso

In che modo i soggetti possono accedere ai dati?

- Need-To-Know(minimo privilegio): permette all'utente l'accesso solo ai dati strettamente necessari per eseguire le proprie attività.
Offre ottime garanzie di sicurezza, adatta a basi di dati con forti esigenze di protezione.
- E' necessario che il sistema non sia eccessivamente protetto, negando anche gli accessi che non comprometterebbero la sicurezza del sistema.
 - Sistema chiuso: accesso permesso solo se esplicitamente autorizzato.
- Maximized Sharing(massima condivisione): consente agli utenti il massimo accesso all'informazioni della base di dati, mantenendo comunque le info riservate.
Adatta ad ambienti in cui non è necessario un alto livello di protezione.
 - Sistema aperto: accesso permesso a meno che non sia esplicitamente negato.

Politiche Discrezionali (DAC - Discretionary Access Control)

Le politiche discrezionali si basano sul controllo dell'acceso gestito dal proprietario dell'oggetto.

In altre parole, ogni utente proprietario ha la possibilità di decidere, quali utenti e con quali permessi accedono agli oggetti di cui esso è proprietario.

Meccanismo di Controllo

Ad ogni oggetto è associato un set di regole che specifica quali soggetti possono compiere quali azioni.

Ad esempio:

- Un file potrebbe essere accessibile in lettura a un gruppo di utenti, ma solo un altro utente potrebbe avere il diritto di modificarlo.

Vantaggi

- **Flessibilità:** i proprietari decidono autonomamente chi può fare cosa sui propri oggetti.
- **Adattabilità a contesti variabili:** Queste politiche sono particolarmente adatte a contesti in cui gli utenti hanno frequente necessità di estendere o restringere l'accesso agli oggetti.

Svantaggi

- **Mancanza di controllo sul flusso di informazioni:** Una volta che un utente ha accesso a un oggetto, non ci sono restrizioni su come quel dato possa essere usato, elaborato o trasmesso ad altri. Per esempio, un utente potrebbe leggere una risorsa sensibile, e copiarne il contenuto all'interno di un'altro file, a cui hanno accesso utenti non autorizzati sul file originario (ho scalacato i permessi)
- **Rischio di abusi o errori:** c'è il rischio che i dati vengano accidentalmente esposti o condivisi senza il permesso adeguato. Questo rende il sistema vulnerabile ad attacchi interni o esterni.

Politiche Mandatorie (MAC - Mandatory Access Control)

Le politiche mandatorie, al contrario, non sono basate sulla discrezione dei singoli utenti, ma su **regole predefinite e imposte da un'autorità centrale**. Queste politiche sono particolarmente utilizzate in contesti dove la sicurezza è critica, come nel caso di sistemi governativi, militari, e organizzazioni che gestiscono dati sensibili e riservati.

Gli oggetti e gli utenti sono associati a **etichettature di sicurezza**, come livelli di classificazione (es: Pubblico, Confidenziale, Segreto, Top Secret)

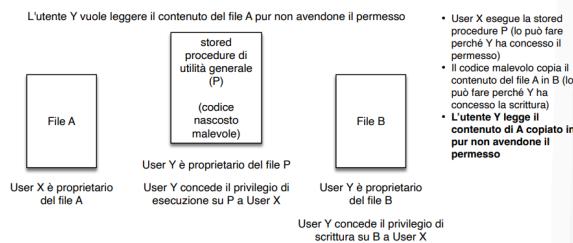
Un utente può leggere un oggetto solo se possiede un livello di autorizzazione **uguale o superiore** a quello dell'oggetto, e scrivere solo se il livello dell'utente è minore o uguale a quello dell'oggetto (evitare fuga di dati sensibili, cioè che utenti con privilegi superiori, scrivano su files accessibili da utenti con privilegi inferiori)

Vantaggi

- **Elevata sicurezza.**
- **Impossibilità di modificare i diritti di accesso:** Gli utenti non possono aggirare o modificare le regole di accesso, quindi non c'è rischio che un utente conceda accesso a un oggetto a chi non ha le autorizzazioni adeguate.
- **Controllo sul flusso di dati:** limita il movimento dei dati da oggetti più protetti a oggetti meno protetti, evitando la fuga di informazioni sensibili.

Svantaggi

- **Meno flessibilità**
- **Complesso da gestire:** etichette e regole d'accesso vanno studiate precisamente
- **Difficile adattabilità a contesti dinamici, data la rigidità imposta.**



System R

Il modello implementa una politica di tipo discrezionale e supporta il controllo dell'accesso in base sia al nome che al contenuto.

- Il sistema è un sistema chiuso: un accesso **e concesso solo se esiste una esplicita regola che lo autorizza.**
- L'amministrazione dei privilegi è decentralizzata mediante ownership: quando un utente crea **tutti i diritti di accesso su di essa ed anche la possibilità** di delegare ad altri tali privilegi.

Comando GRANT/REVOKE

```
GRANT lista Privilegi | ALL [PRIVILEGES] ON  
Lista Relazioni | Lista Viste TO Lista Utenti | PUBLIC [WITH GRANT OPTION]
```

L'Opzione GRANT, permette la delega dei privilegi.

L'utente che riceve dei privilegi, se essi sono stati concessi con grant option, avrà anche la possibilità di concederli ad altri.

Da questa, suddividiamo i privilegi di un utente in:

- Delegabili
- Non delegabili

La keyword PUBLIC, permette una concessione di privilegi a tutti gli utenti.

Esempio:

```
GRANT UPDATE(Stipendio, PremioP) ON Impiegato TO Rossi;
```

Rossi può modificare gli attributi Stipendio e PremioP delle tuple della relazione Impiegato.

```
GRANT SELECT, INSERT ON Impiegato TO Verdi, Gialli;
```

Verdi e Gialli possono selezionare ed inserire tuple nella relazione Impiegato

```
GRANT ALL PRIVILEGES ON Impiegato TO Neri WITH GRANT OPTION;
```

Neri ha tutti i privilegi sulla relazione Impiegato e può delegare ad altri tali privilegi

Revocare i privilegi:

```
REVOKE lista Privilegi | ALL [PRIVILEGES] ON  
Lista Relazioni | Lista Viste FROM Lista Utenti | PUBLIC
```

Un utente può revocare solo privilegi che lui stesso ha concesso.

Quando si esegue un'operazione di revoca, l'utente a cui i privilegi vengono revocati perde tali privilegi, a meno che essi non provengono anche da sorgenti diversi da quella che effettua la revoca.

SYSAUTH e SYSCOLAUTH

Le regole di autorizzazione specificate dagli utenti sono memorizzate in due cataloghi di sistema di nome sysauth e syscolauth, implementati come relazioni.

SYSAUTH

id_utente	nome	creator	T	D	I	S	U	GO
bianchi	impiegato	bianchi	R	Y	Y	Y	all	Y
verdi	impiegato	bianchi	R	N	Y	Y	N	Y
rossi	impiegato	bianchi	R	N	N	Y	N	Y
rossi	impiegato	bianchi	R	N	Y	Y	N	N

T: type (Relation o View)
D: privilegio DELETE
I: privilegio INSERT
S: privilegio SELECT
U: privilegio di update sulle colonne
GO: l'utente possiede la Grant Option?

Una tupla di sysauth ha i seguenti attributi:

- id utente: id dell'utente a cui sono concessi i privilegi;
- nome: nome della relazione/vista su cui sono concessi i privilegi;
- creatore: utente che ha creato la relazione;
- tipo ∈ {R, V}: indica se l'oggetto è una relazione (tipo ='R') o una vista (tipo ='V');
- P ∈ {Y, N}: indica se l'oggetto ha (Y) o meno (N) il privilegio sulla relazione.

Sysauth contiene un attributo per ciascuno dei privilegi

- update ∈ {ALL, SOME, N}: indica se il soggetto ha il privilegio di update su tutte (ALL) alcune (SOME), o nessuna (N) colonna della relazione
- grantopt ∈ {Y, N}: indica se i privilegi sono delegabili (Y) o meno (N)

SYSCOLAUTH

id_utente	nome	colonna	GO
bianchi	impiegato	imp	Y
bianchi	impiegato	mansione	Y
...

Il catalogo SYSCOLAUTH è relativo
al solo privilegio UPDATE

Quando un utente esegue un comando di GRANT, il meccanismo di controllo accede alle tabelle SYSAUTH o SYSCOLAUTH, per determinare se tale utente ha il diritto di delegare i privilegi specificati nel comando.

Il DBMS esegue un'operazione di intersezione tra i privilegi delegabili dell'utente, e quelli specificati nel comando di GRANT: se l'intersezione è vuota, il comando non viene eseguita, se coincide con i privilegi specificati nel comando, vengono concessi tutti i privilegi specificati, altrimenti il comando viene eseguito parzialmente, solo sui privilegi contenuti all'interno dell'intersezione.

Revoca ricorsiva

Durante un'operazione di revoca di un permesso, **se l'utente a cui revoco un permesso non ha concesso quel permesso ad altri** (cioè non ha usato `WITH GRANT OPTION` per concederlo), allora un semplice `REVOKE` basta: il permesso viene tolto solo a lui.

Se invece l'utente ha concesso quel permesso ad altri (Tramite `GRANT OPTION`), l'operazione di revoca fallisce.

I Permessi concessi da lui a terzi si chiamano **permessi dipendenti**: per rimuoverli si utilizza la keyword `CASCADE` associata a `REVOKE`, a indicare un'operazione di revoca ricorsiva.

Si percorre il grafo delle concessioni, togliendo il permesso a tutti coloro che lo hanno ottenuto solo tramite l'utente originale.

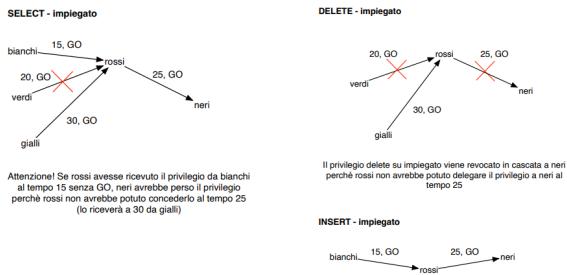
- Il sistema mantiene internamente un **grafo delle concessioni** (chi ha concesso cosa a chi).
- Quando fai `REVOKE ... CASCADE`, si segue questo grafo partendo dall'utente da cui togli il permesso, e si rimuovono tutti i permessi concessi **discendentemente**.
- Se un utente ha ricevuto lo stesso permesso anche da altre vie (da altri utenti o tramite il gruppo PUBLIC), quel permesso può rimanere valido.

Esempio:

id_utente	nome	grantor	T	I	S	D	GO
rossi	impiegato	bianchi	R	15	15	0	Y
rossi	impiegato	verdi	R	0	20	20	Y
neri	impiegato	rossi	R	25	25	25	Y
rossi	impiegato	gialli	R	0	30	30	Y

La tabella è modificata per mantenere, per ogni privilegio, l'utente che ha concetto il privilegio (grantor), e il timestamp che denota il tempo in cui il privilegio è stato concesso (al posto di un semplice `Y` o `N`).

Se al tempo 35, l'utente verdi eseguisse il comando `REVOKE ALL on impiegato from rossi`



Transazioni

Una **transazione** è un **insieme di operazioni SQL** che vengono eseguite come un **blocco unico e indivisibile, senza interruzioni**.

Al termine di una transazione, possiamo giungere a due possibili esiti:

- Tutte le operazioni sono andate a buon fine, la transazione è approvata. (COMMIT)
- Alcune operazioni falliscono: Nessuna modifica viene applicata al database. (ROLLBACK)

Esempio:

Identificare sequenze di operazioni che vanno concettualmente considerate come uniche e indivisibili.

Trasferimento di denaro dal conto bancario A al conto B.

```
UPDATE Conti SET Saldo = Saldo - 100 WHERE ID = 'A';
UPDATE Conti SET Saldo = Saldo + 100 WHERE ID = 'B';
```

Se il sistema dovesse crashare dopo l'esecuzione della prima istruzione, A perderebbe soldi, B non li riceverebbe: i soldi verrebbero persi.

La soluzione, è garantire che queste operazioni vengano eseguite in modo indivisibile, o entrambe o nulla.

```
START TRANSACTION;

UPDATE Conti SET Saldo = Saldo - 100 WHERE ID = 'A';
UPDATE Conti SET Saldo = Saldo + 100 WHERE ID = 'B';

COMMIT;
```

Problemi di concorrenza

Le operazioni di un DBMS sono eseguite tramite le primitive di read dei dati da memoria secondaria a memoria principale, e di write da memoria principale a memoria secondaria.

Quando più utenti o applicazioni lavorano in modo concorrente sui dati, i tempi di RW possono portare ad alcune anomalie:

- Lost Update - Due utenti U1 e U2, leggono il medesimo dato X, ed eseguono operazioni di aggiornamento.
U2 legge il dato X prima che U1 scriva la sua modifica, pertanto la modifica di U1 viene persa.
 1. `U1: X := read(X) + 100`
 2. `U2: X := read(X) + 200`
 3. `U1: write(X)`
 4. `U2: write(X)`
- Dirty Read - U1 aggiorna un dato X, e successivamente fallisce. U2 legge il dato X prima che venga riportato al suo valore originale, pertanto leggerà il valore aggiornato da U1.
 1. `U1: X := read(X) + 100`
 2. `U1: write(X)`
 3. `U2: X := read(X) + 200`
 4. `U2: write(X)`
 5. `U1: Abort`

Poichè U2 legge X prima che U1 fallisca, il dato letto da U2 è potenzialmente errato.

- Incorrect Summary - Se un client calcola una funzione aggregata su un insieme di record, mentre un altro client effettua un aggiornamento su tali record, il primo client calcolerà la funzione aggregata su valori non aggiornati (o parzialmente aggiornati).
 1. `U2: sum := 0`
 2. `U1: X := read(X) - 200`
 3. `U1: write(X)`
 4. `U2: sum := sum + read(X)`
 5. `U2: sum := sum + read(Y)`
 6. `U1: Y := read(Y) + 200`
 7. `U1: write(Y)`

La somma non sarà coerente coi valori finali di X e Y, poichè leggo X dopo l'aggiornamento, ma Y prima dell'aggiornamento.

Proprietà delle transazioni

Il DBMS esegue le transazioni secondo le proprietà ACID:

- Atomicity - Se si verifica un errore durante l'esecuzione di una transazione prima del COMMIT, gli effetti parziali della transazione non vengono memorizzati. (ROLLBACK).
Il rollback può essere eseguito automaticamente in caso di errore, o invocato esplicitamente.
- Consistency - Ogni transazione, può assumere di lavorare in un DB in cui sono verificati tutti i vincoli di integrità. Deve inoltre lasciare un DB che soddisfa tutti i vincoli di integrità.

Ai fini delle transazioni, i vincoli possono essere specificati come:

- IMMEDIATE - La verifica del vincolo avviene durante l'esecuzione della transazione. (Se qualche vincolo viene violato, viene cancellata la singola istruzione che porta alla violazione)
- DEFERRED - Verifica al termine della transazione, dopo il commit (Se qualche vincolo viene violato, ROLLBACK dell'intera sequenza di istruzioni)

Esempio:

```

CREATE TABLE Studente (
    Matricola INT PRIMARY KEY,
    Nome VARCHAR(50),
    CorsoLaurea VARCHAR(50),
    Tutor INT,
    FOREIGN KEY (Tutor) REFERENCES Studente(Matricola)
        DEFERRABLE INITIALLY DEFERRED
);

```

- Ogni studente può avere un **tutor** che è **anch'esso uno studente**.
- Il vincolo di **chiave esterna su** **Tutor** è definito come **DEFERRABLE INITIALLY DEFERRED**, cioè **verrà controllato al momento del COMMIT**, non subito.

Inserire **due studenti**, uno dei quali è tutor dell'altro, ma **nessuno dei due è ancora presente nella tabella al momento dell'inserimento**.

Se i vincoli fossero **IMMEDIATE**, una delle due **INSERT** fallirebbe subito.

```

BEGIN;

-- Inserisco lo studente A (Mario), il cui tutor è B (che non esiste ancora)
INSERT INTO Studente (Matricola, Nome, CorsoLaurea, Tutor)
VALUES (1, 'Mario Rossi', 'Informatica', 2);

-- Inserisco lo studente B (Luigi), il cui tutor è A
INSERT INTO Studente (Matricola, Nome, CorsoLaurea, Tutor)
VALUES (2, 'Luigi Verdi', 'Matematica', 1);

COMMIT;

```

Funziona perché:

- Il vincolo è **DEFERRED**: viene controllato **solo al COMMIT**.
- Al momento del commit, **entrambe le righe esistono**, e i riferimenti ai tutor sono validi.

Modifichiamo il vincolo come:

```
FOREIGN KEY (Tutor) REFERENCES Studente(Matricola)
```

Allora, se esegui:

```

BEGIN;

-- Mario ha come tutor Luigi (non esiste ancora)
INSERT INTO Studente (Matricola, Nome, CorsoLaurea, Tutor)
VALUES (1, 'Mario Rossi', 'Informatica', 2); -- ⚡ ERRORE qui

-- Non si arriva nemmeno a questa istruzione
INSERT INTO Studente (Matricola, Nome, CorsoLaurea, Tutor)
VALUES (2, 'Luigi Verdi', 'Matematica', 1);

COMMIT;

```

- **Errore immediato:** il DB rifiuta la prima `INSERT` perché non esiste ancora un `Matricola = 2`.
 - Se il vincolo è `IMMEDIATE`, viene verificato **subito** dopo ogni istruzione.
 - Isolation - Durante le operazioni di RW da parte di più utenti su di una stessa tabella contemporaneamente, si può garantire che le loro transazioni siano eseguite senza interferire l'una con l'altra.
 - Durability - Se si verifica un crash dopo che una transazione ha eseguito COMMIT, gli effetti della transazione non vengono persi.
-

Sistemi NoSQL

Il termine **NoSQL** nasce nel 1998 per indicare un database leggero **senza interfaccia SQL** (Carlo Strozzi).

Dal 2009, NoSQL significa **Not Only SQL** e si riferisce a:

- **Database non relazionali**
- **Distribuiti**
- **Non vincolati dalle proprietà ACID**

"NoSQL sono database di nuova generazione, generalmente non relazionali, distribuiti, open source e scalabili orizzontalmente".

Caratteristiche principali dei DB NoSQL

1. Scalabilità orizzontale

- È possibile aumentare la capacità del sistema **aggiungendo nuovi nodi**, senza fermarlo.
- Ideale per grandi volumi di dati.

2. Replica e disponibilità

- I dati vengono **replicati automaticamente** su più nodi:
 - Maggiore **velocità di lettura**
 - Scritture più complesse (devono aggiornare tutte le copie)

3. Consistenza eventuale

- I dati **non sono subito consistenti** su tutti i nodi.
- Dopo un po', **tutti i nodi convergono sullo stesso stato**.

Se l'utente A sa che domani piove e lo dice a B, poi B lo dice a C... alla fine tutti sapranno la verità, ma non nello stesso momento.

4. Frammentazione orizzontale (sharding)

- I dati sono **suddivisi tra diversi nodi**, ciascuno con una porzione (shard).
- Insieme a tecniche di **replica**, migliora le prestazioni e la tolleranza ai guasti.

5. Assenza di schema rigido

- I dati sono spesso in formato **JSON** o simili.
 - Nessun vincolo formale di schema: la struttura è **flessibile**.
 - Le **regole di integrità** devono essere gestite **nel codice dell'applicazione**, non dal DB.
-

Tipologie di database NoSQL

I DB NoSQL si classificano in 4 categorie principali:

Categoria	Esempi	Descrizione breve
Key-Value Store	DynamoDB, Redis, Berkeley DB	Ogni dato ha una chiave e un valore associato
Document Store	MongoDB, CouchDB	I dati sono documenti (es. JSON), organizzati in collezioni
Column-Family Store	BigTable, SimpleDB	Dati organizzati per colonne, adatti a query analitiche
Graph DB	Neo4j, Sones	I dati sono nodi e relazioni, utili per reti sociali, raccomandazioni

Document-Based Databases

- Concetti chiave: **documento** e **collezione**
- Un **documento** è:
 - Autodescrittivo (include sia struttura che dati)
 - Gerarchico (può contenere dati annidati)
 - In formato flessibile (come JSON)
- Una **collezione**:
 - Raggruppa documenti **simili**, ma non identici
 - Non richiede uno schema fisso

Esempio MongoDB:

```
{  
  "nome": "Alice",  
  "eta": 30,  
  "interessi": ["lettura", "viaggi"],  
  "indirizzo": {  
    "citta": "Roma",  
    "CAP": "00100"  
  }  
}
```

Esercizi Algebra Relazionale (π , σ , \bowtie , ρ)

Schema "Sentenze":

PERSONA(CF, Nome, Cognome, DataNascita, CittaNascita, CittResidenza)
CONDANNA(CFPersona, CFGiudice, Data, TipoReato, TipoCondanna, Durata)
GIUDICE(CF, Nome, Cognome, Tribunale, AnnoIngressoMagistratura)

Interrogazione 1:

Determinare il nome e cognome delle persone condannate nel 2002 all'ergastolo per uxoricidio e che hanno subito almeno una condanna per furto.

Determino le persone condannate nel 2002 all'ergastolo per uxoricidio, ed interseco con l'insieme delle persone che hanno subito condanne per furto

R1 := $\pi_{CFPersona} (\sigma_{Data \geq '2002-01-01' \wedge Data \leq '2002-12-31'} \wedge TipoCondanna = 'Ergastolo' \wedge TipoReato = 'Uxoricidio')$ (CONDANNA))

R2 := $\pi_{CFPersona} (\sigma_{TipoReato = 'Furto'})$ (CONDANNA))

RIS := $\pi_{Nome, Cognome} ((R1 \cap R2) \bowtie_{CF=CFPersona} PERSONA)$

Ricordarsi che un intersezione tra valori che non sono chiave E' SBAGLIATA! Altrimenti potrei avere falsi positivi.

Esempio soluzione SBAGLIATA:

R1: $\pi_{Nome, Cognome} (\sigma_{Data=2002} \wedge TipoCondanna='Ergastolo' \wedge TipoReato='Uxoricidio')$ (PERSONA $\bowtie_{CF=CFPersona}$ CONDANNA))

R2: $\pi_{Nome, Cognome} (\sigma_{TipoReato='Furto'})$ (PERSONA $\bowtie_{CF=CFPersona}$ CONDANNA))

Soluzione: $R1 \cap R2$

In R1 e R2 proietti direttamente Nome e Cognome:

Questo ti impedisce di intersecare i codici fiscali, che sono l'identificativo univoco.

Se una persona ha lo stesso nome e cognome di un'altra, potresti ottenere falsi positivi.

Interrogazione 2:

Determinare, per ciascun giudice del tribunale di Milano, la massima durata delle condanne che ha emesso

R1: $\pi_{CFGiudice, Durata} (\sigma_{Tribunale='Milano'})$ [GIUDICE $\bowtie_{(CF=CFGiudice)}$ CONDANNA])

A questo punto si individuano tutte le coppie di giudici con la durata di una loro condanna e si selezionano le coppie che fanno riferimento allo stesso giudice e per cui la durata del primo elemento nella coppia è inferiore a quella del secondo.

Si ottiene quindi, per ciascun giudice, l'elenco delle durate non massime. Attraverso un'operazione di sottrazione si ottiene quindi la durata massima di condanna per ciascun giudice del tribunale di Milano.

R2: $\rho_{c,d} \leftarrow CFGiudice, Durata (R1)$

R3: $R1 \bowtie_{(CFgiudice=c \text{ AND } Durata < d)} R2$

Risultato: R1 - R3.

Trovare il massimo/minimo di un attributo per gruppo, senza usare aggregazioni:

1. Definire la relazione di partenza

R1 := $\pi_{\text{GruppoAttr}, \text{ValoreAttr}} (\sigma \text{ condizioni} (\text{Relazione}))$

- Seleziona la relazione con solo **gli attributi rilevanti**:

- **GruppoAttr** = attributo su cui raggruppare (es. **CFGgiudice**)
- **ValoreAttr** = attributo su cui cercare il max/min (es. **Durata**)

2. Rinomina R1 per confronto

R2 := $\rho_g, v \leftarrow \text{GruppoAttr}, \text{ValoreAttr} (R1)$

- Serve per confrontare ciascuna tupla con le altre **dello stesso gruppo**.

3. Join per trovare le tuple "battute"

R3 := R1 \bowtie (GruppoAttr = g AND ValoreAttr < v) R2 -- per il massimo

R3 := R1 \bowtie (GruppoAttr = g AND ValoreAttr > v) R2 -- per il minimo

- Se vuoi il **massimo**, cerca le tuple con un valore **minore** di un'altra.
- Se vuoi il **minimo**, cerca le tuple con un valore **maggior** di un'altra.
- Ottieni così tutte le **non-massime** (o non-minime).

4. Sottrazione per isolare il massimo/minimo

Risultato := R1 - R3

- Togli le tuple "battute" → restano solo quelle **massime/minime** per gruppo.

Schema "Missioni":

MISSIONE(Codice, Città, DataPartenza, Scopo, DurataPrevista)
AGENTE(Codice, Nome, Cognome, Specializzazione)
PARTECIPA(CodiceMissione, CodiceAgente, Ruolo)

Interrogazione 1:

| Determinare il codice delle missioni che hanno la minima durata prevista.

R1: $\pi_{\text{Codice}, \text{DurataPrevista}} (\text{MISSIONE})$

R2: $\rho_{\text{c}, \text{d}} \leftarrow \text{Codice}, \text{DurataPrevista}(R1)$

R3: $R1 \bowtie_{(\text{d} > \text{DurataPrevista})} R2$

Risultato: R1-R3

Interrogazione 2:

| Determinare il codice degli agenti che hanno partecipato con lo stesso ruolo ad almeno due missioni iniziate nell'anno 2002.

R1: $\pi \text{CodiceMissione}, \text{CodiceAgente}, \text{Ruolo} (\sigma \text{DataPartenza} \geq 01-01-2002 \text{ AND } \text{DataPartenza} < 01-01-2003 \text{ (PARTECIPA } \bowtie \text{ (CodiceMissione=Codice) MISSIONE)})$

R2: $\rho c, ca, r \leftarrow \text{CodiceMissione}, \text{CodiceAgente}, \text{Ruolo} (R1)$

R3: $R1 \bowtie (\text{CodiceAgente}=ca \text{ AND } \text{Ruolo}=r \text{ AND } \text{CodiceMissione} \neq c) R2$

Risultato: $\pi \text{CodiceAgente}(R3)$

Schema "Supermercato":

`SUPERMERCATO(Codice, Nome, Telefono, Via, Citta, CAP)`

`ACQUISTO(CodiceSupermercato, CodiceProdotto, CodiceTransazione)`

`PRODOTTO(Codice, Nome, Costo, CodiceScaffaleSupermercato)`

`TRANSAZIONE(Codice, Data, CodiceCliente, NomeCliente, CittaResidenzaCliente)`

Interrogazione 1:

| Determinare il nome e l'indirizzo dei supermercati presso i quali sono state acquistate delle patate sia da clienti residenti a 'Milano' sia da clienti residenti a 'Crema'.

R1: $\text{ACQUISTO } \bowtie (\text{CodiceProdotto}=\text{Codice}) \text{ PRODOTTO}$

R2: $R1 \bowtie (\text{CodiceTransazione}=\text{Codice}) \text{ TRANSAZIONE}$

R3: $\pi \text{CodiceSupermercato} (\sigma \text{Nome}=\text{"Patate"} \text{ AND } \text{CittaResidenzaCliente}=\text{"Milano"} (R2))$

R4: $\pi \text{CodiceSupermercato} (\sigma \text{Nome}=\text{"Patate"} \text{ AND } \text{CittaResidenzaCliente}=\text{"Crema"} (R2))$

R5: $R3 \cap R4$

Risultato: $\pi \text{Via,Citta,CAP} (R5 \bowtie (\text{CodiceSupermercato}=\text{Codice}) \text{ SUPERMERCATO})$

Interrogazione 2:

| Determinare, per ogni transazione, il nome del prodotto più costoso.

R1: $\pi \text{CodiceTransazione, CodiceProdotto}(\text{ACQUISTO})$

R2: $\pi \text{CodiceTransazione, CodiceProdotto, Costo, Nome} (R1 \bowtie (\text{CodiceProdotto}=\text{Codice}) \text{ PRODOTTO})$

R3: $\rho CT, CP, N, C \leftarrow \text{CodiceTransazione, CodiceProdotto, Nome, Costo} (R3)$

R4: $R2 \bowtie (\text{CodiceTransazione}=CT \text{ AND } \text{Costo} < C) R3$

Risultato: $R2 - R4$

Schema "Arredamento":

`MOBILE(Codice, Linea, Dimensione, Colore, Costo)`

`CLIENTE(CF, Nome, Cognome, CittaResidenza)`

`ORDINE(CFCliente, CodiceMobile, DataOrdine, DataConsegna, ModalitaTrasporto)`

Interrogazione 1:

| Determinare, per ciascuna linea, il mobile più costoso

R1 := $\pi \text{Codice, Linea, Costo} (\text{MOBILE})$

R2 := $\rho C, L, CS \leftarrow \text{Codice, Linea, Costo} (R1)$

R3 := $R1 \bowtie \text{Linea} = L \text{ AND } \text{Costo} < CS (R2)$

RISULTATO := $R1 - R3$

Interrogazione 2:

Determinare il codice fiscale dei clienti che hanno effettuato solo ordini con modalit`a di trasporto a 'carico del cliente'.

R1: π CFCliente(σ ModalitaTrasporto ≠ "carico del cliente" (ORDINE))

Risultato: π CFCliente(ORDINE) - R1

Schema "Gommista":

PNEUMATICO(Codice, Materiale, Dimensione, CodiceProduttore)

PRODUTTORE(Codice, Nome, Via, Citta)

ACQUISTO(CFCliente, CodicePneumatico, DataVendita, Costo)

CLIENTE(CF, Nome, Telefono, Via, Citta)

Interrogazione 1:

Determinare il codice fiscale dei clienti che nel 2003 hanno acquistato almeno due volte lo stesso pneumatico

R1: π CFCliente,CodicePneumatico,DataVendita(σ DataVendita ≥ "01-01-2003" AND DataVendita < "01-01-2004" ACQUISTO)

R2: ρ CF,CP,DV \leftarrow CFCliente,CodicePneumatico,DataVendita(R1)

R3: R1 \bowtie (CFCliente=CF AND CodicePneumatico=CP AND DataVendita≠DV)R2

Risultato π CFCliente (R3)

Interrogazione 2:

Determinare il codice dei produttori che producono solo pneumatici realizzati con 'gomma secca' oppure con 'gomma sintetica'.

π , σ , \bowtie , ρ

R1: π CodiceProduttore (σ Materiale != "Gomma secca" AND Materiale != "Gomma sintetica" (PNEUMATICO))

Risultato: π CodiceProduttore(PNEUMATICO) - R1

Schema "Calendari":

CALENDARIO(Codice, Prezzo, Tipo, AnnoCalendario, CodiceProduttore)

PRODUTTORE(Codice, Nome, Indirizzo)

FOTOGRAFIA(Numerico, NomeFoto, MeseDelCalendario, CodiceCalendario)

Interrogazione 1:

Determinare, per ogni produttore, il codice del calendario di tipo 'da tavolo' di prezzo maggiore.

Interrogazione 2:

Determinare il codice dei calendari del 2003 che contengono solo fotografie scattate dal fotografo 'Sferrante'.

Schema "Biscotti":

BISCOTTO(Codice, Nome, Forma, NomeCreatore, NomeDittaProduttrice)

INGREDIENTE(Codice, Nome, Tipologia)

COMPOSIZIONE(CodiceBiscotto, CodiceIngrediente, Quantità)

Interrogazione 1:

Determinare il codice dei biscotti che contengono lo 0,3% di 'farina di riso' e lo 0,3% di 'zucchero integrale di canna'.

Interrogazione 2:

Determinare il nome dei biscotti di forma 'circolare' creati da 'MangiaTutti' e che contengono almeno l'1,6% di 'cioccolato in gocce'.

Schema "Rimborso":

FONDO(Codice, Importo, DataInizioErogazione, Scadenza, MatricolaTitolare)

DIPENDENTE(Matricola, Nome, Cognome, Posizione)

PARTECIPA(MatDipendente, CodiceFondo)

RIMBORSOSPESA(MatDipendente, CodiceFondo, Data, Importo, Motivazione)

Interrogazione 1:

Determinare, per ciascun fondo, il nome e cognome dei dipendenti che hanno chiesto l'ultimo rimborso spesa.

Interrogazione 2:

Determinare la matricola dei dipendenti che hanno chiesto almeno un rimborso spesa superiore a 3000 Euro e che non hanno mai chiesto un rimborso con motivazione 'viaggio aereo'.

Schema "Conferenze":

PERSONA(CF, Nome, Cognome, Affiliazione, Indirizzo, CittàResidenza)

CONFERENZA(Nome, Data, Indirizzo, Città)

ISCRIZIONE(NomeConferenza, DataConferenza, CFPersona, Data, Importo)

Interrogazione 1:

Determinare il nome ed il cognome delle persone che si sono iscritte ad almeno due conferenze diverse.

Interrogazione 2:

Determinare il nome delle conferenze del 2002 per le quali tutti gli iscritti vivono in una città diversa da quella dove ha avuto luogo la conferenza.

Schema "Donut":

TIPODONUT(Codice, Nome, DataPrimaProduzione)

CLIENTE(CF, Nome, Cognome, CittàResidenza, DataNascita)

ACQUISTO(CFCliente, CodiceDonut, DataAcquisto)

Interrogazione 1:

Determinare il nome e cognome dei clienti che hanno acquistato donut esattamente nello stesso giorno della loro prima produzione.

Interrogazione 2:

Determinare il codice dei donut acquistati solo da clienti residenti a San Francisco e nati dopo il 1971.

Schema "Ambulatorio":

MEDICO(Codice, Nome, Cognome, Specializzazione, Città, Telefono)
PAZIENTE(CodiceSSN, Nome, Cognome, DataNascita, Città, Telefono)
VISITA(CodiceMedico, CodicePaziente, Data, Diagnosi, CodiceMedicinale)

Interrogazione 1:

Determinare il nome, cognome e codice sanitario del paziente più vecchio.

Interrogazione 2:

Determinare il nome e cognome dei medici che hanno visitato tutti i pazienti.

Schema "Orologi":

PRODUTTORE(Nome, Nazione)
NEGOZIO(Codice, Città)
OROLOGIO(Codice, Tipo, PrezzoProd, AnnoProduzione, NomeProduttore)
FORNITURA(NomeProduttore, CodiceNegozio, CodiceOrologio, PrezzoVendita)

Interrogazione 1:

Determinare le città in cui ci sono negozi che vendono 'cronografi' con produttore di sede svizzera.

Interrogazione 2:

Determinare i produttori che servono tutti i negozi di orologi di Milano.

Esercizi SQL

Schema "Pittori"

PITTORE(Nome, DataNascita, DataMorte, LuogoNascita)
QUADRO(Titolo, NomePittore, NomeMuseo, Data)
MUSEO(Nome, Indirizzo, Citta)

Interrogazione 1:

Determinare il nome dei pittori nati a Parigi dopo il 1968 che hanno dipinto almeno tre quadri esposti al 'MOMA' di New York.

```
WITH Pittori_Parigi_1968 AS (
    SELECT Nome FROM PITTORE WHERE LuogoNascita="Parigi" AND DataNascita>"31-12-1968"
)

SELECT NomePittore,COUNT(Titolo) FROM QUADRO INNER JOIN Pittori_Parigi ON NomePittore=Pittori_Parigi.Nome IN
NER JOIN MUSEO ON NomeMuseo=MUSEO.Nome WHERE NomeMuseo="MOMA" AND Citta="New York" GROUP BY N
omePittore HAVING COUNT(Titolo)≥3`
```

Interrogazione 2:

Determinare i nomi dei musei che contengono almeno un quadro dipinto fra il 1300 ed il 1600 ma non contengono quadri dipinti da Leonardo da Vinci

```
SELECT DISTINCT NomeMuseo FROM QUADRO WHERE Data>="01-01-1300" AND Data<"01-01-1601" EXCEPT SELECT
DISTINCT NomeMuseo FROM QUADRO WHERE NomePittore="Leonardo Da Vinci"
```

Interrogazione 3:

Determinare i nomi dei musei che espongono almeno un quadro di un pittore ancora in vita.

```
SELECT DISTINCT NomeMuseo FROM QUADRO INNER JOIN PITTORE ON NomePittore=Nome WHERE DataMorte IS N
ULL
```

Schema "Feste"

FESTA(Codice, Costo, NomeRistorante)
REGALO(NomeInvitato, CodiceFesta, Regalo)
INVITATO(Nome, Indirizzo, Telefono)

Interrogazione 1:

Determinare, per ogni festa, il nome dell'invitato più generoso, ovvero dell'invitato che ha portato il maggior numero di regali.

```
SELECT R.CodiceFesta, R.NomeInvitato
FROM Regalo AS R
GROUP BY R.CodiceFesta, R.NomeInvitato
HAVING COUNT(*) >= ALL (
    SELECT COUNT(*)
    FROM Regalo AS R0
    WHERE R0.CodiceFesta = R.CodiceFesta
    GROUP BY R0.NomeInvitato )
```

Interrogazione 2:

Determinare il nome degli invitati che hanno partecipato alla festa più costosa.

```
SELECT DISTINCT NomeInvitato FROM REGALO WHERE CodiceFesta IN (SELECT CODICE FROM FESTA WHERE Costo >= ALL (SELECT Costo FROM FESTA))
```

Interrogazione 3:

| Determinare il codice delle feste dove almeno un invitato ha portato tre regali.

```
SELECT DISTINCT CodiceFesta FROM REGALO GROUP BY CodiceFesta,NomeInvitato HAVING Count(*)>=3
```

Schema "Autobus":

```
AUTISTA(CF, Nome, Cognome, Et'a, NumAutobus)  
PERCORRE(NumAutobus, NomeStrada)  
STRADA(NomeStrada, Lunghezza, Pedonale)
```

Interrogazione 1:

| Determinare le coppie di autisti che guidano lo stesso autobus.

```
SELECT A1.CF,A1.Nome,A2.CF,A2.Nome FROM AUTISTA A1 INNER JOIN AUTISTA A2 ON (A1.CF<>A2.CF AND A1.NumAutobus=A2.NumAutobus)
```

SBAGLIATA! Questa soluzione, considererebbe due volte le coppia di autisti. Per evitare ciò, imponiamo un ordinamento lessicografico

```
SELECT A1.CF,A1.Nome,A2.CF,A2.Nome FROM AUTISTA A1 INNER JOIN AUTISTA A2 ON (A1.CF<A2.CF AND A1.NumAutobus=A2.NumAutobus)
```

Interrogazione 2:

| Determinare gli autobus (NumAutobus) che percorrono tutte le strade pedonali, ed eventualmente anche altre strade.

Quello che in algebra sarebbe una divisone, in SQL è tradotto come un doppio NOT EXISTS.

Voglio tutti gli autobus per cui NON ESISTE una strada pedonale che l'autobus NON percorre.

```
SELECT DISTINCT P.NumAutobus  
FROM PERCORRE AS P  
WHERE NOT EXISTS(  
    SELECT A.NomeStrada FROM STRADA A WHERE A.Pedonale=TRUE AND NOT EXISTS (  
        SELECT * FROM PERCORRE B WHERE B.NumAutobus=P.NumAutobus AND B.NomeStrada=A.NomeStrada  
    )  
)
```

Interrogazione 3:

Conteggiare gli autobus che percorrono solo strade pedonali (non devono necessariamente percorrerle tutte).

Voglio tutti gli autobus per cui NON ESISTE una strada da esso percorso che NON SIA pedonale.

```
SELECT COUNT(DISTINCT P.NumAutobus)
FROM PERCORRE AS P
WHERE NOT EXISTS (
    SELECT *
    FROM PERCORRE AS B
    JOIN STRADA AS A ON A.NomeStrada = B.NomeStrada
    WHERE B.NumAutobus = P.NumAutobus AND A.Pedonale = FALSE
)
```

Soluzione alternativa

```
SELECT COUNT(*) FROM (
    SELECT DISTINCT NumAutobus
    FROM PERCORRE
    EXCEPT
    SELECT DISTINCT NumAutobus
    FROM PERCORRE JOIN STRADA ON NomeStrada = Nome
    WHERE Pedonale = FALSE
) AS SoloStradePedonali;
```

Schema “Supermercati”:

`SUPERMERCATO(Codice, Nome, Telefono, Via, Citta, CAP)`
`ACQUISTO(CodiceSupermercato, CodiceProdotto, CodiceTransazione)`
`PRODOTTO(Codice, Nome, Costo, CodiceScaffaleSupermercato)`
`TRANSAZIONE(Codice, Data, CodiceCliente, NomeCliente, CittaResidenzaCliente)`

Interrogazione 1:

Determinare le coppie (codice prodotto, codice cliente) tali per cui il cliente ha acquistato il prodotto sempre nello stesso supermercato.

```
SELECT DISTINCT CodiceProdotto,CodiceCliente FROM ACQUISTO A INNER JOIN TRANSAZIONE B ON A.CodiceTransazione=B.Codice WHERE NOT EXISTS(
    SELECT * FROM ACQUISTO C INNER JOIN TRANSAZIONE D ON C.CodiceTransazione=D.Codice WHERE C.CodiceProdotto=A.CodiceProdotto AND D.CodiceCliente=B.CodiceCliente AND C.CodiceSupermercato<>A.CodiceSupermercato
)
```

Interrogazione 2:

Determinare il codice delle transazioni che contengono almeno tre prodotti disposti sullo stesso scaffale.

```
SELECT DISTINCT CodiceTransazione,COUNT(*) FROM ACQUISTO INNER JOIN PRODOTTO ON CodiceTransazione=Codice GROUP BY CodiceTransazione,CodiceScaffaleSupermercato HAVING COUNT(*)>=3
```

Interrogazione 3:

Determinare quanto il signor 'Andrea Rossi' ha speso nel supermercato 'Famila' di 'Crema'

```
SELECT SUM(Costo) FROM SUPERMERCATO S INNER JOIN ACQUISTO A ON S.Codice=A.CodiceSupermercato INNER JOIN TRANSAZIONE T ON A.CodiceTransazione=T.Codice INNER JOIN PRODOTTO P On A.CodiceProdotto=P.Codice WHERE S.Nome="Famila" AND S.Città="Crema" AND NomeCliente="Andrea Rossi"
```

In Questo caso, la mia tabella intermedia conterrà solo transazioni del signor Rossi nel supermercato Famila di Crema, quindi tutti i dati sono già relativi al caso di interesse, non mi serve un raggruppamento.

Immagina questa mini-tabella dopo il **JOIN** e il **WHERE**:

NomeCliente	Supermercato	Città	Costo
Andrea Rossi	Famila	Crema	10.00
Andrea Rossi	Famila	Crema	15.00
Andrea Rossi	Famila	Crema	8.50

La query semplicemente esegue

```
SELECT SUM(Costo)
-- Output: 33.50
```

Schema "Ambulatorio"

```
MEDICO(Codice, Nome, Cognome, Specializzazione, Citta, Telefono)
PAZIENTE(CodiceSSN, Nome, Cognome, DataNascita, Citta, Telefono)
VISITA(CodiceMedico, CodicePaziente, Data, Diagnosi, CodiceMedicinale)
MEDICINALE(Codice, Nome, PrincipioAttivo, Prezzo)
```

Interrogazione 1:

Determinare nome e cognome dei pazienti, nati dopo il 1980, che assumono il medicinale 'Aulin' per curare l'emicrania.

```
SELECT DISTINCT P.Nome, P.Cognome FROM PAZIENTE P INNER JOIN VISITA V ON CodiceSSN=CodicePaziente INNER JOIN MEDICINALE M ON CodiceMedicinale=Codice WHERE DataNascita>"31-12-1980" AND M.Nome="Aulin" AND V.Diagnosi="Emicrania"
```

Interrogazione 2:

Determinare il codice sanitario dei pazienti che nel 2005 hanno speso più di 150 euro per curare l'anemia (si consideri la visita contestuale all'acquisto dei medicinali prescritti).

```
SELECT CodicePaziente
FROM Visita JOIN Medicinale ON CodiceMedicinale = Codice
WHERE Diagnosi = 'anemia'
AND Data BETWEEN 01-01-2005 AND 31-12-2005
```

```
GROUP BY CodicePaziente  
HAVING SUM(Prezzo) > 150
```

Interrogazione 3:

Determinare nome e cognome dei medici cardiologi che non hanno mai visitato pazienti della loro città per problemi di ipertensione.

```
SELECT Codice, Nome, Cognome  
FROM MEDICO  
WHERE Specializzazione = 'cardiologia'  
EXCEPT  
SELECT M.Codice, M.Nome, M.Cognome  
FROM MEDICO M  
JOIN VISITA V ON M.Codice = V.CodiceMedico  
JOIN PAZIENTE P ON P.CodiceSSN = V.CodicePaziente  
WHERE M.Citta = P.Citta  
AND Diagnosi = 'ipertensione'
```

Schema "Biscotti"

```
BISCOTTO(Codice, Nome, Forma, NomeCreatore, NomeDittaProduttrice)  
INGREDIENTE(Codice, Nome, Tipologia)  
COMPOSIZIONE(CodiceBiscotto, CodiceIngrediente, Quantit'a)
```

Interrogazione 1:

Determinare il codice dei biscotti di forma triangolare che sono prodotti da 'biSCOTTI&Figli' e che sono composti solo da ingredienti naturali.

```
SELECT CODICE FROM BISCOTTO B WHERE Forma="Triangolare" AND NomeDittaProduttrice="Biscotti&Figli" AND NO  
T EXISTS(  
    SELECT * FROM COMPOSIZIONE C INNER JOIN INGREDIENTE I ON CodiceIngrediente=Codice WHERE CodiceBiscot  
to=B.Codice AND Tipologia="Artificiale"  
)
```

Soluzione alternativa:

```
SELECT Codice  
FROM Biscotto  
WHERE Forma = 'triangolo'  
AND NomeDittaProduttrice = 'biSCOTTI&Figli'  
AND Codice NOT IN (  
    SELECT CodiceBiscotto  
    FROM Composizione, Ingrediente  
    WHERE CodiceIngrediente = Codice  
    AND Tipologia <> 'naturale' )
```

Interrogazione 2:

Determinare il codice dei biscotti che sono prevalentemente composti da ingredienti artificiali (cioè tali per cui il numero di ingredienti artificiali è prevalente).

```
CREATE VIEW Artificiale (Codice, NumArtificiali) AS
SELECT CodiceBiscotto, COUNT(*)
FROM Composizione JOIN Ingrediente
ON CodiceIngrediente = Codice
WHERE Tipologia = 'artificiale'
GROUP BY CodiceBiscotto

CREATE VIEW Naturale (Codice, NumNaturali) AS
SELECT CodiceBiscotto, COUNT(*)
FROM Composizione JOIN Ingrediente
ON CodiceIngrediente = Codice
WHERE Tipologia = 'naturale'
GROUP BY CodiceBiscotto

SELECT A.Codice
FROM Artificiale AS A JOIN Naturale AS N ON A.Codice = N.Codice
WHERE A.NumArtificiali > N.NumNaturali
```

Interrogazione 3:

Determinare i nomi dei creatori dei biscotti di forma 'esagonale' che sono composti almeno al 70% da ingredienti di tipo 'naturale'.

```
SELECT DISTINCT NomeCreatore FROM BISCOTTO B INNER JOIN COMPOSIZIONE C ON B.Codice=C.CodiceBiscotto I
NNER JOIN INGREDIENTE I ON C.CodiceIngrediente=I.Codice WHERE Forma="Esagonale" AND Tipologia="Naturale" G
ROUP BY CodiceBiscotto HAVING SUM(Quantità)>=70
```

Schema "Conferenze"

```
PERSONA(CF, Nome, Cognome, Affiliazione, Indirizzo, CittaResidenza)
CONFERENZA(Nome, Data, Indirizzo, Citta)
ISCRIZIONE(NomeConferenza, DataConferenza, CFPersona, Data, Importo)
```

Interrogazione 1:

Determinare le conferenze (nome e data) che hanno un numero di iscrizioni superiore a 100.

```
SELECT NomeConferenza, DataConferenza FROM ISCRIZIONE GROUP BY NomeConferenza, DataConferenza HAVING C
OUNT(*)>100
```

Interrogazione 2:

Determinare le conferenze (nome e data) per le quali tutti gli iscritti appartengono alla stessa città.

```

SELECT DISTINCT NomeConferenza, DataConferenza FROM ISCRIZIONE I INNER JOIN PERSONA P ON I.CFPersona=P.CF
WHERE NOT EXISTS(
    SELECT * FROM ISCRIZIONE I1 INNER JOIN PERSONA P1 ON I1.CFPersona=P1.CF WHERE I1.NomeConferenza=I.NomeConferenza AND I1.DataConferenza=I.DataConferenza AND P1.CittaResidenza<>P.CittaResidenza
)

```

Soluzione Alternativa:

```

SELECT NomeConferenza, DataConferenza
FROM ISCRIZIONE I
JOIN PERSONA P ON I.CFPersona = P.CF
GROUP BY NomeConferenza, DataConferenza
HAVING COUNT(DISTINCT P.CittaResidenza) = 1;
//Conto il numero di città distinte di quella conferenza

```

Interrogazione 3:

Determinare le conferenze (nome e data) cui si sono iscritte persone che risiedono in almeno 20 città diverse.

```

SELECT NomeConferenza, DataConferenza FROM ISCRIZIONE INNER JOIN PERSONA ON CFPersona=CF GROUP BY NomeConferenza, DataConferenza HAVING COUNT(DISTINCT Persona.Citt'aResidenza) >= 20

```

Schema “Donut”

```

DONUT(Codice, Nome, Crema, Glassa, Decorazione, DataPrimaProduzione)
CLIENTE(CF, Nome, Cognome, Citt'aResidenza, DataNascita)
ACQUISTO(CFCliente, CodiceDonut, DataAcquisto)

```

Interrogazione 1:

Determinare il codice dei donut con glassa al ‘cioccolato’ che sono stati prodotti per la prima volta nel 1999 e che sono stati acquistati da più di 2000 diversi clienti

```

SELECT Codice FROM DONUT INNER JOIN ACQUISTO ON Codice=CodiceDonut WHERE Glassa="Cioccolato" AND DataPrimaProduzione>="01-01-1999" AND DataPrimaProduzione < "01-01-2000" GROUP BY Codice HAVING COUNT(DISTINCT CFCliente)>2000

```

E' necessario scrivere DISTINCT CFCliente, perchè uno stesso CFCliente potrebbe comparire più volte nella tabella acquisto (un utente può effettuare più ordini)

Interrogazione 2:

Determinare il nome e cognome dei clienti che hanno acquistato donut farciti di crema ‘pasticciera’ e decorati con ‘zucchero a granelli’ ma che non hanno mai acquistato donut con glassa al ‘cioccolato’.

```

WITH CF_SoliCrema AS (
    SELECT CFCiente AS CF
    FROM ACQUISTO INNER JOIN DONUT ON CodiceDonut = Codice
    WHERE Crema = 'Pasticciera' AND Decorazione = 'Zucchero a granelli'

    EXCEPT

    SELECT CFCiente
    FROM ACQUISTO INNER JOIN DONUT ON CodiceDonut = Codice
    WHERE Glassa = 'Cioccolato'
)

SELECT Nome, Cognome
FROM CF_SoliCrema
JOIN CLIENTE ON CF_SoliCrema.CF = CLIENTE.CF;

```

Interrogazione 3:

Determinare il nome, cognome e il codice fiscale dei clienti che hanno acquistato almeno tre differenti tipi di Donut.

```

SELECT CF, Nome, Cognome
FROM CLIENTE
WHERE CF IN (
    SELECT CFCiente
    FROM ACQUISTO
    GROUP BY CFCiente
    HAVING COUNT(DISTINCT CodiceDonut) >= 3
);

```

Schema “Missioni”

MISSIONE(Codice, Citt'a, DataPartenza, Scopo, DurataPrevista)
AGENTE(Codice, Nome, Cognome, Specializzazione)
PARTECIPA(CodiceMissione, CodiceAgente, Ruolo)

Interrogazione 1:

Determinare il codice degli agenti specializzati in ‘travestimenti’ che hanno partecipato a missioni svolte nella citt'a di Washington.

```

SELECT DISTINCT A.Codice FROM AGENTE A INNER JOIN PARTECIPA P ON A.Codice=P.CodiceAgente INNER JOIN MISSIONE M ON P.CodiceMissione=M.Codice WHERE Specializzazione="Travestimenti" AND Città="Washington"

```

Interrogazione 2:

Determinare il codice delle missioni a cui hanno partecipato almeno due agenti specializzati in ‘spionaggio’ e due agenti con il ruolo di ‘infiltrati’ che non devono essere specializzati in ‘spionaggio’.

```

SELECT DISTINCT CodiceMissione
FROM Agente JOIN Partecipa ON Agente.Codice = CodiceAgente
WHERE Specializzazione = 'spionaggio'
GROUP BY CodiceMissione
HAVING COUNT(*) >= 2
INTERSECT
SELECT DISTINCT CodiceMissione
FROM Agente JOIN Partecipa ON Agente.Codice = CodiceAgente
WHERE Specializzazione <> 'spionaggio'
AND Ruolo = 'infiltrato'
GROUP BY CodiceMissione
HAVING COUNT(*) >= 2

```

Esercitazioni in classe

Data la seguente tabella: `movie(id, official_title, year, budget, length, plot)`

- Selezionare il titolo delle pellicole del 2010

```
SELECT official_title FROM imdb.movie WHERE year=2010;
```

- Selezionare i record delle pellicole con titolo "Interstellar"

```
SELECT * FROM imdb.movie WHERE official_title ILIKE 'interstellar'
```

Per essere più precisi, tale query fallisce nel confronto, nel caso ci fossero stringhe con spazi all'inizio o alla fine, per esempio, ipotizzando di avere questi record nella tabella

```
'Interstellar'  
'Interstellar ' -- con spazi extra  
'Interstellar ' -- con spazio alla fine
```

Il primo verrebbe correttamente selezionato, ma gli ultimi due no.

Risolviamo questo problema, con la funzione `TRIM()`, e le sue varianti `LTRIM()` e `RTRIM()` (Sinistra e destra)

```
SELECT *
FROM imdb.movie
WHERE TRIM(official_title) ILIKE 'interstellar';
```

Non ci sono alternative " valide e semplici ". Utilizzare l'operatore wildcard di carattere singolo non funzionerebbe

```
SELECT *
FROM imdb.movie
WHERE official_title ILIKE '_interstellar_';
```

Perchè così facendo, ammettere anche stringhe del tipo `XinterstellarX`

- Selezionare le pellicole nel cui titolo compaiano happy e family in qualsiasi posizione

```
SELECT * FROM imdb.movie WHERE official_title ILIKE '%happy%' AND official_title ILIKE '%family%';
```

- Selezionare le persone che hanno 'Mark' nel nominativo

```
SELECT * FROM imdb.person WHERE given_name ILIKE '%mark%';
```

- Selezionare le persone che hanno 'Mark' nel nome

```
SELECT * FROM imdb.person WHERE first_name ILIKE '%mark%';
```

- Selezionare le persone che si chiamano 'Mark' di cognome

```
SELECT * FROM imdb.person WHERE last_name ILIKE 'mark';
```

- Selezionare i film che sono prodotti in due country diverse

```
select distinct official_title, p1.country from imdb.movie m,imdb.produced p1,imdb.produced p2
where m.id = p1.movie and p1.movie=p2.movie and p1.country <> p2.country order by official_title ASC
```

- Selezionare il nome degli attori che hanno recitato nel film "Inception"

```
SELECT first_name,last_name FROM movie INNER JOIN crew ON movie.id=crew.movie INNER JOIN person ON cre
w.person=person.id WHERE role="actor" AND lower(official_title) = "inception"
```

- Selezionare le persone che sono decedute in un paese diverso da quello di nascita

Data la tabella `location(person,country,d_role)`

```
SELECT p.id,p.first_name,p.last_name,l_birth.country as birth_country, l_death.country as death_country
FROM person p,location l_birth, location l_death
WHERE l_birth.person=l_death.person AND l_birth.d_role="B" AND l_death.d_role="D" AND l_birth.country <> l_dea
th.country AND l_birth.person=p.id;
```

- Selezionare i film che non hanno materiali associati

Date le tabelle `movie(id,official_title...)` `material(id,description,language,movie)`

Possiamo risolverla con una differenza di tabelle (EXCEPT)

- Prendo tutti i record di movie
- Prendo tutti i movie presente in material
- Sottraggo

```
SELECT movie.id FROM imdb.movie EXCEPT SELECT DISTINCT material.movie FROM imdb.material
```

- Selezionare per ogni film, il numero di recensioni

```
SELECT id,count(rating.movie) AS numero_recensioni FROM imdb.movie LEFT JOIN imdb.rating ON movie.id = ratin
g.movie GROUP BY movie.id
```

- Selezionare per ogni film, la sua miglior valutazione

```
select movie,max(score/scale)*10 from imdb.rating group by movie order by 2 desc
```

"ORDER BY 2 DESC" vuol dire ordinare per la colonna 2, che sarebbe quella corrispondente alla funzione di aggregazione **max**

- Selezionare l'attore che ha recitato nel maggior numero di film

```
select person, count(distinct movie) from imdb.crew where p_role='actor' group by person order by 2 desc limit 1
```

Con questa soluzione, il "limit 1" mi taglierebbe eventuali attori diversi che hanno apparizioni uguali e massime.

```
WITH recitazioni AS (
    SELECT person, COUNT(DISTINCT movie) AS num_film
    FROM imdb.crew
    WHERE p_role = 'actor'
    GROUP BY person
),
massimo AS (
    SELECT MAX(num_film) AS max_film FROM recitazioni
)
SELECT person, num_film
FROM recitazioni
WHERE num_film = (SELECT max_film FROM massimo);
```

- Selezionare tutti gli attori che han recitato in tutti i film di genere crime

Utilizziamo l'operatore di divisione

A=Tutti gli attori, con associati i film in cui hanno recitato
B=Tutti i film crime
Risultato=A/B

In SQL:

```
Select id,given_name from imdb.person p where not exists (
    select * from imdb.genre g where g.genre="crime" and not exists (
        select * from imdb.crew where p_role="actor" and p.id=crew.person and g.movie=crew.movie
    )
);
```

```
CREATE TABLE publications.publication (
    id integer PRIMARY KEY,
    title character varying,
```

```

citedby integer,
issn character varying,
pubdate date,
pubname character varying,
pubtype character varying
);

CREATE TABLE publications.affiliation (
afid character varying PRIMARY KEY,
afname character varying,
afc city character varying,
afc country character varying
);

CREATE TABLE publications.author (
authid character varying PRIMARY KEY,
authname character varying,
authsurname character varying,
given_name character varying
);

CREATE TABLE publications.pub_author (
pubid integer NOT NULL REFERENCES publications.publication(id),
authid character varying NOT NULL REFERENCES publications.author(authid),
afid character varying REFERENCES publications.affiliation(afid),
PRIMARY KEY(pubid, authid, afid)
);

CREATE TABLE publications.abstract (
pubid integer NOT NULL REFERENCES publications.publication(id),
language character varying NOT NULL,
content text,
PRIMARY KEY(pubid, language)
);

CREATE TABLE publications.keyword (
pubid integer NOT NULL REFERENCES publications.publication(id),
keyword character varying NOT NULL,
language character varying NOT NULL,
PRIMARY KEY(pubid, keyword)
);

CREATE TABLE publications.expertise (
field character varying PRIMARY KEY,
parent_field character varying
);

ALTER TABLE publications.expertise ADD CONSTRAINT expertise_fk FOREIGN KEY (parent_field) REFERENCES publications.expertise(field) ON UPDATE CASCADE ON DELETE NO ACTION;

CREATE TABLE publications.author_expertise (
authid character varying NOT NULL REFERENCES publications.author(authid),
expertise character varying REFERENCES publications.expertise(field),

```

PRIMARY KEY(authid, expertise)
);

Trovare tutte le keyword non utilizzate in pubblicazioni di tipo "article"

$\pi, \sigma, \bowtie, \rho$

A = $\pi(id)(\sigma(pubtype="article"))(PUBLICATION)$

B = $\pi(keyword)(KEYWORD \bowtie (pubid=id A))$

Risultato = $\pi(keyword) KEYWORD - B$

In SQL:

```
with article_keyword as(
    select distinct keyword from publications.keyword inner join pulication.publication on pubid=id where pubtype="artic
le"
)
select keyword from publications.keyword EXCEPT select * from acticle_keyword
```

Con l'uso dell'operatore EXCEPT, assicurarsi che lo schema delle due relazioni a SX e DX dell'operatore siano uguali.

Esercizi normalizzazione

Docente	Dipartimento	Facoltà	Preside	Corso
Verdi	Matematica	Ingegneria	Neri	Analisi
Verdi	Matematica	Ingegneria	Neri	Geometria
Rossi	Fisica	Ingegneria	Neri	Analisi
Rossi	Fisica	Scienze	Bruni	Analisi
Bruni	Fisica	Scienze	Bruni	Fisica

- Individuare le proprietà della corrispondente applicazione, individuare eventuali ridondanze e anomalie nella relazione.

Notiamo una dipendenza funzionale tra Docente e Dipartimento, la ripetizione del valore di dipartimento per ogni tupla è ridondante, idem per la dipendenza funzionale tra Facoltà e Preside.

Se un docente cambiasse dipartimento, dovrei riflettere la modifica in tutte le tuple con valore ridondato (anomalia d'aggiornamento).

Una chiave per questa relazione, potrebbe essere l'insieme (Docente, Facoltà, Corso).

Se un docente volesse smettere di insegnare momentaneamente, ma senza "licenziarsi" dall'università, l'unico modo per tener traccia di tale docente è avere una tupla con valore nullo per facoltà e corso (inammissibile, anomalia d'eliminazione)

Per lo stesso motivo, avrei problemi ad aggiungere un docente che ancora non insegna in nessun corso.

- Proponi una chiave minimale e le dipendenze funzionali, decomporre in BCNF

Chiave minimale: (Docente, Facoltà, Corso)

Dipendenze funzionali: Docente-Dipartimento, Facoltà-Preside

BCNF: Per ogni dipendenza funzionale A→B, A è una chiave (superchiave minimale)

Si scomponete allora nelle seguenti tabelle:
A(Docente,Dipartimento), con Docente come chiave
B(Facoltà,Preside), con Facoltà come chiave
C(Docente,Facoltà,Corso), l'insieme dei 3 attributi è chiave.

Si consideri R1:

R1(person, first_name, last_name, given_name, movie, title, year, character)

Dipendenze funzionali:

person → first_name, last_name

person → given_name

movie → title, year

person,movie → character

Decomposizione in BCNF, Per ogni dip.funz. A→B, A deve essere una chiave.

RA(Person, first_name, last_name)

RB(movie,title,year)

RC(Person,Movie,Character,given_name)

Si consideri R2:

R2(movie, person, country)

La relazione descrive una persona e i film ai quali ha partecipato. Country rappresenta la sede della produzione del film e coincide con il paese di residenza dell'attore durante le riprese

Dipendenze funzionali:

Movie → Country

BCNF:

RA(Movie,Country), Ora da R2, tolgo Country e lascio solo ciò che serve ad identificare RA: rimangono fuori Movie e Person

RB(Movie,Person)

Consideriamo R3

R3(movie, country, agency)

La relazione descrive un film con i relativi paesi in cui è stato distribuito e per ciascuno di essi l'agenzia di distribuzione.

Si sappia che ogni agenzia è attiva in un solo paese

Non ci sono dipendenze funzionali, se non banali

Movie,Country → Agency

Agency → Country //viola BCNF

Non posso raggiungere BCNF, ma siamo però 3NF.

Movie,Country → Agency (OK!, Movie,Country è chiave)

Agency → Country (OK!, Country è parte della chiave della relazione)

Consideriamo R4

La relazione descrive l'associazione fra attori e film riportando la data di nascita dell'attore

R4(person, movie, birthdate)

Dipendenze funzionali:

Person → Birthdate, Viola BCNF, Viola 3NF, Viola 2NF

Chiave(person,movie)

Scomposizione BCNF:

RA(Person,Birthdate), Ora da person,movie,birthdate, tolgo tutti gli attributi nella tabella RA, tranne la PK: rimangono person e movie.

RB(Person,Movie)

Consideriamo R5

La relazione descrive il luogo di nascita di una persona in termini di città e paese dove le città hanno nome univoco

R5(person, city, country)

Dipendenze funzionali:

person → city

person → country

city → country //viola BCNF

Chiave: Person

Scomposizione in BCNF:

RA(Person,city)

RB(city,Country)

Consideriamo la relazione 9.11

Un giocatore può giocare in una sola squadra (o nessuna).

Un allenatore può allenare una sola squadra (o nessuna).

Una squadra ha un solo allenatore.

Una squadra ha diversi giocatori.

Una squadra appartiene a una sola città.

R(squadra, allenatore, città, giocatore)

Dipendenze funzionali:

Giocatore → Squadra

Allenatore → Squadra

Squadra → Allenatore

Squadra → Città

Chiave: Giocatore

Scomposizione in 3NF (Non posso raggiungere BCNF per via di Allenatore → Squadra):

RA(Giocatore, Squadra), Dipendenza Giocatore → Squadra

RB(Squadra, Allenatore, Città), Dipendenza Squadra → Allenatore, Squadra → Città, Allenatore → Squadra (Viola BCNF, ma è in 3NF)

ESERCIZI NORMALIZZAZIONE

Esercizio 1

Si consideri la seguente Tabella in cui valgono le dipendenze funzionali:

- CodEsame → Nome, NumStudenti
- CodProfessore → Dipartimento, NomeProfessore

Tabella (CodEsame, Nome, CodProfessore, NomeProfessore, NumStudenti, Dipartimento)

Portare Tabella in 3 Forma Normale, eseguendo una suddivisione senza perdite e che mantenga le dipendenze funzionali.

Esercizio 2

a) Individuare le dipendenze funzionali non banali presenti nella seguente tabella contenente informazioni sui corsi seguiti dagli studenti.

Tabella (CodCorso, NomeCorso, NomeProfessore, MatricolaProfessore, Dipartimento, MatricolaStudente, NomeStudente, NumeroOreCorso, NumeroCreditiCorso)

Si supponga che:

- Ciascun corso sia tenuto da un solo docente
- Ciascun professore afferisce ad un solo dipartimento
- Ciascuno studente possa seguire più corsi
- Ciascun professore possa tenere più corsi

b) Decomporre Tabella in 3 Forma Normale.

La decomposizione deve essere priva di perdite e deve mantenere le dipendenze funzionali.

Esercizio 1:

Tabella(CodEsame, Nome, CodProfessore, NomeProfessore, NumStudenti, Dipartimento)

Chiave Primaria: {CodEsame, CodProfessore}

Per essere in terza forma normale: Per ogni dipendenza funzionale $A \rightarrow B$, o A è una PK, oppure B fa parte della PK.

Dipendenze funzionali:

CodEsame → Nome, NumStudenti

CodProfessore → NomeProfessore, Dipartimento

Regole per normalizzare:

- Ogni dipendenza funzionale, diventa una tabella a se, in cui l'attributo/gli attributi che compaiono a sinistra della freccetta, sono PK per quella tabella
- Stabilire la relazione tra le tabelle create

Esame(**CodEsame**, Nome, NumStudenti)

Professore(**CodProfessore**, NomeProfessore, Dipartimento)

Esercizio 2:

Tabella(CodCorso, NomeCorso, NomeProfessore, MatricolaProfessore, Dipartimento, MatricolaStudente, NomeStudente, NumeroOreCorso, NumeroCreditiCorso)

Chiave primaria: {CodCorso, MatricolaStudente}

Dipendenze funzionali:

CodCorso → NomeCorso, NumeroOreCorso, NumeroCreditiCorso, MatricolaProfessore

MatricolaProfessore → NomeProfessore, Dipartimento

MatricolaStudente → NomeStudente

Si supponga che:

- Ciascun corso sia tenuto da un solo docente
- Ciascun professore afferisce ad un solo dipartimento
- Ciascuno studente possa seguire più corsi
- Ciascun professore possa tenere più corsi

Corso (**CodCorso**, NomeCorso, NumeroOreCorso, NumeroCreditiCorso, MatricolaProfessore)

Professore(**MatricolaProfessore**, NomeProfessore, Dipartimento)

Studente(**MatricolaStudente**, NomeStudente)

Seguire(**CodCorso**, **MatricolaStudente**)

Temi d'esame

