**UNIVERSITÀ DEGLI STUDI DI VERONA**
Dipartimento di Informatica
Corso di Laurea in Informatica

# BASI DI DATI

*Appunti del corso — A.A. 2025/2026*

*Studente:* **Kristian Xhani**

*Anno accademico:* 2025 / 2026

# Indice

**1 Introduzione** **6**
1.1 Problemi e soluzione . . . 6
1.2 DBMS . . . 6
1.3 Architettura di un DBMS . . . 7
1.4 Indipendenza dei dati . . . 7

**2 Progettazione di una base di dati** **8**
2.1 Cos'è una metodologia di progettazione . . . 9
2.2 Caratteristiche di una buona metodologia . . . 9

**3 Progettazione Concettuale : il Modello ER** **11**
3.1 Cos'è il modello ER . . . 11
3.2 Costrutti del modello ER . . . 11
3.3 Entità . . . 11
3.4 Relazione . . . 12
3.4.1 Relazione Ricorsiva . . . 13
3.4.2 Relazione Ternaria . . . 13
3.4.3 Indizi di modellazione errata (uso improprio della relazione ternaria) . . . 14
3.4.4 Esempio coretto di relazione ternaria . . . 15
3.4.5 Come capire se un concetto ha vita propria . . . 15
3.5 Attributo . . . 16
3.5.1 Attributo opzionale e multivalore . . . 16
3.5.2 Attributo composto . . . 17
3.6 Identificatore di Entità . . . 18
3.7 Generalizzazione . . . 19
3.7.1 Classificazione delle Generalizzazioni . . . 20

**4 Strategie per la progettazione concettuale** **22**
4.1 Strategia Top-Down . . . 22
4.2 Strategia Bottom-Up . . . 22
4.3 Strategia Inside-Out . . . 23
4.4 Analisi di qualità dello schema . . . 23
4.5 Ridondanze . . . 24
4.6 Semplificazioni dello schema . . . 25

**5 Modello Relazionale** **27**
5.1 Costrutti Principali del Modello Relazionale . . . 27
5.2 Domini di Base . . . 27
5.3 Costrutto Relazionale . . . 28
5.3.1 Definizione: Relazione come insieme di ennuple (LISTS) . . . 28
5.3.2 Definizione RELAZIONE come insieme di tuple (MAPPINGS) . . . 29
5.4 Come progettare una basi di dati relazionale . . . 30
5.4.1 Terminologia di schema di una relazione . . . 33
5.4.2 Terminologia Schema di una base di dati . . . 33
5.4.3 Terminologia istanza di una relazione di schema $R(X)$ . . . 33
5.4.4 Terminologia istanza di una base di dati di schema $S = \{R_1(X_1), \ldots, R_n(X_n)$ 33
5.5 Valori nulli . . . 33
5.6 Vincoli di integrità . . . 34

5.7 Esempi di vincoli . . . 34
5.8 Classificazione vincoli di integrità . . . 35
5.9 Notazione per indicare i vincoli strutturali . . . 37
5.9.1 Chiave primaria . . . 37
5.9.2 Vincolo di integrità referenziale . . . 37
5.10 Esercizio . . . 38
5.11 Progettazione Logica . . . 39
5.11.1 Analisi della derivabilità . . . 39
5.11.2 Analisi delle ridondanze . . . 40
5.11.3 Eliminazione delle Generalizzazioni . . . 40
5.11.4 Accorpamento delle entità figlie nel padre . . . 40
5.11.5 Accorpamento del padre nelle entità figlie . . . 41
5.11.6 Sostituzione con relazioni . . . 42
5.12 Traduzione verso il modello relazionale . . . 43
5.13 Classificazione delle relazioni binarie . . . 44
5.14 Regole di traduzione . . . 44

**6 Tecnologie NoSQL: Sistemi Document-Based 51**
6.1 Approcci NoSQL . . . 52
6.2 Sistemi Document-Store . . . 52
6.3 Regole di traduzione di uno schema ER in collezioni di documenti . . . 53
6.3.1 Prima Fase (entità principali) . . . 54
6.3.2 Seconda Fase (incapsulamento entità non principali) . . . 54
6.3.3 Terza Fase (relazioni non strutturali) . . . 57
6.4 Quarta fase . . . 58

**7 Algebra relazionale 65**
7.1 Classificazione degli operatori . . . 65
7.2 Operatori insiemistici . . . 66
7.2.1 Cardinalità degli operatori insiemistici . . . 66
7.3 Operatori specifici . . . 67
7.3.1 Operatore di rinominaizone . . . 67
7.3.2 Operatore di Selezione . . . 68
7.3.3 Proiezione . . . 69
7.3.4 Cardinalità degli operatori specifici . . . 69
7.3.5 Operatori di giunzione (JOIN) . . . 70
7.3.6 JOIN naturale . . . 70
7.3.7 Cardinalità del JOIN naturale . . . 72
7.3.8 THETA JOIN . . . 73
7.4 Algebra con valori nulli . . . 74
7.4.1 Selezione . . . 75
7.4.2 Join naturale . . . 75
7.5 Ottimizzazione di espressioni DML . . . 75
7.6 Equivalenza tra espressioni algebriche . . . 76
7.6.1 Equivalenza dipendente dallo schema . . . 76
7.6.2 Equivalenza assoluta . . . 76
7.7 Trasformazioni di equivalenza . . . 77
7.7.1 Idempotenza delle proiezioni . . . 77
7.8 Trasformazioni di equivalenza con il join . . . 77
7.9 Combinazione di anticipazione e idempotenza . . . 77

**8 Calcolo Relazionale** **82**
8.1 Sintassi . . . 83
8.2 Semantica . . . 84
8.3 Calcolo relazionale sulle tuple . . . 85

**9 Linguaggi di interrogazione nei sistemi NoSql** **90**

**10 Tecnologie - Transazioni** **93**
10.1 Tecnologia dei DBMS . . . 93
10.2 Architettura su 3 livelli dei DBMS . . . 93
10.3 Transazione . . . 94
10.3.1 Proprietà delle transazioni . . . 95
10.4 Architettura di un DBMS . . . 96

**11 Strutture fisiche e accesso ai dati** **101**
11.1 Rappresentazione logica del buffer . . . 102
11.2 Primitive del gestore del buffer . . . 103
11.3 Persistenza, scritture differite e log . . . 104
11.4 Algoritmo di ripresa in caso di guasto . . . 106
11.4.1 Algoritmo ripresa a caldo . . . 106
11.4.2 Algoritmo ripresa a freddo . . . 107

**12 Gestore metodi d'accesso** **112**
12.1 Operazioni sulle pagine . . . 113
12.1.1 Inserimento . . . 113
12.1.2 Cancellazione . . . 113
12.1.3 Accesso . . . 114
12.2 Criteri di organizzazione delle tuple nei blocchi . . . 114
12.3 Organizzazione sequenziale dei file . . . 114
12.3.1 Sequenziale disordinata (seriale) . . . 114
12.3.2 Sequenziale ordinata . . . 115
12.4 Indici . . . 117
12.4.1 Struttura degli indici primari . . . 117
12.4.2 Indici primari densi e sparsi . . . 118
12.4.3 Operazioni sugli indici primari . . . 119
12.4.4 Indici secondari . . . 120
12.4.5 Operazioni sugli indici secondari . . . 121
12.5 Strutture di accesso avanzate . . . 122
12.6 $B^+$-tree . . . 122
12.6.1 Struttura dei nodi . . . 123
12.6.2 Esempio $B^+$-tree . . . 124
12.6.3 Ricerca con chiave K . . . 127
12.6.4 Inserimento . . . 129
12.6.5 Cancellazione . . . 131
12.7 Strutture ad accesso calcolato (Hash) . . . 135
12.7.1 Confronto tra $B^+$-tree e Hash . . . 137

**13 Gestione della concorrenza** **138**
13.1 Anomalie da esecuzione concorrente . . . 138
13.1.1 Perdita di aggiornamento . . . 138

13.1.2 Lettura inconsistente . . . 139
13.1.3 Lettura sporca . . . 140
13.1.4 Aggiornamento fantasma . . . 140
13.1.5 Inserimento fantasma . . . 141
13.2 Schedule . . . 141
13.2.1 Ipotesi di commit-proiezione . . . 142
13.2.2 View-Equivalenza . . . 142
13.3 Conflict equivalenza . . . 146
13.4 Relazione tra CSR e VSR . . . 148
13.5 Locking a due fasi stretto (2PL stretto) . . . 152
13.5.1 Meccanismo di base . . . 152
13.5.2 Politica di concessione dei lock . . . 152
13.6 Regola per la serializzabilità . . . 153
13.6.1 2PL stretto . . . 154
13.6.2 Relazione tra 2PL stretto, CSR e VSR . . . 155
13.7 Blocco critico (Deadlock) . . . 155
13.8 Livelli di isolamento . . . 156
13.8.1 Implementazione dei livelli di isolamento . . . 156

**14 Gestore delle ottimizzazioni** **158**
14.1 Scansione sequenziale delle tuple di una tabella . . . 158
14.2 Operazione di ordinamento . . . 158
14.2.1 Esempio di Sort-Merge a 2 vie ($z = 2$) . . . 159
14.2.2 Costo del $z$-way Sort-Merge . . . 160
14.3 Accesso diretto tramite indice . . . 160
14.4 Calcolo del costo di un Join . . . 161
14.4.1 Nested Loop Join . . . 161

**15 Laboratorio** **163**
15.1 PostgreSQL . . . 163
15.2 SQL comandi base . . . 164
15.2.1 Creazione Tabella . . . 164
15.2.2 Concetti fondamentali del DDL . . . 165
15.3 Vincoli intrarelazionali . . . 169
15.4 Vincoli interrelazionali . . . 170
15.5 Modifica ed eliminazione delle strutture . . . 173
15.6 Cancellazione dei dati . . . 174
15.6.1 Eliminazione di una tabella . . . 174
15.7 Linguaggio di Interrogazione SQL . . . 175
15.8 Clausola SELECT . . . 176
15.9 Clausola FROM . . . 178
15.10 Clausola WHERE . . . 179
15.10.1 Operatore LIKE . . . 181
15.10.2 Operatore SIMILAR TO . . . 182
15.10.3 Operatore BETWEEN . . . 183
15.10.4 Operatore IN . . . 184
15.10.5 Operatore IS NULL . . . 185
15.11 SQL e Algebra Relazionale . . . 185
15.12 SELECT . . . 186
15.12.1 SELECT : alias di tabella o di attributo . . . 187

15.12.2 SELECT: senza proiezione . . . 187
15.12.3 Proiezione:gestione dei duplicati in SQL . . . 187
15.12.4 SELECT: Selezione,Proiezione,Join . . . 189
15.13 Interrogazioni con JOIN . . . 189
15.13.1 INNER JOIN . . . 190
15.13.2 LEFT OUTER JOIN . . . 191
15.13.3 RIGHT OUTER JOIN . . . 192
15.13.4 FULL OUTER JOIN . . . 193
15.14 Uso di variabili (alias) . . . 194
15.15 Ordinamento del risultato . . . 194
15.15.1 Operazioni Insiemistiche . . . 196
15.15.2 Unione . . . 196
15.15.3 Intersezione . . . 197
15.15.4 Differenza . . . 198
15.16 Operatori aggregati . . . 198
15.16.1 COUNT . . . 198
15.16.2 Operatori aggregati: SUM.MAX,MIN,AVG . . . 199
15.17 Interrogazioni con raggruppamento . . . 201
15.17.1 HAVING . . . 204
15.18 Interrogazioni nidificate . . . 205
15.18.1 Interrogazioni di tipo insiemistico . . . 209
15.18.2 Le VISTE . . . 210

# 1 Introduzione

- **Sistema informativo**: è l'insieme delle attività umane e dei dispositivi di memorizzazione ed elaborazione che organizzano e gestiscono le informazioni di interesse per un'organizzazione, di qualunque dimensione. Un sistema informativo **non contiene necessariamente tecnologia informatica**.
- **Dato**: è un elemento grezzo, un fatto isolato o un valore che, preso da solo, non ha significato.
- **Informazione**: nasce quando un dato viene interpretato e collocato in un contesto che lo rende comprensibile e utile.

Ad esempio: il numero *37* da solo è un semplice dato. Se però diciamo che "37 °C è la temperatura corporea normale di un essere umano", allora stiamo fornendo un'informazione.

**Per capire meglio**

Un dato è come un mattone: preso da solo non serve a molto. Quando i mattoni vengono messi insieme per costruire una casa, essi diventano informazione.

Per capire meglio come funziona un sistema informativo (cioè come circolano i dati e chi li utilizza) si utilizzano **diagrammi di flusso**, analizzando quattro aspetti principali:

1. Definizione di archivi e sorgenti di dati
2. Definizione degli utenti
3. Definizione di procedure e processi
4. Definizione dei flussi di dati

- **Base di dati**: è una collezione di dati utilizzata per rappresentare, con l'ausilio della tecnologia informatica, le informazioni di interesse di un sistema informativo.

## 1.1 Problemi e soluzione

Negli anni '70 ogni programma aveva i propri file di dati separati, gestiti direttamente dal **File System**. Questa soluzione presentava alcuni problemi:

- Scarsa efficienza nell'accesso ai dati su file
- Ridondanza dei dati (duplicazione)
- Inconsistenza dei dati (informazioni non aggiornate ovunque)
- Progettazione dei dati replicata per ogni programma

Per risolvere questi limiti vennero introdotti i **DBMS (Database Management System)**: software che fanno da intermediari tra programmi e dati.

**Vantaggi dei DBMS:**

- Maggiore astrazione e potenza espressiva per descrivere i dati
- Operazioni complesse di accesso e interrogazione tramite linguaggi specifici

## 1.2 DBMS

- **Linguaggi di interazione**:

- **DDL (Data Definition Language)**: definisce la struttura del database.
- **DML (Data Manipulation Language)**: consente di interrogare e modificare i dati (es. SQL).

**Differenza tra Database e DBMS**

**Database**: è l'insieme organizzato dei dati. *Esempio*: un archivio con le informazioni sugli studenti di una scuola.
**DBMS (Database Management System)**: è il software che permette di creare, gestire e consultare un database. *Esempi*: MySQL, PostgreSQL, Oracle, SQLite.

## 1.3 Architettura di un DBMS

- **Schema logico**: rappresentazione delle strutture e delle proprietà della base di dati, definita tramite i costrutti del modello del DBMS.
- **Schema interno**: rappresentazione della base di dati attraverso le strutture fisiche di memorizzazione.
- **Schema esterno**: descrive una porzione dello schema logico di interesse per uno specifico utente o applicazione.

## 1.4 Indipendenza dei dati

- **Indipendenza fisica**: lo schema logico della base di dati è indipendente dallo schema interno.
- **Indipendenza logica**: gli schemi esterni della base di dati sono indipendenti dallo schema logico.

## 2 Progettazione di una base di dati

Il ciclo di vita del processo di automazione di un sistema informativo comprende diverse fasi fondamentali, che vanno dall'analisi iniziale fino alla progettazione vera e propria.

### 1. Studio di fattibilità

In questa fase si valuta se il progetto è **realizzabile e conveniente**. Si analizzano i **costi**, i **tempi** e le **alternative possibili**. L'obiettivo è decidere se vale la pena procedere con lo sviluppo del sistema informativo. *È come chiedersi: "Ha senso costruire questo sistema?"*

### 2. Raccolta e analisi dei requisiti

In questa fase si cercano di capire i **bisogni reali degli utenti** e cosa dovrà fare il sistema. Si individuano le **proprietà e le funzionalità** principali (dati e applicazioni), producendo una descrizione **completa ma informale**. *È come chiedersi: "Cosa deve fare il sistema per essere utile?"*

### 3. Progettazione

Una volta compresi i requisiti, si passa alla fase di **progettazione del database e delle applicazioni**. Si decide come saranno organizzati i **dati** (modello ER, tabelle, relazioni) e come le applicazioni interagiranno con essi. *È la fase in cui si costruisce il progetto su carta, prima di realizzarlo.*

### 4. Progettazione del sistema

Una volta definiti i requisiti, si passa alla **progettazione del sistema**, che si divide in due parti principali:

- **Progettazione dei dati**: descrive in modo formale come i dati saranno organizzati, ovvero la struttura logica del database. Il risultato è lo **schema** del database (modello concettuale e logico).
- **Progettazione delle applicazioni**: definisce come gli utenti potranno interagire con i dati. Il risultato è la **specifica delle applicazioni**, cioè le operazioni e le funzionalità del sistema.

*In sintesi: si progetta sia la parte "tecnica dei dati", sia la parte "funzionale" che li utilizza.*

### 5. Implementazione e collaudo

Dopo la progettazione, si passa alla fase di **realizzazione concreta**:

- **Implementazione su un DBMS**: lo schema del database viene tradotto e caricato in un sistema di gestione (DBMS). Si inseriscono i dati e si realizzano le applicazioni.
- **Validazione e collaudo**: si verifica che il sistema funzioni correttamente e rispetti i requisiti definiti nelle fasi precedenti. Eventuali errori o mancanze vengono corretti prima dell'utilizzo effettivo.

*In questa fase il progetto diventa finalmente operativo e utilizzabile.*

**6. Progettazione dei dati**

In questa fase si realizza la **struttura logica del database**. Si definiscono:
- le **entità**, gli **attributi** e le **relazioni** tra di essi;
- le **chiavi primarie** e le **chiavi esterne**;
- i vincoli di integrità che garantiscono la coerenza dei dati.

Il risultato è uno **schema completo del database**, pronto per essere implementato.

**7. Implementazione su un DBMS**

Qui il progetto diventa concreto: lo schema logico del database viene **tradotto in un linguaggio SQL** e installato in un **DBMS** (Database Management System).
- Si creano le tabelle e le relazioni.
- Si inseriscono i dati iniziali.
- Si configurano le applicazioni che interagiranno con il database.

*È la fase di costruzione reale del sistema.*

**8. Validazione e collaudo**

Ultima fase del ciclo: si verifica che il sistema funzioni in modo corretto.
- Si confronta il risultato con i requisiti iniziali.
- Si testano le funzionalità e le prestazioni.
- Si correggono eventuali errori o incongruenze.

Solo dopo la validazione, il sistema è **pronto per l'uso reale** da parte degli utenti.

## 2.1 Cos'è una metodologia di progettazione

Una **metodologia di progettazione** definisce il modo in cui si affronta e si organizza il progetto di un database. Serve a garantire che il lavoro sia ordinato, coerente e ripetibile, evitando errori e migliorando la qualità del risultato finale.

Una metodologia di progettazione è costituita da:

- una **decomposizione** in passi dell'attività di progetto, che suddivide il lavoro in fasi chiare e gestibili;
- un insieme di **strategie** da seguire e di **criteri di scelta**, che guidano le decisioni del progettista;
- un insieme di **modelli di riferimento**, che forniscono strutture e standard per rappresentare correttamente i dati.

## 2.2 Caratteristiche di una buona metodologia

Una buona metodologia deve essere:

- **Generale**, ovvero applicabile a diversi tipi di progetti e contesti;
- **Facile da usare**, in modo che i progettisti possano adottarla senza difficoltà;
- **Efficace**, cioè in grado di produrre un risultato di qualità, con un progetto **completo, coerente e corretto**.

In sintesi, una metodologia ben definita aiuta a costruire database **ben strutturati, comprensibili e facilmente mantenibili nel tempo**.

Per essere una buona metodologia deve essere:

- Generale
- Facile da Usare
- In grado di produrre un risultato di qualità
- **Progettazione concettuale**:L'obiettivo della progettazione concettuale è quello di rappresentare in modo chiaro, formale e completo le informazioni che dovranno essere memorizzate nel database, senza ancora preoccuparsi di come verranno salvate in un sistema reale.
  In questa fase:
  - si analizza cosa deve contenere la base di dati (le entità, i loro attributi e le relazioni tra di esse);
  - si realizza un modello concettuale, di solito attraverso il diagramma Entità-Relazione (E-R);
  - si mantiene tutto indipendente dalla tecnologia, cioè senza pensare al tipo di DBMS o al linguaggio che verrà usato in seguito.
- **Progettazione logica** : il suo obbiettivo è quello di tradurre lo schema concettuale nello schema logico aderente al modello dei dati del DBMS scelto per l'implementazione.
- **Progettazione fisica** : Completare lo schema logico con i parametri relativi alla memorizzazione fisica dei dati e con gli opportuni metodi d'accesso (INDICI) per garantire un accesso efficiente ai dati.
- **Schema Concettuale**:È la rappresentazione astratta dei dati e delle loro relazioni, realizzata durante la progettazione concettuale. Usa strumenti come il diagramma E-R (Entità-Relazione) e mostra cosa deve contenere la base di dati, senza dipendere dal tipo di DBMS.
- **Schema logico**:È la traduzione dello schema concettuale in una forma più tecnica, adatta al modello di database scelto (ad esempio, relazionale). Si decide come rappresentare le entità e le relazioni (tabelle, chiavi primarie, chiavi esterne).
- **Schema fisico**: È la realizzazione concreta del database all'interno di un DBMS specifico. Si definiscono i nomi delle tabelle, i tipi di dato (VARCHAR, INT, DATE...), gli indici, i vincoli e l'organizzazione fisica dei file.

**Differenza tra Progettazione e Schema**

**Progettazione** → è il **processo**, cioè l'attività di analisi e definizione che porta alla costruzione del modello dei dati. In questa fase si studiano le informazioni, si identificano le entità e le relazioni e si costruisce il modello concettuale.
**Schema** → è il **risultato finale** della progettazione. Rappresenta in modo formale e sintetico la struttura dei dati ottenuta al termine del processo.

**In sintesi:** la **progettazione** è l'attività di creazione, mentre lo **schema** è il prodotto che ne deriva.

# 3 Progettazione Concettuale : il Modello ER

## 3.1 Cos'è il modello ER

Il **modello Entità-Relazione (ER)** è un linguaggio visivo usato per rappresentare i dati e comprendere come essi siano collegati tra loro. Viene utilizzato soprattutto nella fase iniziale di progettazione di un database, per avere un'idea chiara di quali informazioni siano necessarie e di come esse siano organizzate. Esso è un linguaggio formale e non ambiguo.

## 3.2 Costrutti del modello ER

Gli strumenti fondamentali del modello ER vengono definiti come **costrutti**. Ogni costrutto viene descritto specificando:

- il **significato**;
- la **sintassi grafica**;
- la **rappresentazione delle sue istanze**.

**Cos'è un'istanza**

Un'**istanza** è un singolo elemento concreto che appartiene a un'entità.
- L'**entità** descrive il concetto generale (es. *Studente*);
- L'**istanza** è un esempio reale di quell'entità (es. *Mario Rossi*, matricola S123).

**Esempio:**
- Entità: *Studente* → attributi: matricola, nome, cognome;
- Istanza: {Matricola: S123, Nome: Luca, Cognome: Rossi}.

In sintesi: l'entità è il modello generale, mentre l'istanza è un caso reale e specifico di quell'entità.

## 3.3 Entità

- **Significato**: un'entità $E$ rappresenta una classe di oggetti con le seguenti caratteristiche:
  - hanno proprietà comuni;
  - hanno esistenza autonoma;
  - hanno identificazione univoca (esiste una chiara corrispondenza tra gli oggetti istanze di $E$ e i concetti istanziati nel sistema informativo).
- **Sintassi grafica**: un'entità $E$ si rappresenta nello schema con un rettangolo che contiene il nome dell'entità.
- **Istanza**: un'istanza dell'entità $E$ è un oggetto appartenente alla classe rappresentata da $E$. Si indica con $I(E)$ l'insieme delle istanze di $E$ che esistono nella base di dati in un certo istante. (come definito nel riquadro precedente).
- **Esempio**: rappresento con il costrutto entità il concetto di `PERSONA`, ipotizzando che nei requisiti sia indicata la necessità di gestire nella base di dati le informazioni che descrivono un gruppo di persone e che tali informazioni possiedano nel sistema informativo le caratteristiche indicate nella semantica del costrutto entità.

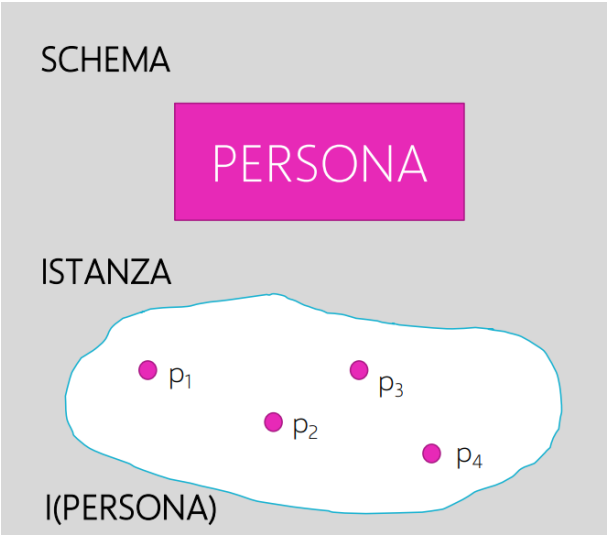



Figura 1: Esempio di Entita

## 3.4 Relazione

- **Significato**: una relazione $R$ rappresenta un **legame logico** tra due o più entità. Possono esistere relazioni:
  - **binarie**, quando coinvolgono due entità diverse;
  - **ternarie**, quando coinvolgono tre entità;
  - **ennarie**, quando coinvolgono $n$ entità.

  **Caso particolare:** una relazione binaria può essere definita anche sulla stessa entità (detta **relazione ricorsiva**), ad esempio la relazione *"impiegato-supervisore"* all'interno dell'entità *Dipendente*.

- **Sintassi grafica**: una relazione $R$ si rappresenta nello schema con un **rombo** collegato, tramite linee spezzate, alle entità coinvolte nella relazione. Il nome della relazione viene scritto all'interno o accanto al rombo.

- **Istanza**: data una relazione $R$ tra $n$ entità $E_1, E_2, \ldots, E_n$, un'istanza della relazione è una **ennupla** di istanze delle entità coinvolte. L'insieme di tutte le ennuple costituisce la **popolazione della relazione**.

  In altre parole, le ennuple rappresentano i **collegamenti concreti** tra le istanze delle entità. Se immaginiamo la relazione come una tabella, ciascuna riga (ennupla) contiene i riferimenti alle istanze delle due entità collegate.

  **Esempio:** Sia $E = \textit{Studente}$ e $F = \textit{Corso}$, e sia $R = \textit{Iscrizione}$. Un'istanza della relazione può essere l'ennupla (Mario Rossi, Analisi 1), che rappresenta il fatto che lo studente "Mario Rossi" è iscritto al corso "Analisi 1". L'insieme di tutte le ennuple del tipo (Studente, Corso) forma la popolazione della relazione *Iscrizione*.

- **Esempio** : Supponiamo di aver inserito nello schema le entità *PERSONA* e *COMUNE*, e ipotizziamo che nei requisiti sia indicata la necessità di gestire la *RESIDENZA* delle persone nei comuni italiani. Rappresento con il costrutto relazione il concetto di *RESIDENZA*

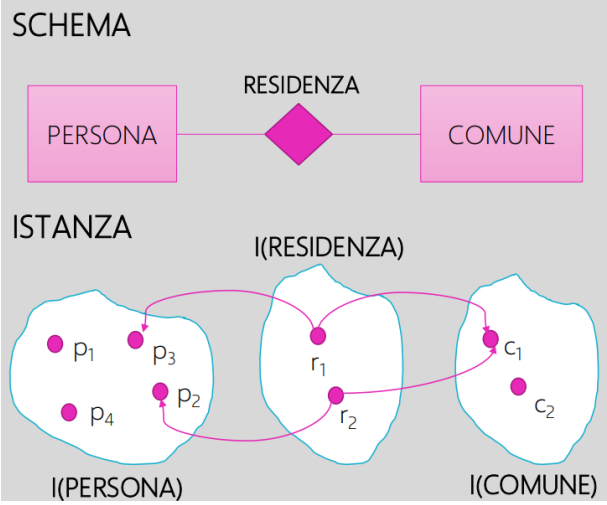



Figura 2: Esempio di Relazione

### 3.4.1 Relazione Ricorsiva

E' una relazione binaria sulla stessa entità.

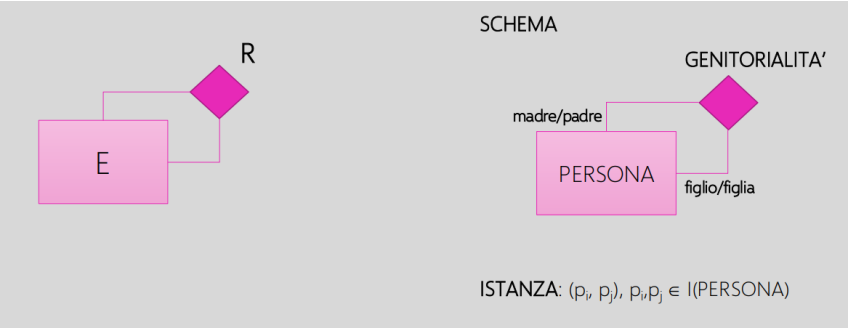
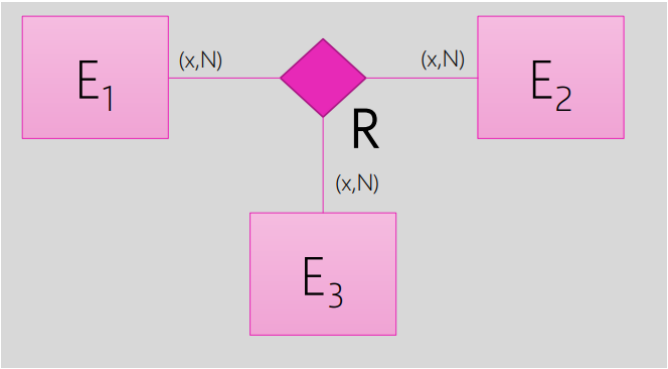



Figura 3: Relazione Ricorsiva

### 3.4.2 Relazione Ternaria

Una **relazione ternaria** coinvolge tre entità e viene utilizzata quando un concetto del sistema informativo può essere rappresentato solo se sono presenti contemporaneamente tre istanze, una per ciascuna entità coinvolta.

- Ogni istanza della relazione associa **una terna** di entità ($E_1$, $E_2$, $E_3$).
- L'esistenza della relazione **richiede sempre** la presenza di tutte e tre le entità: non è possibile che ne manchi una.
- Non esistono casi in cui la stessa terna debba essere rappresentata più volte.




### 3.4.3 Indizi di modellazione errata (uso improprio della relazione ternaria)

In alcuni casi, l'utilizzo di una **relazione ternaria** può indicare una modellazione errata. Questo avviene quando una relazione tra tre entità non richiede realmente la presenza simultanea di tutte e tre le istanze.

- Se per rappresentare lo stato del sistema informativo sarebbe sufficiente collegare **solo due entità per volta**, la relazione ternaria è superflua.
- In tali casi, la relazione ternaria può essere sostituita da **tre relazioni binarie indipendenti**.

**Esempio:** La relazione ternaria *Partecipa(Medico, Paziente, Ospedale)* è errata se nel sistema esistono già le relazioni binarie: *Lavora_in(Medico, Ospedale)*, *Visita(Medico, Paziente)* e *Ricoverato_in(Paziente, Ospedale)*. In questo caso, la relazione ternaria non aggiunge alcuna informazione nuova ed è quindi ridondante. Un ulteriore indizio di modellazione errata si verifica quando, in una relazione ternaria $R(A, B, C)$, si osserva il seguente comportamento:

- Se una coppia $(a, b)$ compare in una terna $(a, b, c)$ istanza della relazione,
- e una coppia $(b, c')$ compare in una terna $(a', b, c')$ istanza della relazione,

allora risultano automaticamente presenti anche le terne $(a, b, c')$ e $(a', b, c)$.

Questo significa che le combinazioni tra le entità non dipendono realmente tutte e tre tra loro: la relazione ternaria può quindi essere scomposta in più relazioni binarie indipendenti senza perdita di informazione. **In sintesi:** se da due terne si possono dedurre automaticamente altre combinazioni tra le stesse entità, la relazione ternaria è ridondante e andrebbe sostituita da più relazioni binarie.

**Kristian semplifica in questo modo**

Le **relazioni ternarie** vanno usate solo quando il significato della relazione dipende **strettamente da tutte e tre le entità coinvolte**.

- Se anche una sola entità può essere esclusa o rappresentata separatamente, allora la relazione non è veramente ternaria.
- Nella maggior parte dei casi, è meglio sostituire una relazione ternaria con **più relazioni binarie indipendenti**.

**In sintesi:** se puoi evitarla, **evitala**. Le relazioni ternarie rendono più difficile la progettazione e la traduzione nel modello relazionale.

### 3.4.4 Esempio coretto di relazione ternaria

Vogliamo rappresentare il fatto che un prodotto venga fornito ad un dipartimento da un fornitore. Ogni dipartimento può scegliere quali prodotti ordinare dai fornitori selezionando un sottoinsieme dei prodotti forniti dal fornitore. Ogni fornitore vende a più dipartimenti prodotti diversi. Ogni prodotto viene venduto da più fornitori.

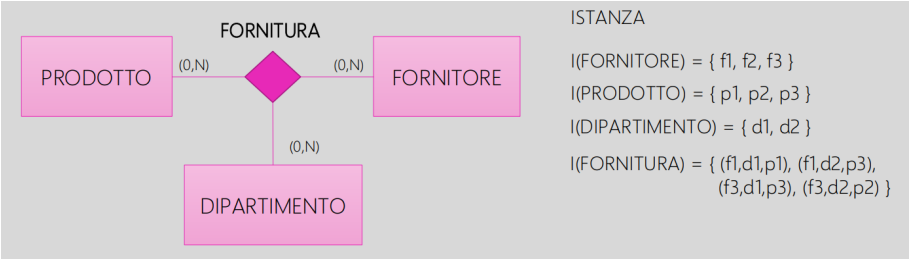



Figura 4: Esempio Relazione Ternaria

### 3.4.5 Come capire se un concetto ha vita propria

Nel modello Entità-Relazione, per stabilire se un concetto debba essere rappresentato come **entità** o come **relazione**, è fondamentale chiedersi se tale concetto *ha una vita propria* all'interno del sistema informativo.

**Definizione**

Un concetto ha **vita propria** se può esistere in modo indipendente dalle altre entità, cioè se ha senso considerarlo come un oggetto autonomo, anche quando non è collegato ad altri.

Per verificarlo, è possibile porsi alcune domande:

- Può esistere anche senza essere collegato ad altre entità?
- Ha senso registrarlo nel sistema anche se non è associato ad altri elementi?
- Possiede attributi che descrivono sé stesso e non il legame con altri?

Se la risposta è **sì** a queste domande, il concetto possiede vita propria e deve essere rappresentato come **entità**. In caso contrario, se esiste solo come collegamento tra altre entità, è più corretto rappresentarlo come **relazione**.

| **Caratteristica** | **Entità (ha v** |
|---|---|
| Esiste indipendentemente da altre | Può esistere da so |
| Ha senso da sola | Ha significato |
| Attributi | Descrivono |
| Esempio | Studente, Insegnam |

**Esempio applicativo:** nel caso della relazione *ESAME*, se essa rappresenta l'atto con cui uno studente sostiene un insegnamento (registrando solo *data* e *voto*), non ha vita propria e va modellata come **relazione**. Se invece si vogliono gestire appelli d'esame con *aula*, *docente* e *orario*, esso assume significato autonomo e può essere rappresentato come **entità**.

## 3.5 Attributo

- **Significato** :Rappresenta una proprietà elementare di un'entità (o relazione). Ogni attributo di entità (o relazione) associa ad ogni istanza di entità (o relazione) UNO e UN SOL valore appartenente ad un dominio (insieme di VALORI AMMISSIBILI).
- **Sintassi grafica** : collegato a una entità e a forma di pallino
- **Esempio** : Rappresento con il costrutto entità il concetto di PERSONA, ipotizzando che nei requisiti sia indicata la necessità di gestire nella base di dati il nome e il cognome di un gruppo di persone. Aggiungo quindi il nome e il cognome come attributi dell'entità PERSONA.

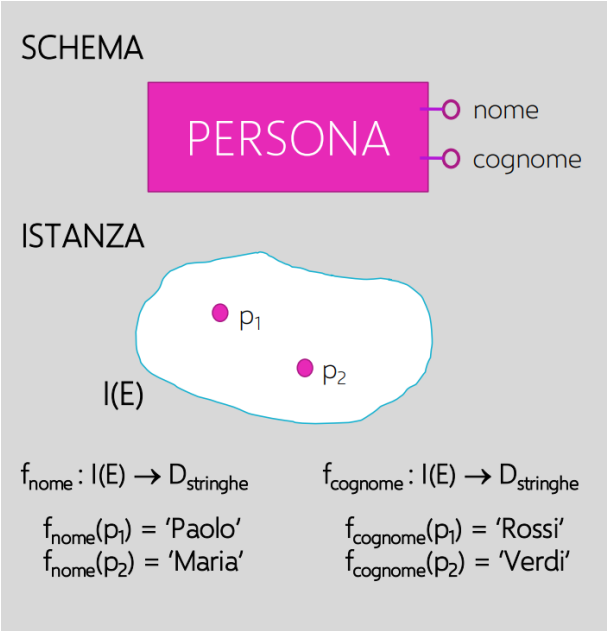



Figura 5: Esempio Attributo

### 3.5.1 Attributo opzionale e multivalore

Un **attributo opzionale** o **multivalore** deriva da un attributo normale, specificando un **vincolo di cardinalità** sui valori che esso può assumere. Il vincolo di cardinalità indica quante volte un attributo può comparire in corrispondenza di una stessa entità. Il valore predefinito è **(1,1)**, che significa "obbligatorio e singolo valore".

I principali casi sono:

- **(0,1)** : attributo opzionale → può essere assente o presente una sola volta;
- **(1,N)** : attributo multivalore → deve essere presente almeno una volta e può comparire più volte;
- **(0,N)** : attributo opzionale e multivalore → può essere assente o, se presente, avere più valori.

**Esempio pratico**

Si consideri l'entità *PERSONA*, per la quale si vogliono memorizzare:

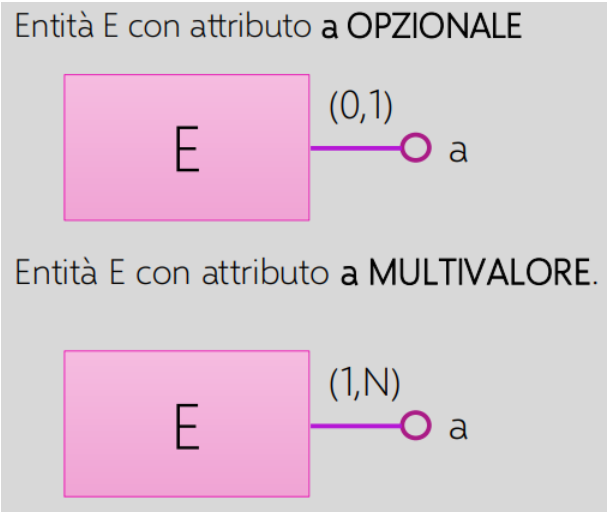



Figura 6: Attributo opzionale e/o multivalore

- **Codice fiscale**, **nome**, **cognome** e **data di nascita** — attributi obbligatori e a singolo valore;
- **Patente di guida** — attributo **opzionale** $(0, 1)$, poiché una persona può possederla o meno;
- **Recapiti telefonici** — attributo **multivalore** $(1, N)$, poiché ogni persona può avere uno o più numeri di telefono (ad esempio cellulare, casa, lavoro).

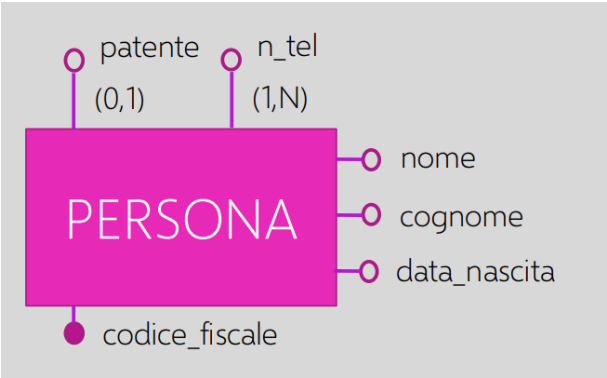



Figura 7: Esempio di attributo opzionale e multivalore per l'entità PERSONA

### 3.5.2 Attributo composto

Un **attributo composto** consente di raggruppare più attributi di un'entità o di una relazione che presentano una stretta **affinità di significato**. In questo modo si possono rappresentare in modo più ordinato e logico informazioni che, pur essendo distinte, appartengono a un unico concetto complessivo.

Ad esempio, per rappresentare l'entità *PERSONA*, si possono gestire nella base di dati i seguenti attributi: **codice fiscale**, **nome**, **cognome**, **data di nascita** e **indirizzo**.

L'attributo **indirizzo** può essere considerato un attributo composto, poiché a sua volta può essere suddiviso nei sottoattributi:

- **Città**
- **Via**
- **Numero civico**

In un diagramma ER, l'attributo composto viene rappresentato con un'ellisse principale collegata a più ellissi minori, ognuna delle quali rappresenta un sottoattributo.

Figura 8: Esempio di attributo composto per l'entità PERSONA

## 3.6 Identificatore di Entità

- **Significato**:Data un'entità E, è un insieme di proprietà (attributi e/o relazioni) che identificano univocamente le istanze dell'entità. Un insieme di proprietà **IDENTIFICA UNIVOCAMENTE** le istanze di E se non esistono due istanze di E che presentano gli stessi valori/istanze nelle proprietà dell'insieme.
- **Sintassi grafica** : collegato a una entità e a forma di un pallino pieno, puo anche essere una linea che intercorre in due attributi sempre a fine con un pallino pieno

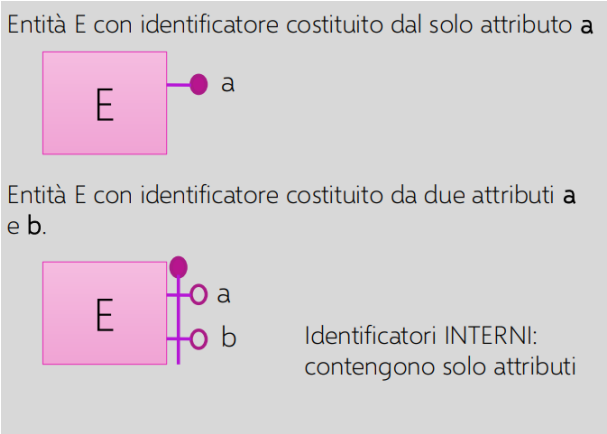



Figura 9: sintassi grafica: identificatori interni

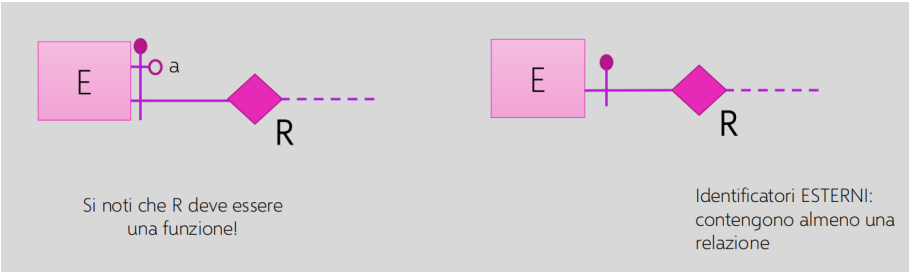
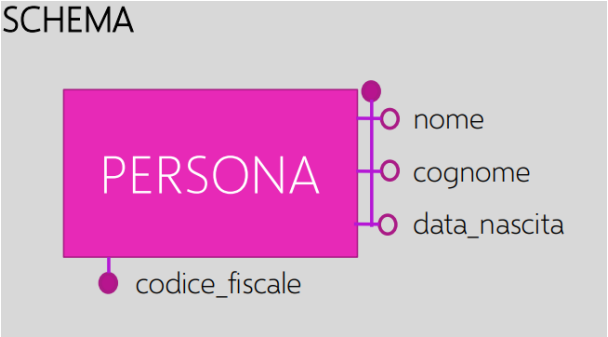



Figura 10: sintassi grafica: identificatori esterni

- **Esempio**: Rappresento con il costrutto entità il concetto di *PERSONA*, ipotizzando che nei requisiti sia indicata la necessità di gestire nella base di dati, il codice fiscale, il nome, il cognome e la data di nascita di un gruppo di persone.




La scelta dell'identificatore va sempre fatta considerando proprietà significative per il sistema informativo. A livello concettuale l'introduzione di nuovi attributi identificatori (ad esempio l'ID) è da **evitare**.

## 3.7 Generalizzazione

La **generalizzazione** rappresenta un legame logico, analogo al concetto di **ereditarietà** nella programmazione a oggetti, tra un'entità padre e una o più entità figlie. Serve per modellare situazioni in cui più entità condividono alcune caratteristiche comuni, ma possiedono anche attributi o relazioni specifiche.

- **Significato**: È una relazione logica tra un'entità padre $E$ e $n$ entità figlie $(E_1, E_2, \ldots, E_n)$, dove $E$ rappresenta una classe di oggetti più generale rispetto a quelle rappresentate dalle entità figlie. Le entità figlie ereditano gli attributi e le relazioni dell'entità padre, aggiungendo eventualmente elementi propri.
- **Sintassi grafica**: Le entità figlie coinvolte nella generalizzazione vengono collegate all'entità padre mediante una linea che converge in un triangolo (o freccia) rivolto verso l'entità padre.
- **Esempio**: Si consideri l'entità *PERSONA*, per la quale nei requisiti è indicata la necessità di distinguere tra *STUDENTE* e *DOCENTE*. In questo caso si può utilizzare una generalizzazione in cui:
  - **PERSONA** è l'entità padre, che contiene gli attributi comuni (ad esempio: nome, cognome, data di nascita);

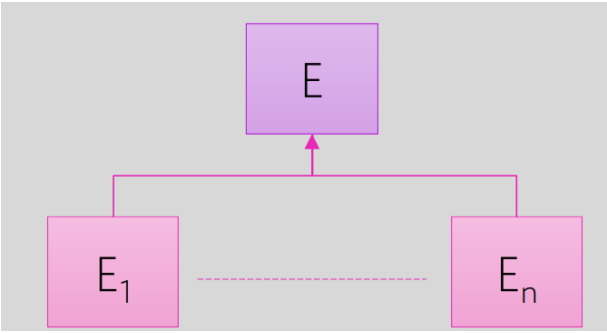



Figura 11: Sintassi grafica della generalizzazione

– **STUDENTE** e **DOCENTE** sono le entità figlie, che ereditano gli attributi di PERSONA e possono avere attributi propri (es. matricola per STUDENTE, dipartimento per DOCENTE).

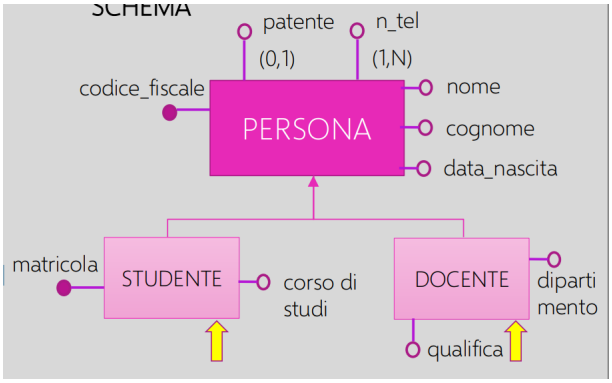



Figura 12: Esempio di generalizzazione tra PERSONA, STUDENTE e DOCENTE

**Proprietà**

Le **proprietà delle istanze di entità** che partecipano a una **generalizzazione** sono le seguenti:

- Ogni istanza di un'entità figlia $E_i$ è anche un'istanza dell'entità padre $E$.
- Ogni proprietà (attributi, identificatori e relazioni) dell'entità padre $E$ è condivisa da tutte le entità figlie $E_1, E_2, \ldots, E_n$.

In altre parole, le entità figlie **ereditano** tutte le caratteristiche dell'entità padre, e possono aggiungerne di proprie.

### 3.7.1 Classificazione delle Generalizzazioni

Nel modello **Entità–Relazione**, una generalizzazione può assumere diverse forme, a seconda del tipo di legame tra l'entità padre (superclasse) e le entità figlie (sottoclassi). Le combinazioni derivano dall'incrocio di due caratteristiche fondamentali:

- il grado di **copertura** (Totale o Parziale);
- il grado di **esclusività** (Esclusiva o Sovrapposta).

Ecco i principali tipi di gerarchia nel modello ER:

1. **Totale ed Esclusiva (t,e)**: ogni istanza della superclasse appartiene **obbligatoriamente** a una e una sola sottoclasse. *Esempio:* l'entità *VEICOLO* è generalizzata in *MOTO* e *AUTO*. Ogni veicolo deve essere o moto o auto, ma non può essere entrambi.
2. **Totale e Sovrapposta (t,s)**: ogni istanza della superclasse deve appartenere ad almeno una sottoclasse, ma può appartenere a **più sottoclassi** contemporaneamente. *Esempio:* l'entità *PERSONA* è generalizzata in *STUDENTE* e *LAVORATORE*. Una persona può essere sia studente sia lavoratore.
3. **Parziale ed Esclusiva (p,e)**: non tutte le istanze della superclasse devono appartenere a una sottoclasse; se appartengono, ne hanno solo una. *Esempio:* l'entità *ANIMALE* è generalizzata in *DOMESTICO* e *SELVATICO*. Un animale può essere domestico, selvatico o nessuno dei due, ma non entrambi.
4. **Parziale e Sovrapposta (p,s)**: non tutte le istanze devono appartenere a una sottoclasse; chi appartiene può essere in più sottoclassi. *Esempio:* l'entità *PERSONA* è generalizzata in *SPORTIVO* e *ARTISTA*. Una persona può essere sportiva, artista, entrambe o nessuna delle due.

**Kristian semplifica in questo modo**

Le **gerarchie di generalizzazione** si classificano in base a due proprietà fondamentali:

- **Totale** → ogni istanza della superclasse deve appartenere ad almeno una sottoclasse.
- **Parziale** → alcune istanze della superclasse possono non appartenere a nessuna sottoclasse.
- **Esclusiva (o Disgiunta)** → un'istanza può appartenere solo a una sottoclasse.
- **Sovrapposta** → un'istanza può appartenere a più sottoclassi contemporaneamente.

In sintesi: **Totale/Parziale** indica *se tutte le istanze partecipano*, mentre **Esclusiva/Sovrapposta** indica *come partecipano*.

# 4 Strategie per la progettazione concettuale

Lo sviluppo di uno **schema concettuale** può essere considerato a tutti gli effetti un **processo di ingegnerizzazione**, in quanto richiede un insieme di fasi metodiche e ragionate per trasformare le specifiche di un problema reale in una rappresentazione astratta ma coerente e precisa.

In altre parole, progettare uno schema concettuale non significa solo disegnare entità e relazioni, ma **analizzare, organizzare e strutturare le informazioni** in modo sistematico, seguendo principi simili a quelli utilizzati nell'ingegneria del software.

**Nota Bene**

Per **ingegnerizzazione** si intende l'*applicazione di un metodo rigoroso e pianificato* alla progettazione, allo sviluppo e alla verifica di un sistema complesso. Nel contesto della modellazione dei dati, ciò significa:

- partire da requisiti chiari e ben definiti;
- progettare in modo graduale e controllato (Top-Down, Bottom-Up o Inside-Out);
- assicurarsi che lo schema finale sia corretto, completo e privo di ambiguità.

*Ricorda che uno schema concettuale è una rappresentazione astratta e indipendente dal linguaggio di implementazione, che descrive le informazioni e le relazioni fondamentali di un sistema informativo.*

## 4.1 Strategia Top-Down

La strategia **Top-Down** consiste nel partire da una **visione globale del sistema** e procedere verso una **maggiore specificazione dei dettagli**. È un approccio deduttivo: si comincia con concetti generali e astratti per poi raffinarli gradualmente fino a ottenere uno schema concettuale completo e preciso.

- **Fase 1:** considerare le specifiche globalmente e produrre uno schema iniziale **completo ma con pochi concetti astratti**.
- **Fase 2: raffinare** progressivamente i concetti astratti fino ad arrivare a uno schema concettuale **dettagliato in ogni parte**.

**Nota Bene**

La strategia Top-Down è particolarmente utile quando si ha una visione chiara e globale del dominio applicativo. Permette di mantenere coerenza tra le parti del progetto, ma richiede una buona capacità di astrazione e pianificazione.

*Esempio:* prima si individuano le entità principali (*Studente*, *Corso*, *Docente*), poi si aggiungono relazioni e attributi specifici come *data di nascita*, *esame sostenuto*, *voto*, ecc.

## 4.2 Strategia Bottom-Up

La strategia **Bottom-Up** adotta un approccio opposto rispetto al Top-Down. In questo caso si parte dai **dettagli elementari** per poi integrarli progressivamente fino a ottenere la visione complessiva del sistema. È un metodo **induttivo**, ideale quando si dispone di molte informazioni di base ma non ancora di una struttura generale.

- **Fase 1: decomporre** le specifiche iniziali in **parti elementari**, ovvero piccole frasi o porzioni di testo che descrivono uno stesso concetto o una singola informazione.

- **Fase 2: generare gli schemi** per ciascuna parte elementare individuata.
- **Fase 3: fondere gli schemi** parziali introducendo, se necessario, altri costrutti del modello E-R, in modo da **integrare** tutti gli elementi e ottenere lo **schema finale**.

**Nota Bene**

La strategia Bottom-Up è utile quando le specifiche non sono complete o sono distribuite in frammenti separati. Permette di costruire lo schema passo dopo passo, ma richiede attenzione nella fase di integrazione per evitare ridondanze o incoerenze tra i diversi schemi parziali.

*Esempio:* si analizzano prima frasi come "uno studente sostiene un esame" o "un docente insegna un corso", si crea uno schema per ciascuna, e infine si uniscono tutti in un unico schema concettuale completo.

## 4.3 Strategia Inside-Out

La strategia **Inside-Out** rappresenta un approccio intermedio rispetto a Top-Down e Bottom-Up. Si parte da alcuni **concetti centrali o fondamentali** del dominio, detti **concetti guida**, e si costruisce lo schema espandendolo progressivamente verso l'esterno.

- **Fase 1: individuare** nelle specifiche alcuni concetti importanti, detti **concetti guida**.
- **Fase 2: generare gli schemi** relativi ai concetti guida scelti.
- **Fase 3: fondere gli schemi** precedenti e aggiungere gradualmente gli altri concetti fino a ottenere lo **schema concettuale completo**.

**Nota Bene**

La strategia Inside-Out è utile quando si riescono a identificare con chiarezza alcuni elementi chiave del sistema. Permette di concentrarsi inizialmente sulle parti più rilevanti e poi di estendere lo schema, garantendo coerenza e controllo durante l'espansione.

*Esempio:* si parte dai concetti guida come *Studente* e *Corso*, si definiscono le loro relazioni principali, e poi si estende lo schema aggiungendo altri concetti come *Esame*, *Docente* o *Dipartimento*.

## 4.4 Analisi di qualità dello schema

L'**analisi di qualità** di uno schema concettuale serve a valutare se il modello prodotto rappresenta correttamente e in modo efficace il dominio di interesse. Essa comprende quattro aspetti fondamentali:

- Verifica di correttezza
- Verifica di completezza
- Verifica di minimalità
- Verifica di leggibilità

**Verifica di correttezza** Uno schema concettuale è **corretto** se utilizza propriamente i costrutti del modello concettuale adottato (nel nostro caso, il modello E-R). La correttezza si valuta controllando che lo schema rispetti le regole sintattiche e semantiche del modello.

- **Errori sintattici:** utilizzo scorretto dei costrutti del modello E-R, senza rispettarne le regole formali. *Esempi:*
  - introduzione di una generalizzazione di relazioni;
  - relazioni non collegate ad alcuna entità;
  - relazioni che possiedono un identificatore.
- **Errori semantici (errori d'uso):** uso improprio dei costrutti del modello, senza rispettarne il significato. *Esempi:*
  - uso di una relazione per rappresentare un concetto che dovrebbe essere un'entità autonoma;
  - uso di un attributo per rappresentare un concetto complesso o collegato ad altri concetti in più punti dello schema.

**Verifica di completezza e minimalità** Uno schema concettuale è **completo** se rappresenta tutte le informazioni di interesse e consente di eseguire tutte le operazioni previste dalle specifiche. È invece **minimale** quando tutti i concetti presenti nei requisiti sono rappresentati una sola volta, evitando ridondanze o duplicazioni.

**Verifica di leggibilità** La leggibilità riguarda la **chiarezza e l'organizzazione visiva** dello schema. Uno schema ben leggibile deve essere ordinato, coerente e facilmente interpretabile anche da chi non lo ha progettato.

**Nota Bene**

Un buon schema concettuale deve essere:
- **Corretto** → rispetta le regole sintattiche e semantiche del modello E-R;
- **Completo** → include tutte le informazioni richieste;
- **Minimale** → evita ridondanze e duplicazioni;
- **Leggibile** → chiaro, ordinato e facilmente comprensibile.

Solo quando tutte queste condizioni sono soddisfatte, lo schema può essere considerato di **alta qualità**.

## 4.5 Ridondanze

Uno **schema concettuale non minimale** può contenere delle **ridondanze**, ossia rappresentazioni multiple dello stesso concetto logico. I casi più comuni di ridondanza sono:

- **Ereditarietà delle relazioni**
- **Cicli di relazioni**

È **probabile, ma non certo**, che la relazione *R2* sia ridondante. Va quindi verificato se *R2* rappresenti lo stesso legame logico già espresso da *R1*; se così fosse, *R2* deve essere eliminata in quanto effettivamente ridondante.

È **probabile, ma non certo**, che una delle relazioni $R_i$ sia ridondante. Va quindi verificato se la relazione $R_i$ possa essere ottenuta **dalla composizione delle altre relazioni**. Se così fosse, $R_i$ deve essere eliminata in quanto effettivamente ridondante.

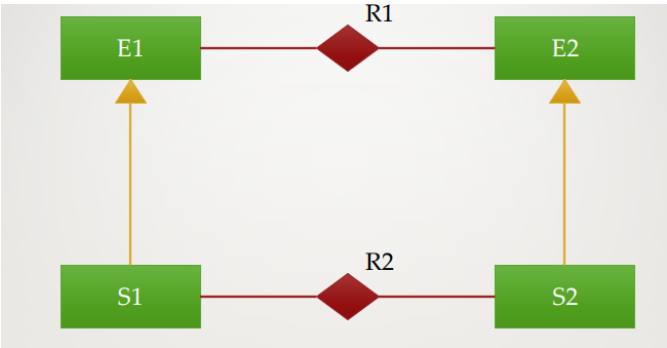



Figura 13: Ereditarietà delle relazioni

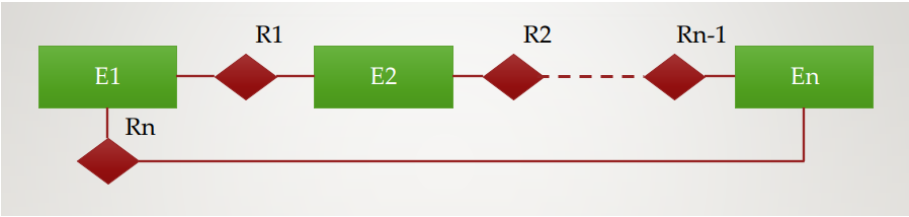



Figura 14: Ciclo di relazioni

## 4.6 Semplificazioni dello schema

Non sempre le **relazioni ternarie** sono facili da leggere e interpretare; per questo motivo, la loro eliminazione può talvolta migliorare la **leggibilità dello schema concettuale**.

Le seguenti operazioni possono essere applicate per semplificare o trasformare una relazione ternaria:

- **Trasformazione di una relazione ternaria in un'entità** Questa operazione è **sempre applicabile** e consiste nel sostituire la relazione ternaria con una nuova entità, collegata tramite relazioni binarie alle entità originarie.
- **Riduzione di una relazione ternaria a due relazioni binarie** Questa operazione è possibile solo se una delle entità coinvolte partecipa alla relazione ternaria con **cardinalità (1,1)**. In tal caso, la relazione ternaria può essere sostituita da due relazioni binarie equivalenti, mantenendo il significato logico dello schema.

**Nota Bene**

La trasformazione o riduzione delle relazioni ternarie deve essere eseguita con attenzione: l'obiettivo è migliorare la leggibilità dello schema senza alterarne il significato semantico.

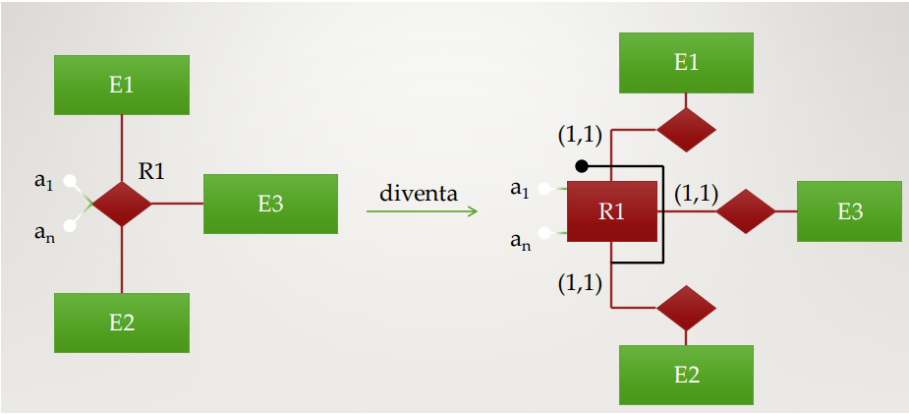



Figura 15: Trasformazione di una relazione ternaria in un'entità

Figura 16: Riduzione di una relazione ternaria a due relazioni binarie

Un ulteriore miglioramento nella leggibilità si ottiene esplicitando, nelle **generalizzazioni sovrapposte**, l'entità che rappresenta la sovrapposizione. In questo modo, la generalizzazione sovrapposta viene trasformata in una **generalizzazione esclusiva**, rendendo lo schema più chiaro e facilmente interpretabile.

# 5 Modello Relazionale

Nel corso del tempo si sono sviluppati diversi modelli per la rappresentazione e l'organizzazione dei dati all'interno dei sistemi informativi. Tra i principali ricordiamo:

- **Modello gerarchico/reticolare** (anni 1960–1970): tra i primi modelli di gestione dei dati, organizzati in forma gerarchica o a rete.
- **Modello relazionale** (anni 1980–oggi): introdotto da Edgar F. Codd, rappresenta i dati mediante tabelle (relazioni) e costituisce la base dei moderni DBMS relazionali.
- **Modello ad oggetti** (anni 1990–2000): integra i concetti di programmazione orientata agli oggetti con la gestione dei dati.
- **Modello a oggetti/relazionale (object–relational)** (anni 2000–oggi): un'evoluzione del modello relazionale che consente l'integrazione di tipi complessi, eredità e relazioni tra oggetti. Esempio: SQL3.
- **Modello a grafo – RDF (Resource Description Framework)** (anni 2005–oggi): rappresenta i dati sotto forma di grafo orientato, utile nel contesto del web semantico. Esempio: Neo4j.
- **Modelli noSQL** (document-based, column-based, key–value, ecc.) (anni 2010–oggi): nati per la gestione di grandi volumi di dati non strutturati e per garantire maggiore scalabilità. Esempio: MongoDB.

## 5.1 Costrutti Principali del Modello Relazionale

Il modello relazionale costituisce la base dei moderni database e si fonda su alcuni **costrutti principali**:

- **Domini di base**: rappresentano l'insieme dei valori ammissibili per ciascun attributo.
- **Relazione (o tabella)**:
  - Definizione come insieme di **n–uple** (righe).
  - Ogni n–upla rappresenta un record all'interno della relazione.
- **Superchiavi, Chiavi e Chiavi primarie**: identificano univocamente le tuple all'interno di una relazione.
- **Vincoli di integrità referenziale**: garantiscono la coerenza dei riferimenti tra tabelle.
- **Vincoli di integrità generici**: assicurano la validità logica dei dati rispetto a condizioni predefinite.

## 5.2 Domini di Base

Sono i domini da cui si scelgono i valori delle proprietà delle istanze di informazione da rappresentare. I domini tipici sono:

- Caratteri
- Stringhe
- Numeri
- Numeri decimali a virgola fissa

- Numeri decimali a virgola mobile
- Dominio del Tempo

## 5.3 Costrutto Relazionale

Una relazione può essere vista come una tabella ad esempio:

| **Città** | **CAP** | **Abitanti** |
|---|---|---|
| Milano | 20100 | 1 300 000 |
| Verona | 37100 | 350 000 |
| Brescia | 25100 | 250 000 |

Tabella 1: Esempio di relazione **Città** con attributi *(Nome, CAP, Abitanti)*

Una tabella è un contenitore di dati la cui struttura è caratterizzata da una lista di collonne:

- I dati sono scritti nelle righe dove ogni riga descrive le caratteristiche di un'istanza dell'informazione da rappresentare
- I valori contenuti nelle colonne descrivono sempre la stessa proprietà delle istanze di informazione da rappresentare.

### 5.3.1 Definizione: Relazione come insieme di ennuple (LISTS)

Dati $n$ insiemi di valori (detti **domini**) $D_1, \ldots, D_n$ con $n > 0$, si indica con $D_1 \times \ldots \times D_n$ il loro **prodotto cartesiano**:

$$D_1 \times \ldots \times D_n = \{(v_1, \ldots, v_n) \mid v_1 \in D_1, \ldots, v_n \in D_n\}$$

Una **relazione** $\rho$ di grado $n$ è un qualsiasi **sottoinsieme** di $D_1 \times \ldots \times D_n$:

$$\rho \subseteq D_1 \times \ldots \times D_n$$

dove $(v_1, \ldots, v_n)$ è una **ennupla** della relazione e $|\rho|$ rappresenta la **cardinalità** della relazione (ovvero il numero di ennuple presenti). Si noti che i domini $D_1, \ldots, D_n$ possono essere a **cardinalità infinita**, mentre le relazioni sono SEMPRE a **cardinalità finita**. Possiamo dire che

- Non è definito alcun ordinamento sulle ennuple di una relazione
- Non sono ammessi DUPLICATI di una ennupla
- Nella definizione di relazione come insieme di ennuple, i valori nelle ennuple sono ordinati

Relazione delle città

$\rho$ = { (MILANO, 20100, 1.300.000), (VERONA, 37100, 350.000), (BRESCIA, 25100, 250.000) }

$$\rho \subseteq D_1 \times D_2 \times D_3$$

$D_1$ = Stringhe di caratteri
$D_2$ = Numeri interi
$D_3$ = Numeri interi

Figura 17: Esempio di relazione con insieme di ennuple

**Accesso ai valori di una ennupla:** Se $t$ e una ennupla $(V_1, \ldots, V_n)$ il valore posto in i-esima posizione si indica con la notazione $t[i]$. Questa modalità di accesso ai valori non è efficace per l'uso pratico delle relazioni. Si preferisce quindi assegnare un nome alle colonne;ciò conduce all'introduzione della definizione come insieme di **TUPLE**.

### 5.3.2 Definizione RELAZIONE come insieme di tuple (MAPPINGS)

Sia $X$ un insieme di nomi e si $\Delta$ l'insieme di tutti i domini di base ammessi dal modello. Si definisce la funzione:
$DOM : X \rightarrow \Delta$

Tale funzione associa ad ogni nome A di X un dominio $DOM(A)$ di $\Delta$.I nomi di X si dicono **attributi**.

Una tupla t su X è una funzione: $t : X \rightarrow \bigcup_{A \in X} DOM(A)$
dove: $t[A] = v \in DOM(A)$

**Una relazione su X è un <u>insieme di tuple su X</u>, dove X è l'insieme di attributi della relazione.**

Esempio di relazione come insieme di tuple

Relazione delle città (vista in precedenza):

X = {Nome, CAP, Abitanti}

DOM(Nome) = Stringhe di caratteri

DOM(CAP) = Numeri interi

DOM(Abitanti) = Numeri interi

$\rho_X = \{ t_1, t_2, t_3 \}$

$t_1$[Nome] = MILANO, $t_2$[Nome] = VERONA $t_3$[Nome] = BRESCIA

$t_1$[CAP] = 20100, $t_2$[CAP] = 37100, $t_3$[CAP] = 25100

$t_1$[Abitanti] = 1.300.000, $t_2$[Abitanti] = 350.000, $t_3$[Abitanti] = 250.000

Figura 18: Esempio

- Una relazione è un insieme di tuple e quindi non può contenere tuple duplicate
- I domini per gli attirbuti possono essere solo domini di base, non sono ammessi altri domini, ne il prodotto cartesiano di domini
- In generale una base di dati relazionale è costitutia da piu relazioni.

## 5.4 Come progettare una basi di dati relazionale

Nel modello relazionale lo schema è costituito da un insieme di relazioni o tabelle. Lo strumento formale che serve per definire dipendenze tra attributi nel modello relazionale è costituito dalle **dipendenze funzionali**.

**Per Esempio:** Vogliamo rappresentare in una base di dati relazionale le informazioni sulla proprietà degli appartamenti siti nel comune di Verona.

- Per ogni appartamento volgiamo memorizzare : il codice ecografico (univoco), la categoria catastale e l'indirizzo composto da : via,numero civico, subalterno
- Per ogni proprietario si registra: il codice fiscale, il nome, il cognome e la data di nascita.
- Un appartamento può essere in comproprietà e un proprietario può possedere più appartamenti.

**PROPRIETÀ**

| Categoria | Via | Civico | Sub | Codice ECO | Codice Fiscale | Nome | Cognome | Data Nas |
|---|---|---|---|---|---|---|---|---|
| A1 | Roma | 23 | A | RMAXXX | RSSMRR75D | Mario | Rossi | 1/1/1975 |
| A1 | Roma | 23 | A | RMAXXX | BNCMRA80F | Maria | Bianchi | 10/9/1980 |
| A3 | Milano | 9 | | MILYYY | BNCMRA80F | Maria | Bianchi | 10/9/1980 |
| A3 | Milano | 9 | | MILYYY | BNCPLO77A | Paolo | Bianchi | 3/1/1977 |
| A1 | Torino | 2 | C | TORZZZ | BNCPLO77A | Paolo | Bianchi | 3/1/1977 |
| A2 | Garibaldi | 33 | | GARWWW | RSSMRR75D | Mario | Rossi | 1/1/1975 |

Figura 19: Schema relazionale con solo una relazione

Nella relazione unica ogni riga rappresenta un contratto di proprietà e richiede di precisare tutti i dati di un appartamento e tutti i dati di un proprietario. Ciò implica che:

- Non è possibile rappresentare un appartamento che non abbia un proprietario, né rappresentare un proprietario senza appartamento,
- I dati di un appartamento vengono replicati per ogni proprietario
- I dati di un proprietario vengono replicati per ogni appartamento di proprietà.

**Anomalie:** La presenza di ridondanza inutile nella base di dati provoca le seguenti anomalie:

- **ANOMALIE DI AGGIORNAMENTO** : per aggiornare il valore di un attributo si è obbligati a modificare tale valore su più tuple.
- **ANOMALIE DI INSERIMENTO** : per inserire una nuova tupla è necessario inserire valori al momento sconosciuti (sostituibili da valori nulli) per gli attributi non disponibili.
- **ANOMALIE DI CANCELLAZIONE** : per cancellare una tupla è necessario cancellare valori ancora validi oppure inserire valori nulli per gli attributi da cancellare.

E possibile ridurre la ridondanza nei dati decomponendo la relazione. Sono possibili diverse decomposizioni.

**APPARTAMENTO**

| Categoria | Via | Civico | Sub | Codice ECO |
|---|---|---|---|---|
| A1 | Roma | 23 | A | RMAXXX |
| A3 | Milano | 9 | | MILYYY |
| A1 | Torino | 2 | C | TORZZZ |
| A2 | Garibaldi | 33 | | GARWWW |

**PROPRIETARIO**

| Codice Fiscale | Nome | Cognome | Data Nas |
|---|---|---|---|
| RSSMRR75D | Mario | Rossi | 1/1/1975 |
| BNCMRA80F | Maria | Bianchi | 10/9/1980 |
| BNCPLO77A | Paolo | Bianchi | 3/1/1977 |

Figura 20: Decomposizione 1

**APPARTAMENTO**

| Categoria | Via | Civico | Sub | Codice ECO | Codice Fiscale |
|---|---|---|---|---|---|
| A1 | Roma | 23 | A | RMAXXX | RSSMRR75D |
| A1 | Roma | 23 | A | RMAXXX | BNCMRA80F |
| A3 | Milano | 9 | | MILYYY | BNCMRA80F |
| A3 | Milano | 9 | | MILYYY | BNCPLO77A |
| A1 | Torino | 2 | C | TORZZZ | BNCPLO77A |
| A2 | Garibaldi | 33 | | GARWWW | RSSMRR75D |

**PROPRIETARIO**

| Codice Fiscale | Nome | Cognome | Data Nas |
|---|---|---|---|
| RSSMRR75D | Mario | Rossi | 1/1/1975 |
| BNCMRA80F | Maria | Bianchi | 10/9/1980 |
| BNCPLO77A | Paolo | Bianchi | 3/1/1977 |

Figura 21: Decomposizione 2

**PROPRIETARIO**

| Codice ECO | Codice Fiscale | Nome | Cognome | Data Nas |
|---|---|---|---|---|
| RMAXXX | RSSMRR75D | Mario | Rossi | 1/1/1975 |
| RMAXXX | BNCMRA80F | Maria | Bianchi | 10/9/1980 |
| MILYYY | BNCMRA80F | Maria | Bianchi | 10/9/1980 |
| MILYYY | BNCPLO77A | Paolo | Bianchi | 3/1/1977 |
| TORZZZ | BNCPLO77A | Paolo | Bianchi | 3/1/1977 |
| GARWWW | RSSMRR75D | Mario | Rossi | 1/1/1975 |

**APPARTAMENTO**

| Categoria | Via | Civico | Sub | Codice ECO |
|---|---|---|---|---|
| A1 | Roma | 23 | A | RMAXXX |
| A3 | Milano | 9 | | MILYYY |
| A1 | Torino | 2 | C | TORZZZ |
| A2 | Garibaldi | 33 | | GARWWW |

Figura 22: Decomposizione 3

**APPARTAMENTO**

| Categoria | Via | Civico | Sub | Codice ECO |
|---|---|---|---|---|
| A1 | Roma | 23 | A | RMAXXX |
| A3 | Milano | 9 | | MILYYY |
| A1 | Torino | 2 | C | TORZZZ |
| A2 | Garibaldi | 33 | | GARWWW |

**PROPRIETARIO**

| Codice Fiscale | Nome | Cognome | Data Nas |
|---|---|---|---|
| RSSMRR75D | Mario | Rossi | 1/1/1975 |
| BNCMRA80F | Maria | Bianchi | 10/9/1980 |
| BNCPLO77A | Paolo | Bianchi | 3/1/1977 |

**PROPRIETÀ**

| Codice ECO | Codice Fiscale |
|---|---|
| RMAXXX | RSSMRR75D |
| RMAXXX | BNCMRA80F |
| MILYYY | BNCMRA80F |
| MILYYY | BNCPLO77A |
| TORZZZ | BNCPLO77A |
| GARWWW | RSSMRR75D |

Figura 23: Decomposizione 4

Si noti che i legami tra relazioni si realizzano attraverso la replicazione di un insieme di attributi, dove il legame tra due tuple si intende stabilito quando esse presentano gli stessi valori negli attributi replicati. Il modello relazionale è **value-based** (basato su valori). Il modello non prevede l'uso di puntatori per rappresentare legami tra i dati. Essere value-based implica:

- Completa indipendenza dall'implementazione e dallo schema fisico
- Facilità di trasferimento dei dati tra sistemi
- Viene rappresentato solo ciò che è rilevante per il sistema informativo

### 5.4.1 Terminologia di schema di una relazione

E costituito dal nome della relazione e da un insieme di nomi per i suoi attributi:
$R(X)$ oppure $R(A_1, \ldots, A_n)$ oppure $R(A_1 : D_1, \ldots, A_n : D_n)$

### 5.4.2 Terminologia Schema di una base di dati

E costituito da un insieme di schemi di relazioni:
$S = \{R_1(X_1), \ldots, R_n(X_n)\}$ con $R_1 \neq R_2 \neq \ldots R_n$

### 5.4.3 Terminologia istanza di una relazione di schema $R(X)$

E un insieme r di tuple su X dove $r = \{t_1, \ldots, t_k\}$

### 5.4.4 Terminologia istanza di una base di dati di schema $S = \{R_1(X_1), \ldots, R_n(X_n)$

E costituito da un insieme di istanze delle relazioni $R_1(X_1), \ldots, R_n(X_n) : db = \{r_1, \ldots, r_n\}$

## 5.5 Valori nulli

Secondo la definizione formale di *relazione*, ogni tupla dovrebbe contenere per ciascun attributo un valore appartenente al relativo dominio. Nelle basi di dati reali, però, questa condizione non è sempre rispettabile: può accadere che un attributo non abbia un valore disponibile.

I motivi principali per cui un valore può mancare sono i seguenti:

- **Valore inesistente** L'attributo $A$ è concettualmente opzionale: per quella specifica istanza non esiste alcun valore significativo da inserire. *Esempio:* il campo "secondo numero di telefono" per un cliente che possiede un solo numero.
- **Valore sconosciuto** Il valore dell'attributo $A$ esiste, ma non è attualmente noto alla base di dati. *Esempio:* l'indirizzo e-mail di un utente appena registrato che non lo ha ancora comunicato.
- **Valore sconosciuto o inesistente** Non è possibile stabilire se l'informazione manchi perché non esiste o perché non è ancora conosciuta, oppure si preferisce rappresentare entrambe le situazioni allo stesso modo. *Esempio:* il campo "data di scadenza del contratto", che potrebbe non essere ancora stata definita oppure semplicemente non comunicata.

Per gestire queste situazioni si introduce il **valore nullo**. In questo modo il valore di un attributo in una tupla può essere:

- un valore appartenente al dominio dell'attributo;
- il `NULL`, utilizzato per rappresentare un'informazione assente (inesistente o sconosciuta).

*Esempio:* nel campo "numero di telefono fisso" di una tabella dei clienti, un cliente che non possiede un telefono fisso avrà come valore `NULL`.

Una tupla su X è una funzione:

$$t : X \rightarrow \boxed{\{NULL\}} \cup \left( \bigcup_{A \in X} DOM(A) \right)$$

dove:

$$t[A] = v \in DOM(A) \boxed{\vee\, t[A] = NULL}$$

Figura 24: Tupla su X con valori nulli

Si noti che:

- **La presenza di valori nulli è ammessa solo in alcuni attributi.** Non è infatti possibile avere tuple composte esclusivamente da valori nulli, poiché non rappresenterebbero alcuna informazione utile.
- **I valori nulli possono creare problemi negli attributi usati per rappresentare legami tra tuple.** In particolare, negli attributi che fungono da chiave esterna, un valore `NULL` potrebbe rendere impossibile stabilire la relazione con altre tuple.

*Esempio:* Si consideri una tabella `Prenotazioni` con l'attributo `idCliente` come chiave esterna verso la tabella `Clienti`. Se `idCliente` fosse `NULL`, non sarebbe possibile sapere quale cliente ha effettuato la prenotazione, rendendo l'informazione inutilizzabile.

## 5.6 Vincoli di integrità

Spesso è necessario introdurre dei **vincoli** (restrizioni) sull'istanza di una base di dati, poiché non tutte le configurazioni possibili sono compatibili con il sistema informativo che si vuole rappresentare.

Per specificare quali istanze sono considerate corrette si introduce il concetto di **vincolo di integrità**.

**Vincolo di Integrità**

Un **vincolo di integrità** è una condizione (espressa come un predicato) che deve essere sempre verificata da ogni istanza della base di dati. I vincoli di integrità assicurano che i dati memorizzati siano consistenti, coerenti e validi rispetto al modello del sistema informativo.

## 5.7 Esempi di vincoli

Siano date le seguenti relazioni:

- `TRENO(Numero, OraPart, MinutoPart, Categoria, Destinazione, OraArr, MinutoArr)`
- `FERMATA(NumTreno, Stazione, OraFer, MinutoFer)`

I seguenti predicati esprimono possibili **vincoli di integrità** sulla relazione `TRENO`:

$$\forall t \in TRENO: \ t[\text{OraPart}] \in \{0, 1, \ldots, 23\}$$

$$\forall t \in TRENO: \ t[\text{MinutoPart}] \in \{0, 1, \ldots, 59\}$$

$$\forall t \in TRENO: \ t[\text{Numero}] > 0$$

$$\forall t \in TRENO: \ t[\text{Numero}] > 5000 \ \Rightarrow \ t[\text{Categoria}] = \text{'regionale'}$$
$$\forall t, t' \in TRENO: \ t \neq t' \ \Rightarrow \ t[\text{Numero}] \neq t'[\text{Numero}]$$

*(Vincolo di unicità sul Numero del treno)*

$$\forall f \in FERMATA: \ \exists t \in TRENO: \ t[\text{Numero}] = f[\text{NumTreno}]$$

*(Vincolo di chiave esterna: ogni fermata deve riferirsi a un treno esistente)*

$$\forall t \in TRENO: \ t[\text{Categoria}] = \text{'regionale'} \Rightarrow \exists f \in FERMATA: \ f[\text{NumTreno}] = t[\text{Numero}]$$

*(Se un treno è regionale allora deve avere almeno una fermata)*

## 5.8 Classificazione vincoli di integrità

Si distinguono inoltre alcuni vincoli particolari che rivestono un ruolo fondamentale, poiché assicurano proprietà generali valide per tutti gli schemi relazionali. Tali vincoli sono detti **vincoli strutturali**.

Essi comprendono:

- **Vincoli di dominio** Specificano quali valori possono essere assegnati a ciascun attributo (esempio: l'attributo `OraPart` deve essere compreso tra 0 e 23).
- **Vincoli di tupla** Impongono condizioni che devono essere verificate all'interno di una singola tupla (esempio: `Numero` $> 0$).
- **Vincoli intrarelazionali** I vincoli intrarelazionali impongono condizioni sul contenuto di una relazione e richiedono che ogni tupla soddisfi una certa proprietà rispetto alle altre tuple della stessa relazione.

  Una categoria particolarmente rilevante di tali vincoli è costituita dai **vincoli di chiave**, che comprendono:

  - **Superchiave**: Data una relazione di schema $R(X)$, un insieme di attributi $K \subseteq X$ è una **superchiave** per $R(X)$ se, per ogni istanza $r$ di $R(X)$, vale la seguente condizione:

    $$\forall t, t' \in r : t \neq t' \ \Rightarrow \ t[K] \neq t'[K]$$

    dove:

    $$t[K] \neq t'[K] \ \equiv \ \exists A_i \in K : t[A_i] \neq t'[A_i]$$

- **Chiave candidata** : Data una relazione di schema $R(X)$, un insieme di attributi $K$, sottoinsieme di $X$, è **chiave candidata** (o **chiave**) per $R(X)$ se $K$ è superchiave per $R(X)$ e vale la seguente condizione:

$$\neg\exists K' \subset K : K' \text{ è superchiave per } R(X)$$

- **Chiave primaria** : Data una relazione di schema $R(X)$, la sua **chiave primaria** è la chiave candidata scelta per *identificare le tuple della relazione.*

  Una chiave primaria $K$ ha le seguenti caratteristiche:

  * Non contiene <u>mai valori nulli</u>.
  * Su $K$ il sistema genera una struttura d'accesso ai dati (o indice) per supportare le interrogazioni.

**Kristian semplifica così – Le tre chiavi spiegate intuitivamente**

**Superchiave** Una superchiave è **qualsiasi insieme di attributi che distingue sempre le tuple**. Se due tuple sono diverse, almeno un attributo della superchiave è diverso.
**Esempio:** Nella tabella STUDENTI(<u>Matricola</u>, Nome, Cognome, Email) - Insiemi come {Matricola}, {Matricola, Nome}, {Email, Cognome} sono tutte superchiavi. Perché? Perché non esistono due studenti con la stessa combinazione di quei valori.

**Chiave candidata** Una chiave candidata è una **superchiave minimale**: non puoi togliere nessun attributo senza perdere l'unicità.
**Esempio:** - {Matricola} è chiave candidata perché basta da sola. - {Email} è un'altra chiave candidata. - Ma {Matricola, Nome} **non** è candidata: contiene attributi inutili, è troppo "larga".

**Chiave primaria** È la **chiave candidata scelta** per identificare ufficialmente le tuple. Non può contenere NULL e il DB crea un indice su di essa.
**Esempio:** Se in STUDENTI scegliamo la chiave candidata {Matricola}, quella diventa la chiave primaria. Serve per identificare uno studente, fare ricerche, creare collegamenti con altre tabelle.

- **Vincoli interrelazionali** I vincoli interrelazionali impongono una restrizione al contenuto di una relazione e specificano una condizione che ogni tupla della relazione deve soddisfare rispetto alle tuple di altre relazioni della base di dati.Una sottocategoria importante di tali vincoli è costituita dai:

  - **Vincoli di integrità referenziale** impone che ogni valore di una *chiave esterna* (foreign key) presente in una relazione debba necessariamente comparire come valore di *chiave primaria* (primary key) nella relazione a cui fa riferimento. In questo modo si garantisce la coerenza dei dati tra più tabelle e si impedisce l'inserimento di tuple "orfane" non collegate a dati esistenti.

**Kristian semplifica in questo modo**

Il vincolo di integrità referenziale è una regola che assicura che due tabelle rimangano sempre coerenti tra loro.
**Significa:** se in una tabella inserisco una *chiave esterna*, quel valore deve esistere come *chiave primaria* nella tabella collegata.
**Esempio intuitivo:**

* Tabella `Studenti(id, nome)`
* Tabella `Esami(idStudente, voto)`

Nella tabella `Esami`, il campo `idStudente` è una **foreign key**: può contenere solo valori che esistono come `id` nella tabella `Studenti`.
Se provo a inserire un esame per lo studente `id = 50`, ma nella tabella `Studenti` non esiste lo studente 50, il database blocca l'inserimento perché violerei il vincolo di integrità referenziale.

## 5.9 Notazione per indicare i vincoli strutturali

### 5.9.1 Chiave primaria

La chiave primaria di una relazione $R(X)$ viene indicata **sottolineando gli attributi** che la compongono.

Supponiamo che, per la tabella `TRENO`, venga scelto come chiave primaria l'attributo `Numero`. La notazione per indicare tale chiave è:

`TRENO(`<u>`Numero`</u>`, OraPart, MinutoPart, Categoria, Destinazione, OraArr, MinutoArr)`

Per la tabella `FERMATA`, la chiave primaria è composta dagli attributi `NumTreno`, `Stazione`, `OraFer` e `MinutoFer`. Si indica così:

`FERMATA(`<u>`NumTreno, Stazione`</u>`, OraFer, MinutoFer)`

### 5.9.2 Vincolo di integrità referenziale

Un **vincolo di integrità referenziale** viene indicato **riquadrando gli attributi soggetti al vincolo** (nella tabella figlia) e collegandoli con una freccia al riquadro degli attributi della tabella da cui la chiave primaria proviene (tabella padre).

Supponiamo che nella tabella `FERMATA` l'attributo `NumTreno` sia una chiave esterna che fa riferimento alla chiave primaria `Numero` della tabella `TRENO`.

La notazione diventa:

`Numero` ⟵ `TRENO(`<u>`Numero`</u>`, OraPart, MinutoPart, Categoria, Destinazione, OraArr, MinutoArr`

`NumTreno` `FERMATA(`<u>`NumTreno, Stazione`</u>`, OraFer, MinutoFer)`

**Kristian semplifica in questo modo**

**Chiave primaria**: sottolineo gli attributi che identificano univocamente le tuple della tabella.
**Chiave esterna (integrità referenziale)**: metto un riquadro attorno all'attributo nella tabella "figlia" e lo collego con una freccia alla chiave primaria della tabella "padre".
In questo modo si vede subito da dove arriva il valore e quale tabella controlla la sua validità.

## 5.10 Esercizio

**2.7** Individuare le chiavi e i vincoli di integrità referenziale che sussistono nella base di dati di Figura 2.23 e che è ragionevole assumere siano soddisfatti da tutte le basi di dati sullo stesso schema. Individuare anche gli attributi sui quali possa essere sensato ammettere valori nulli.

Esercizio 2.7 del libro di testo.

PAZIENTI

| Cod. | Cognome | Nome |
|---|---|---|
| A102 | Necchi | Luca |
| B372 | Rossini | Piero |
| B543 | Missoni | Nadia |
| B444 | Missoni | Luigi |
| S555 | Rossetti | Gino |

RICOVERI

| Paz. | Inizio | Fine | Rep. |
|---|---|---|---|
| A102 | 2/05/11 | 9/05/11 | A |
| A102 | 2/12/11 | 2/01/12 | A |
| S555 | 25/04/11 | 3/05/11 | B |
| B444 | 1/12/11 | 2/01/12 | B |
| S555 | 5/10/11 | 1/11/11 | A |

MEDICI

| Matr. | Cogn. | Nome | Rep. |
|---|---|---|---|
| 203 | Neri | Piero | A |
| 574 | Bisi | Mario | B |
| 461 | Bargio | Sergio | B |
| 530 | Belli | Nicola | C |
| 405 | Mizzi | Nicola | A |
| 501 | Monti | Mario | A |

REPARTI

| Cod. | Nome | Primario |
|---|---|---|
| A | Chirurgia | 203 |
| B | Pediatria | 574 |
| C | Medicina | 530 |

**Figura 2.23** Una base di dati per l'Esercizio 2.6 e 2.7.

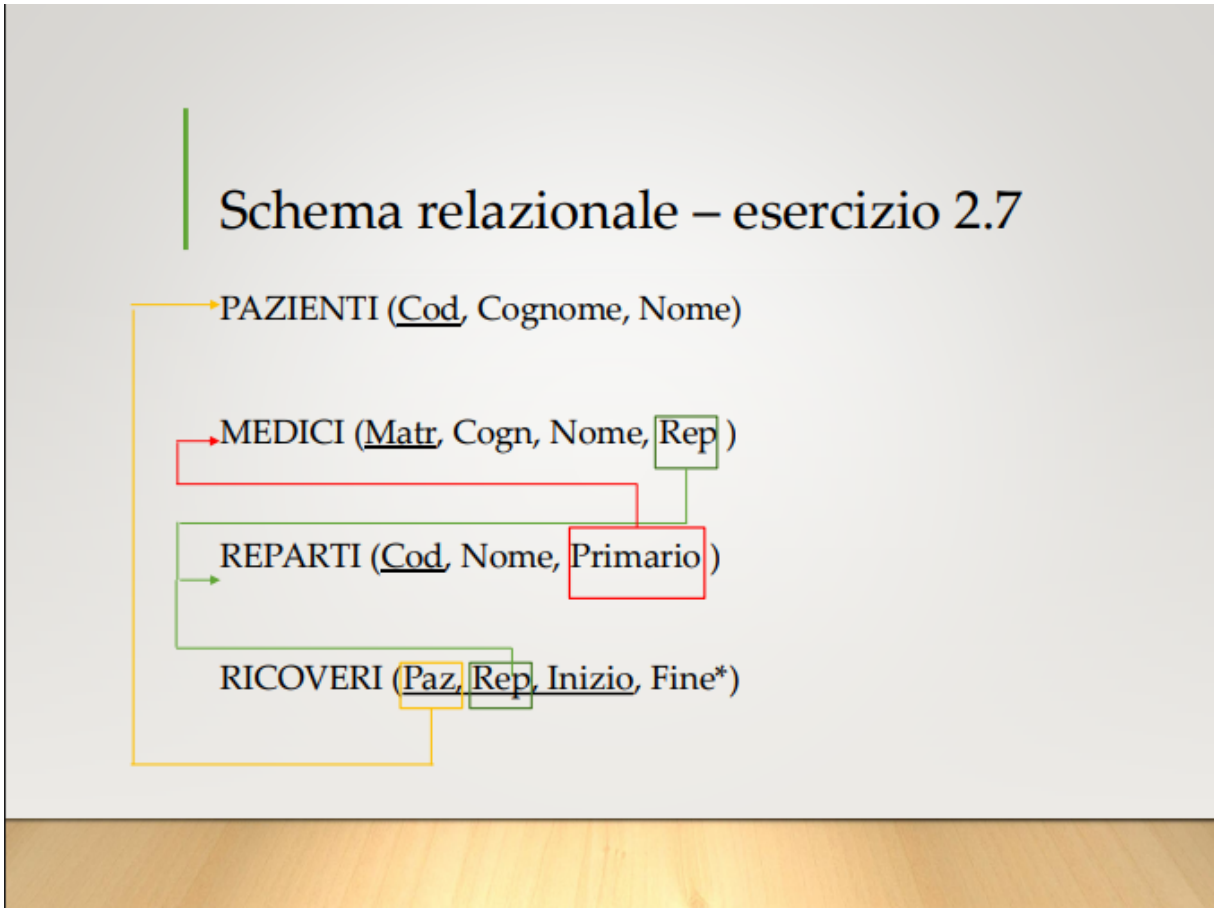



Figura 25: Soluzione

## 5.11 Progettazione Logica

L'obiettivo della progettazione logica è quello di produrre uno **schema logico** che rappresenti in modo corretto, completo ed efficace tutte le informazioni contenute nello **schema concettuale**.

Prima della traduzione può essere necessario **ristrutturare lo schema ER** per i seguenti motivi:

- **Semplificare** la fase di traduzione verso il modello logico.
- **Ottimizzare** le strutture in base al *carico di lavoro* previsto (operazioni e query più frequenti).

Le principali fasi della **ristrutturazione dello schema ER** sono:

- Analisi delle **ridondanze** dovute alla presenza di dati derivabili.
- **Eliminazione delle generalizzazioni** tramite le diverse strategie di trasformazione.
- **Accorpamento** o **partizionamento** di entità e relazioni, quando necessario.
- Scelta degli **identificatori principali** più adatti ed efficienti.

### 5.11.1 Analisi della derivabilità

L'analisi della derivabilità serve a individuare **dati che possono essere ricavati (derivati) da altri dati già presenti nello schema**. Questi dati non vanno memorizzati perché occupano

spazio inutilmente e possono creare inconsistenze.

Esempi di dati derivabili:

- **Attributi calcolabili** a partire da altri attributi della stessa entità (es. "età" derivabile dalla "data di nascita").
- **Attributi derivabili** da informazioni presenti in entità diverse (es. "numero totale degli ordini di un cliente" ricavabile dalla relazione *Ordini*).
- **Attributi ottenuti con un conteggio** (es. "numero di iscritti a un corso" ricavabile contando le partecipazioni).
- **Relazioni derivabili** dalla combinazione di altre relazioni (es. una relazione "LavoraIn" può essere derivata combinando "Impiegato" e "Dipartimento" tramite una relazione intermedia).

### 5.11.2 Analisi delle ridondanze

In questa fase vengono analizzati i **dati derivabili** per decidere se conviene:

- **Memorizzare il dato derivabile**, quando il costo del calcolo è *maggiore* del costo di aggiornamento.
- **Non memorizzarlo e calcolarlo quando serve**, quando il costo del calcolo è *minore* del costo di aggiornamento.

Il **costo** viene misurato in termini di numero di accessi per unità di tempo, valutando:

- la frequenza con cui il dato deve essere calcolato,
- la frequenza con cui il dato dovrebbe essere aggiornato.

### 5.11.3 Eliminazione delle Generalizzazioni

Nel modello relazionale le generalizzazioni non sono direttamente rappresentabili. Per questo motivo esistono tre possibili strategie per trasformarle:

- **Accorpamento delle entità figlie nel padre**.
- **Accorpamento dell'entità padre nelle entità figlie**.
- **Sostituzione della generalizzazione con relazioni**.

### 5.11.4 Accorpamento delle entità figlie nel padre

Questa soluzione prevede:

- Eliminazione delle entità figlie.
- Trasferimento nel padre di tutti gli attributi specifici delle entità figlie, trattandoli come **attributi opzionali**.
- Accorpamento nel padre di tutte le relazioni che coinvolgevano le entità figlie, impostando la **cardinalità minima a 0** dal lato del padre.
- Aggiunta di un attributo **TIPO** per distinguere le occorrenze corrispondenti alle ex entità figlie.

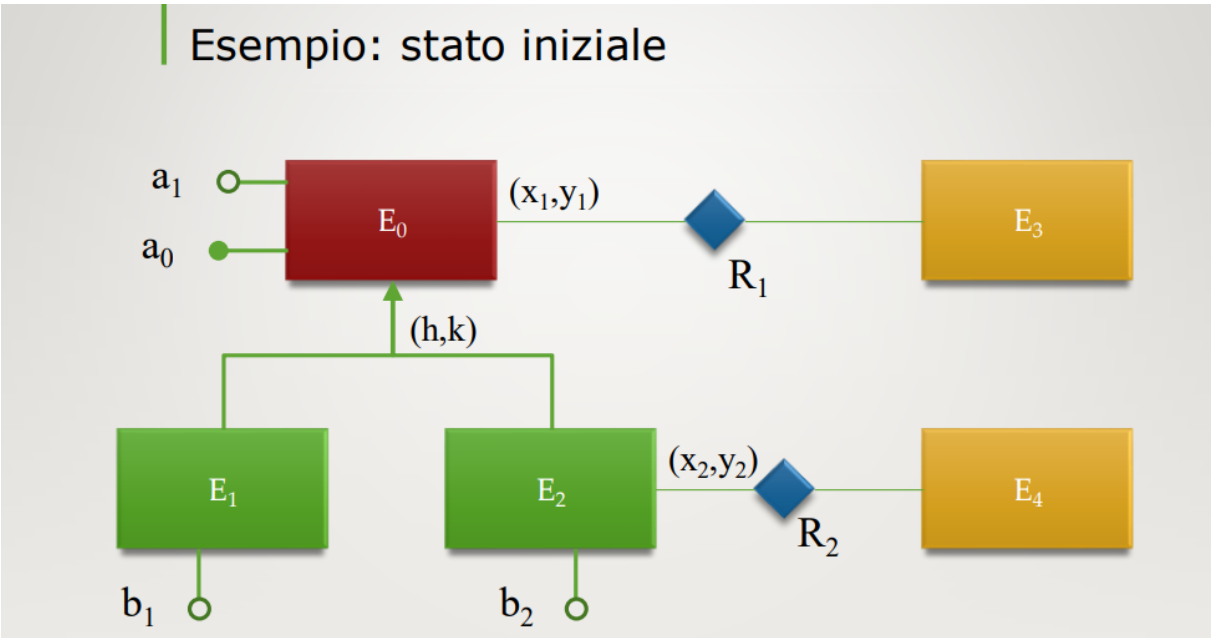



Figura 26: Stato iniziale

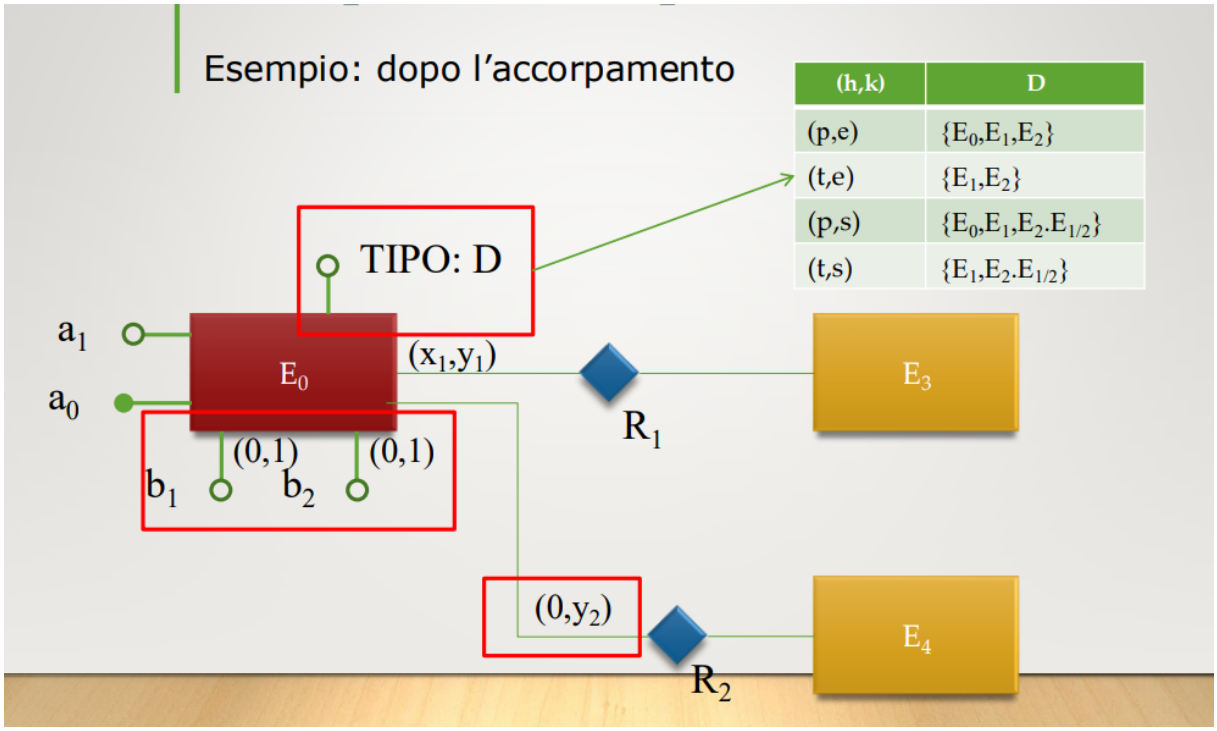



Figura 27: Accorpamento delle entità figlie nel padre

### 5.11.5 Accorpamento del padre nelle entità figlie

Questa trasformazione può essere applicata solo quando la generalizzazione è **totale**. La procedura consiste nei seguenti passi:

- Eliminazione dell'entità padre.

- Replicazione degli attributi dell'entità padre in ciascuna entità figlia.
- Partizionamento, tra le entità figlie, delle relazioni che coinvolgevano l'entità padre, assegnando **cardinalità minima pari a 0** dal lato delle entità esterne.

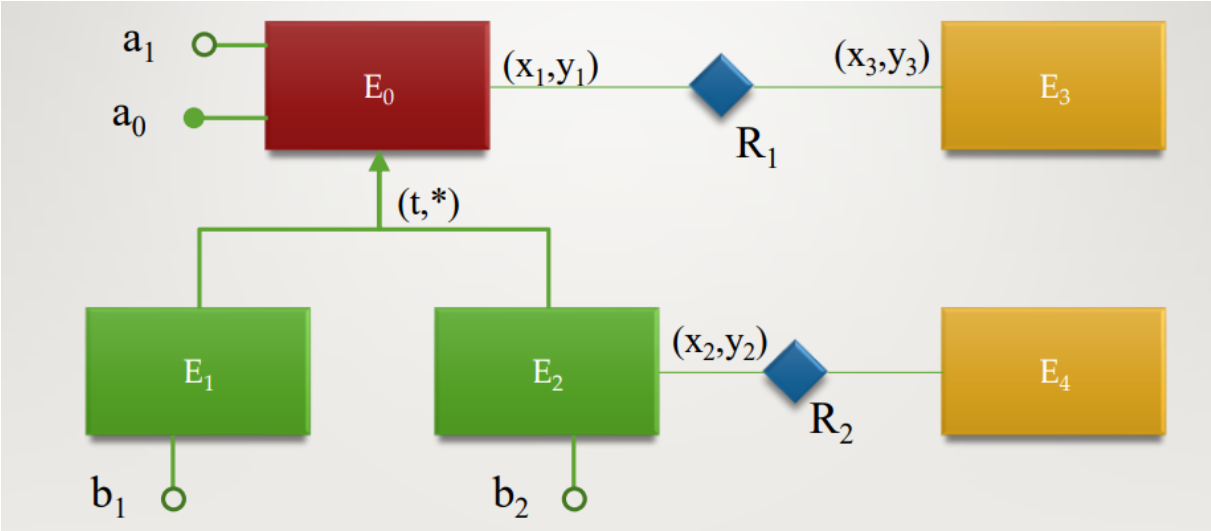



Figura 28: Schema iniziale

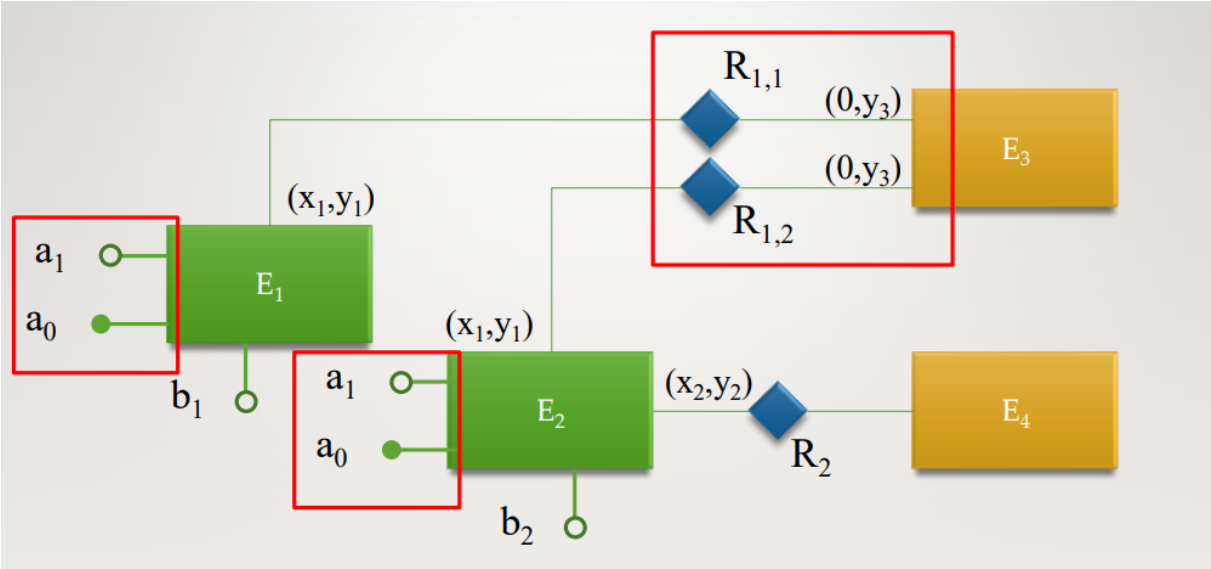



Figura 29: Accorpamento nelle entità figlie

### 5.11.6 Sostituzione con relazioni

Questa strategia sostituisce la generalizzazione con un insieme di relazioni esplicite. I passi da seguire sono:

- Eliminazione della generalizzazione.
- Inserimento di $n$ relazioni tra l'entità padre e ciascuna delle $n$ entità figlie.

- Mantenimento degli attributi nelle entità in cui si trovavano originalmente, senza spostamenti.

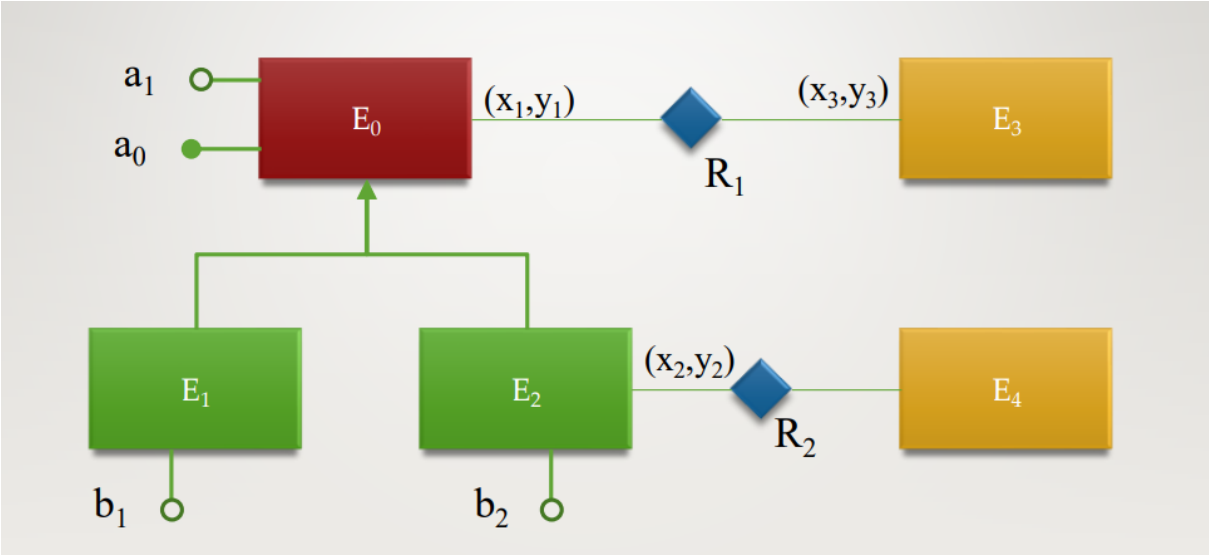



Figura 30: Schema iniziale

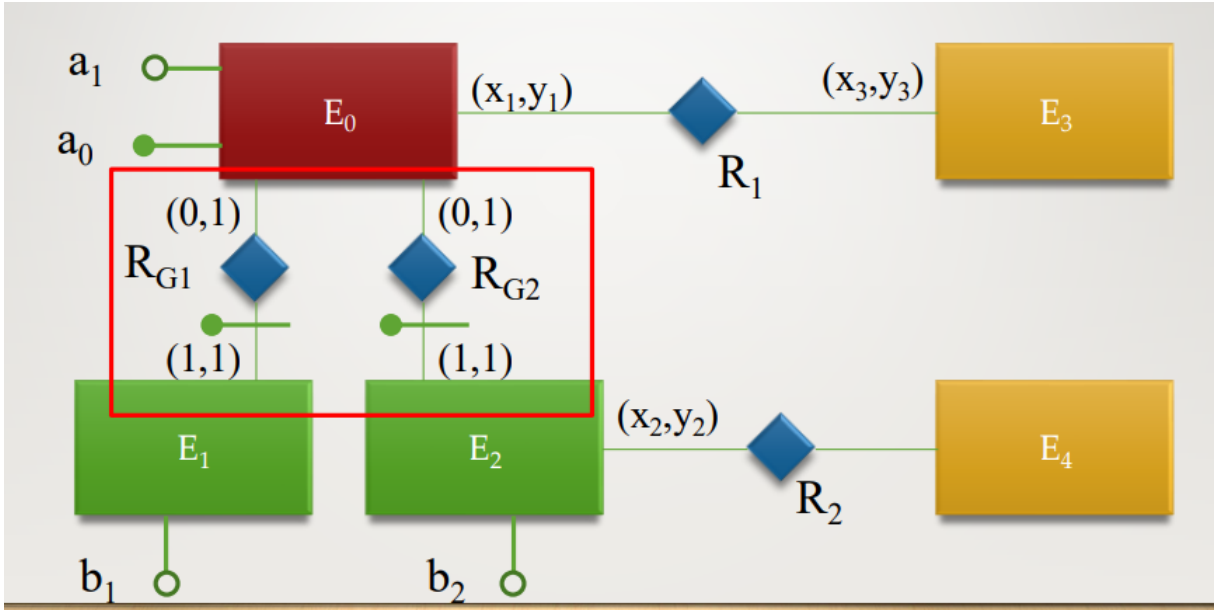



Figura 31: Sostituzione della generalizzazione con relazioni

## 5.12 Traduzione verso il modello relazionale

La traduzione verso il modello relazionale è un processo automatico che applica allo **schema concettuale ristrutturato** un insieme di regole di trasformazione. Ogni regola si applica a uno specifico costrutto del modello concettuale ER e produce una o più strutture del modello relazionale.

I principi fondamentali della traduzione sono:

- Un'**istanza di entità** viene rappresentata mediante una **tupla** della tabella che implementa l'entità.
- Un'**istanza di relazione** viene rappresentata tramite:
  - un **legame tra tuple** (mediante chiavi esterne), oppure
  - una **tupla esplicita** in una tabella dedicata.
- L'**identificatore principale** di un'entità viene rappresentato come **chiave primaria** della tabella che implementa tale entità.

## 5.13 Classificazione delle relazioni binarie

Una relazione binaria del modello ER può assumere tre forme, in base alle **cardinalità massime** delle entità coinvolte:

- **Uno a Uno (1:1)**: entrambe le entità coinvolte nella relazione hanno cardinalità massima pari a 1.
- **Uno a Molti (1:N)**: una delle entità ha cardinalità massima pari a 1, mentre l'altra ha cardinalità massima maggiore di 1.
- **Molti a Molti (N:M)**: entrambe le entità hanno cardinalità massima maggiore di 1.

## 5.14 Regole di traduzione

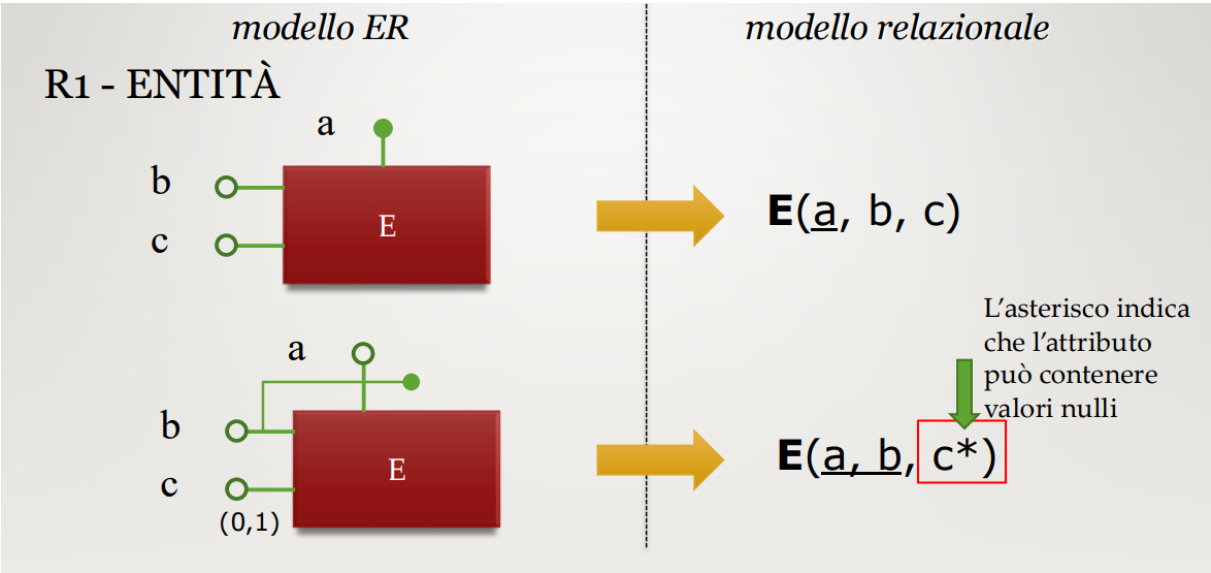



Figura 32: Entità

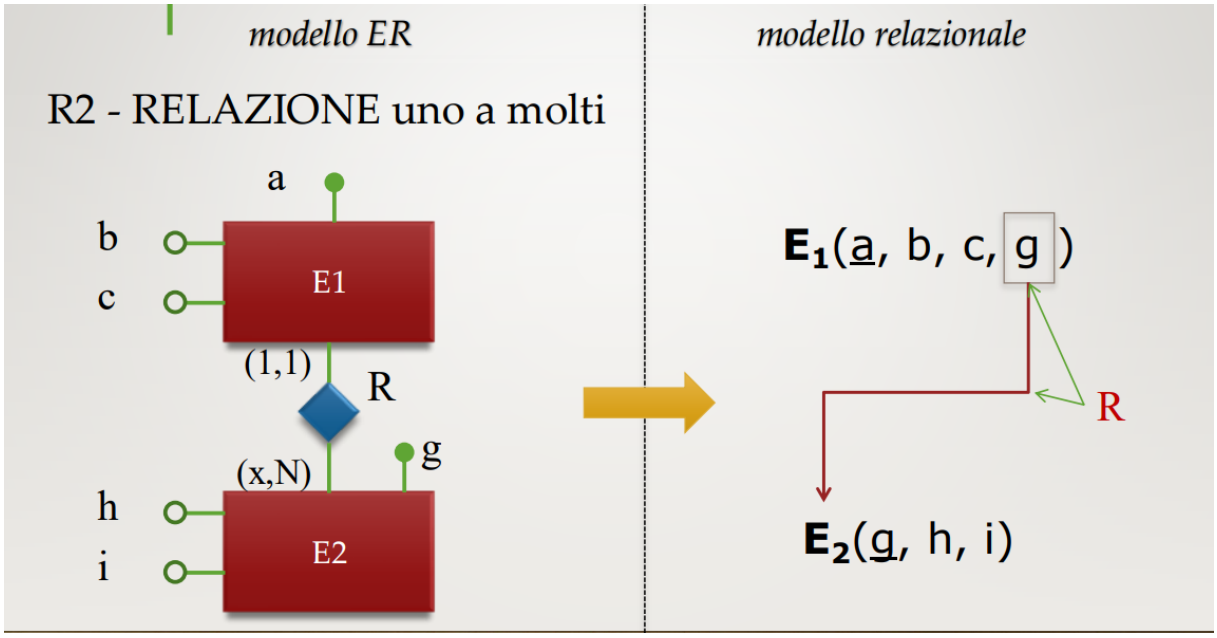



Figura 33: Relazione uno a molti

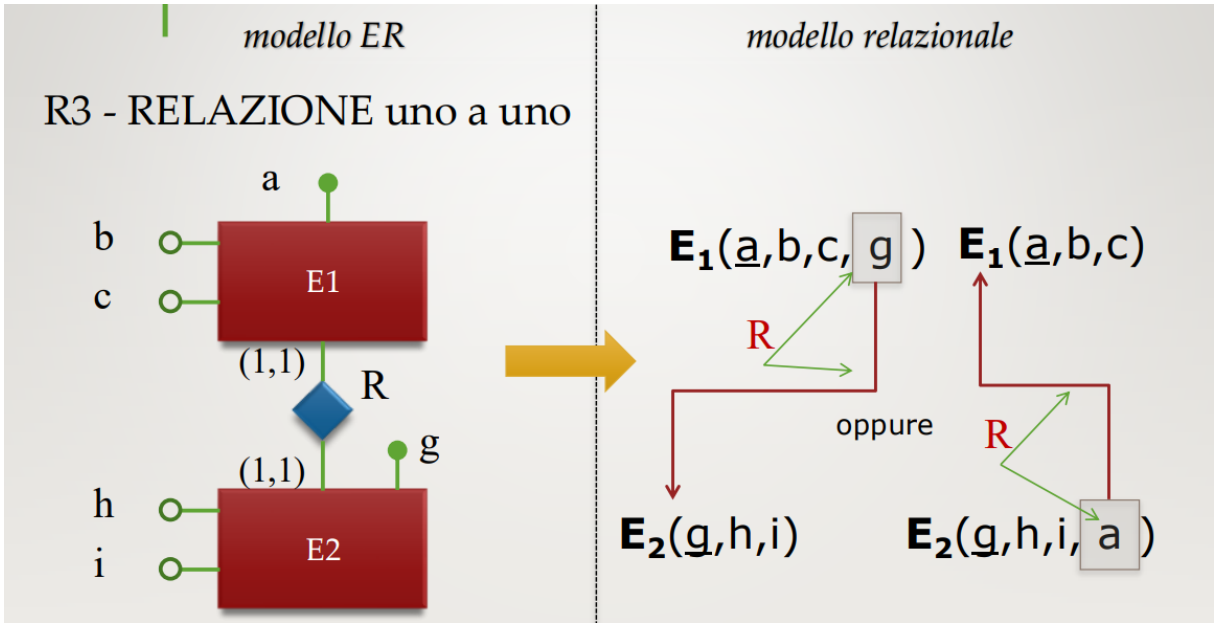



Figura 34: Relazione uno a uno

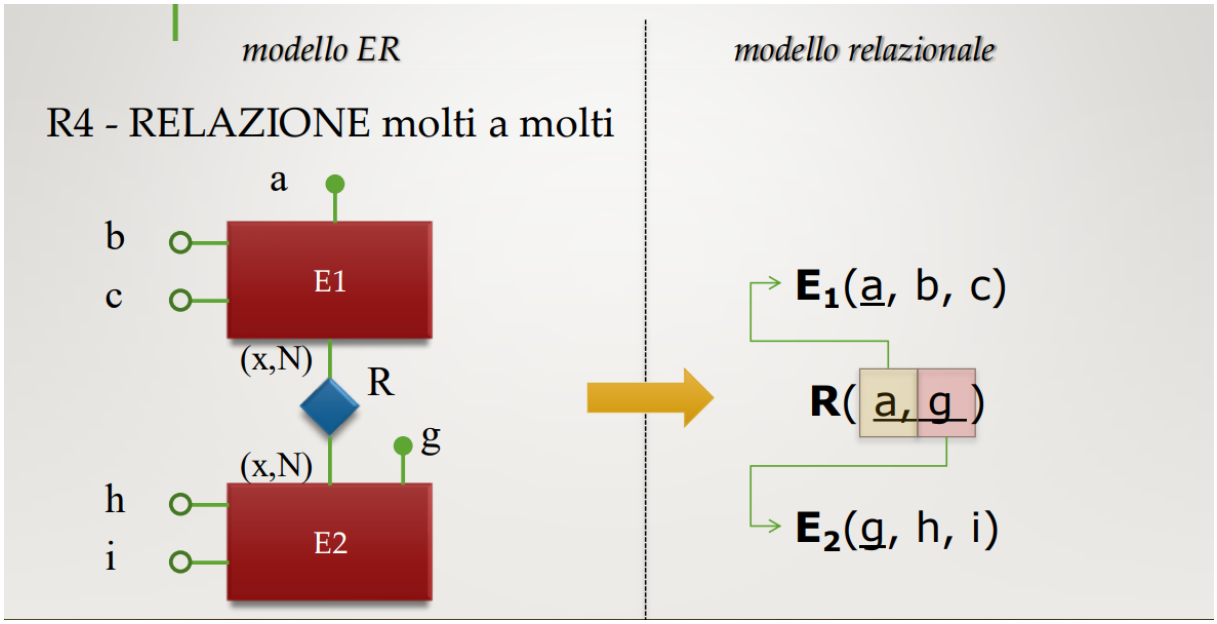



Figura 35: Relazione molti a molti

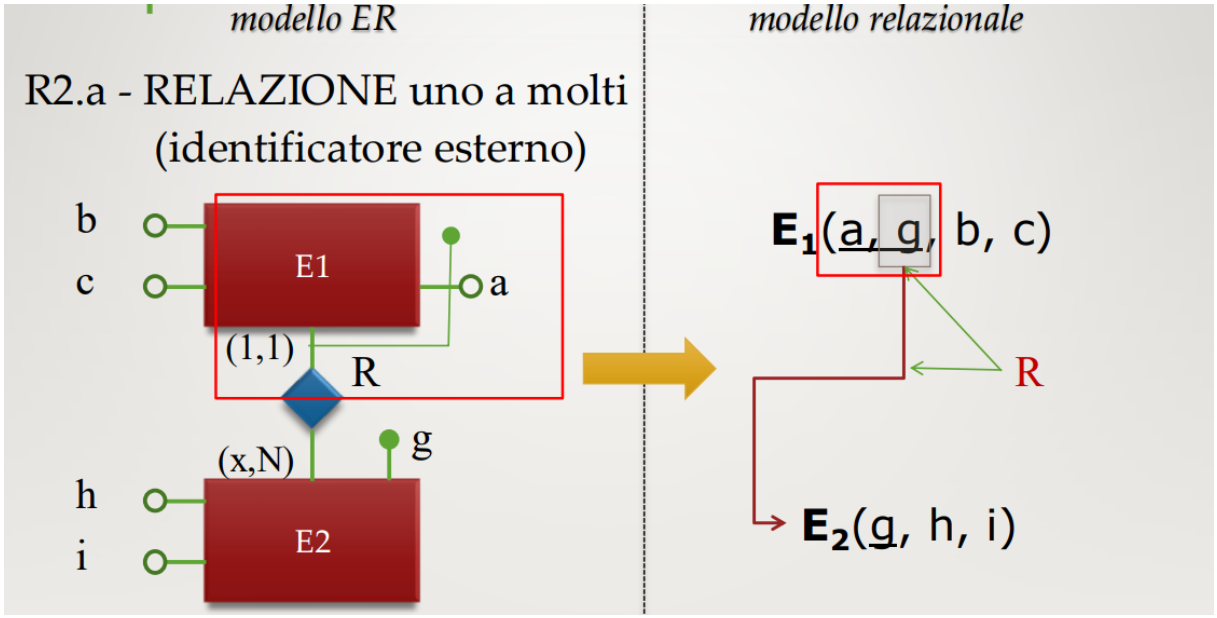



Figura 36: Relazione uno a molti identificatore esterno

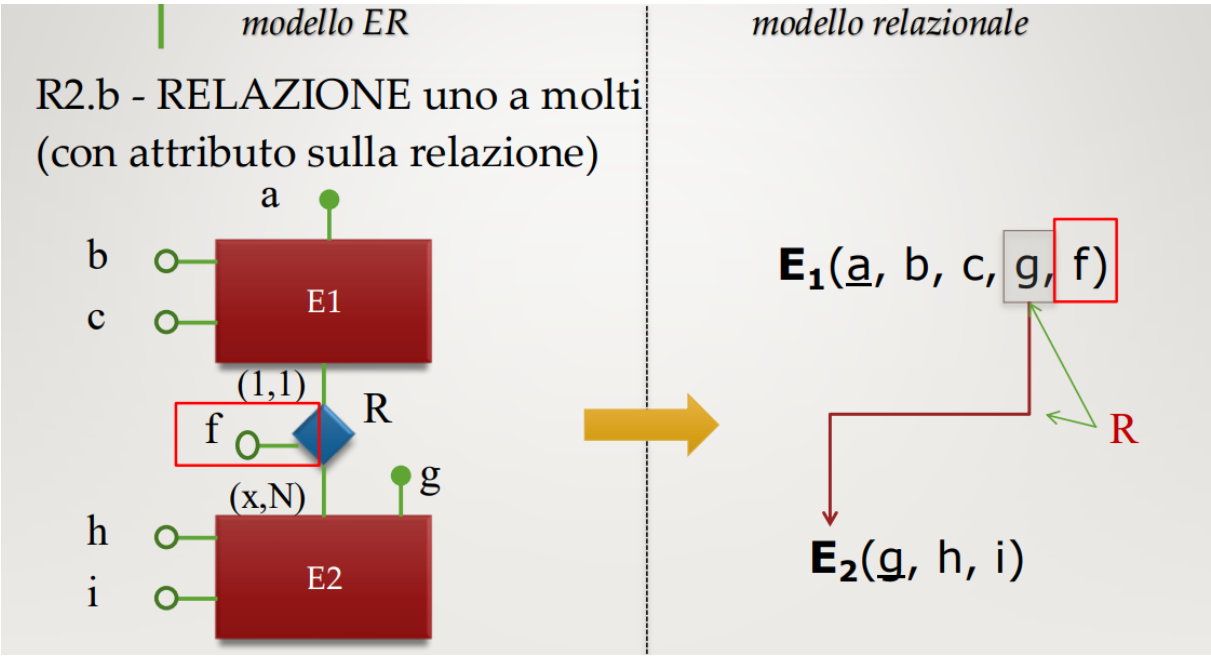



Figura 37: Relazione uno a molti con attributo sulla relazione

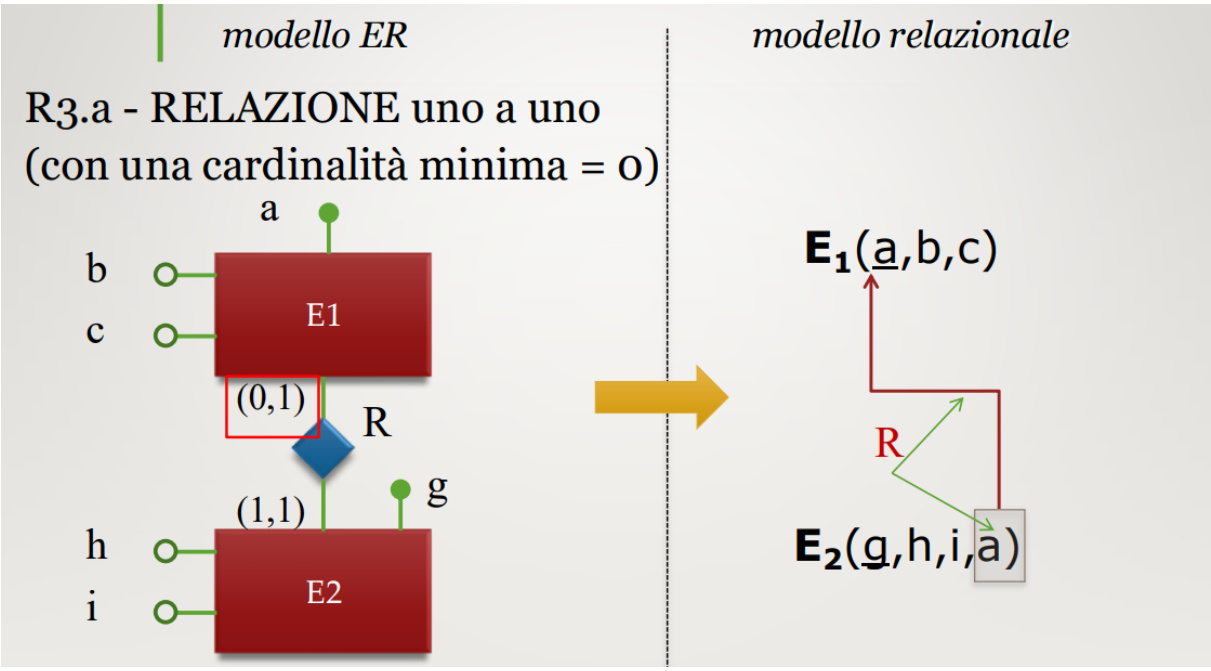



Figura 38: RELAZIONE uno a uno (con una cardinalità minima = 0)

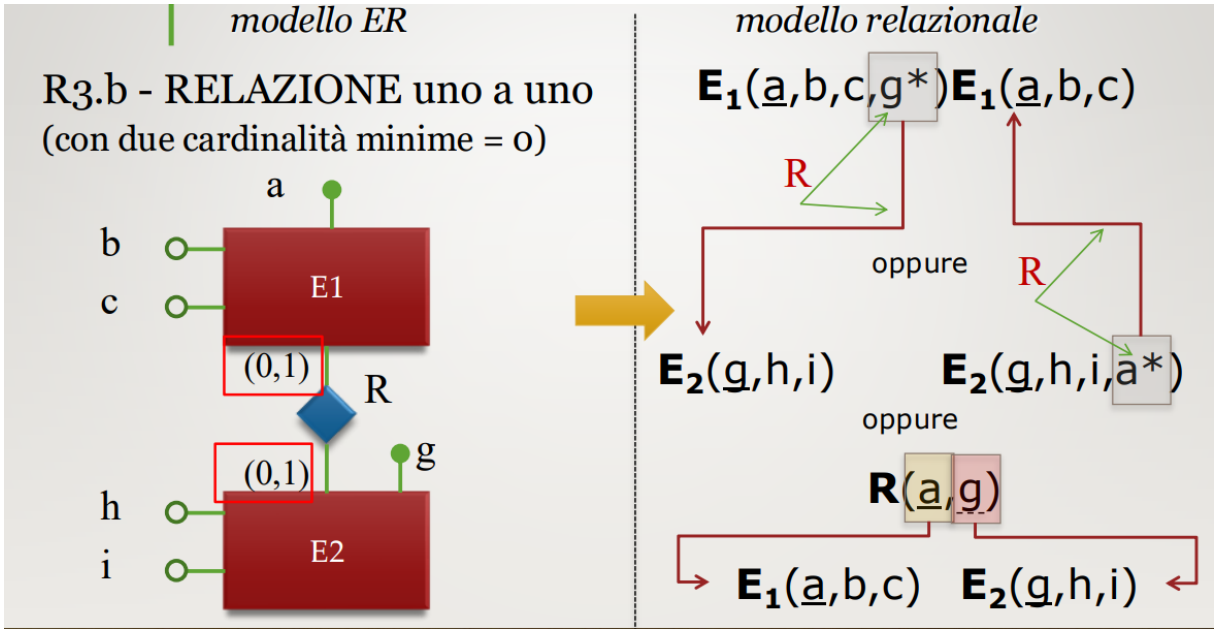



Figura 39: RELAZIONE uno a uno (con due cardinalità minime = 0)

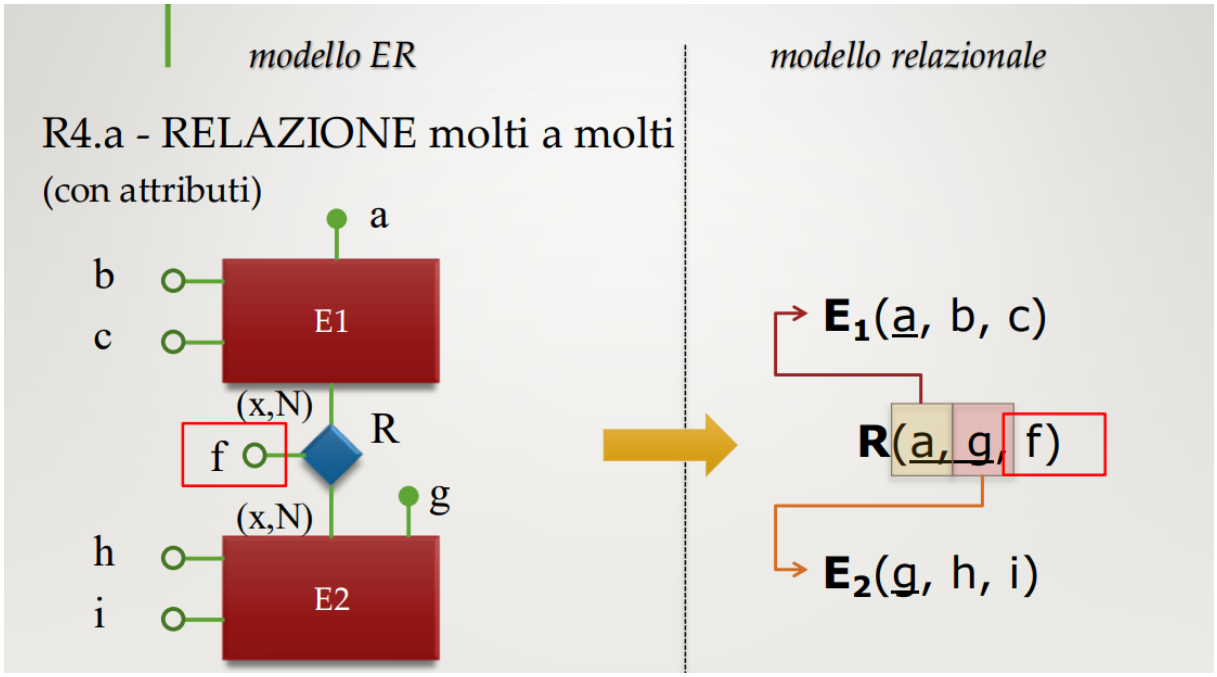



Figura 40: - RELAZIONE molti a molti (con attributi)

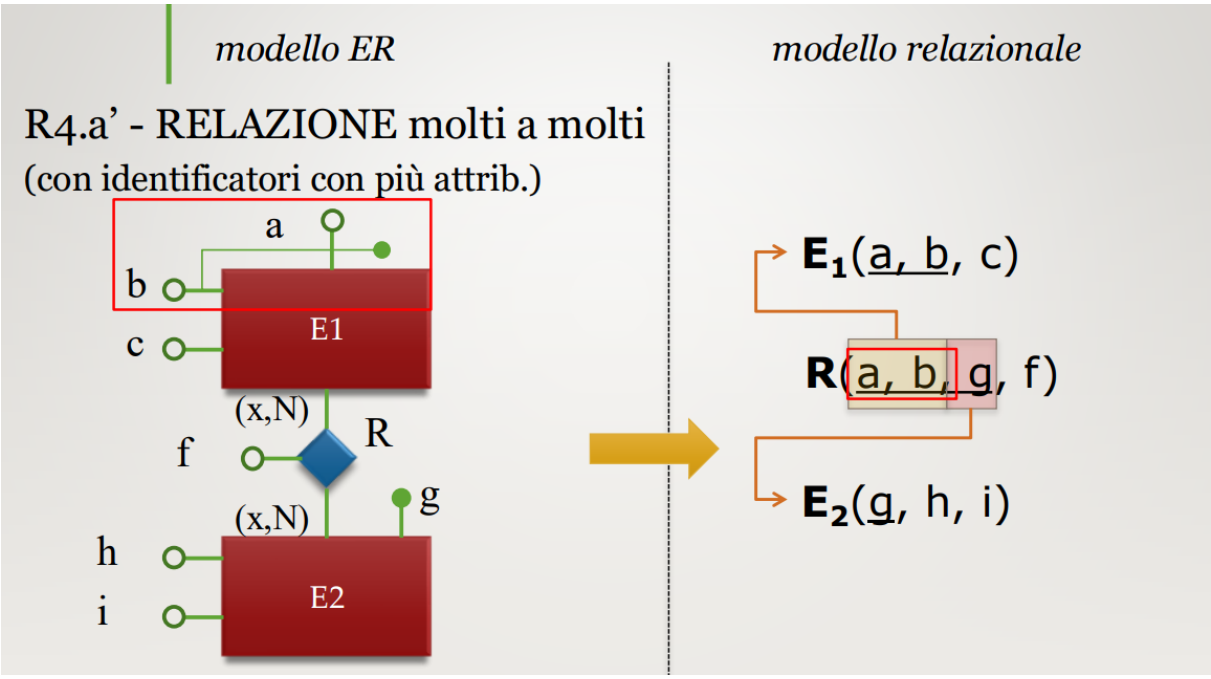



Figura 41: RELAZIONE molti a molti (con identificatori con più attrib.)

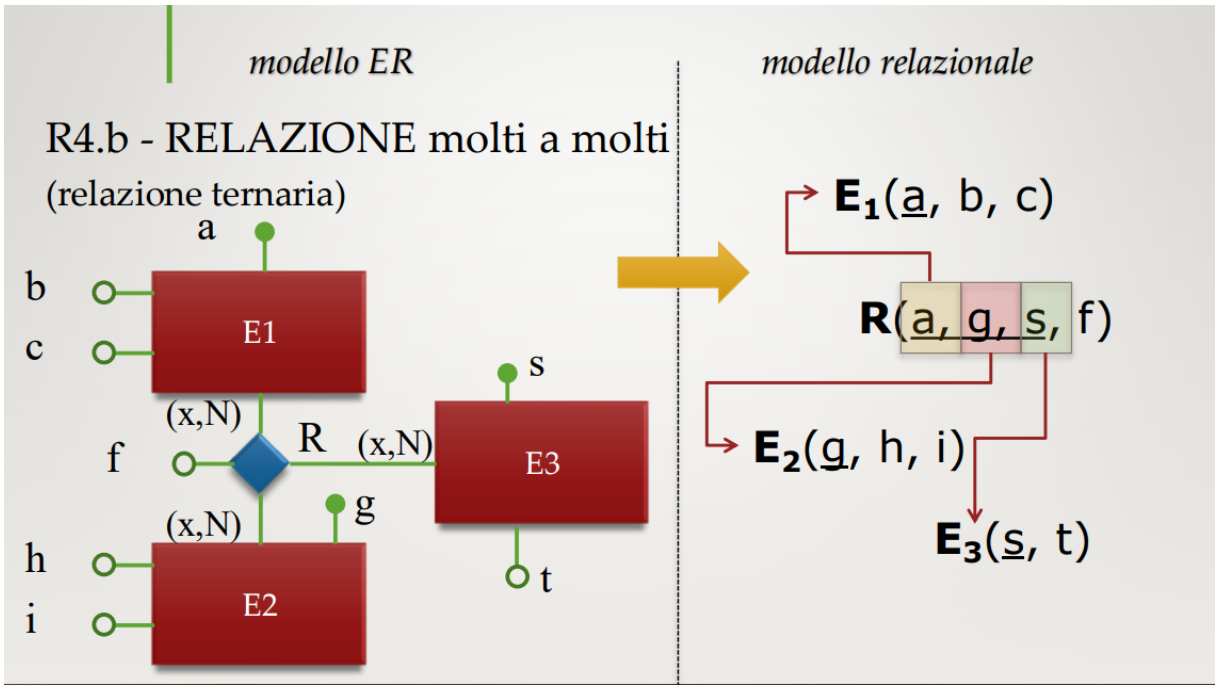



Figura 42: RELAZIONE molti a molti (relazione ternaria)

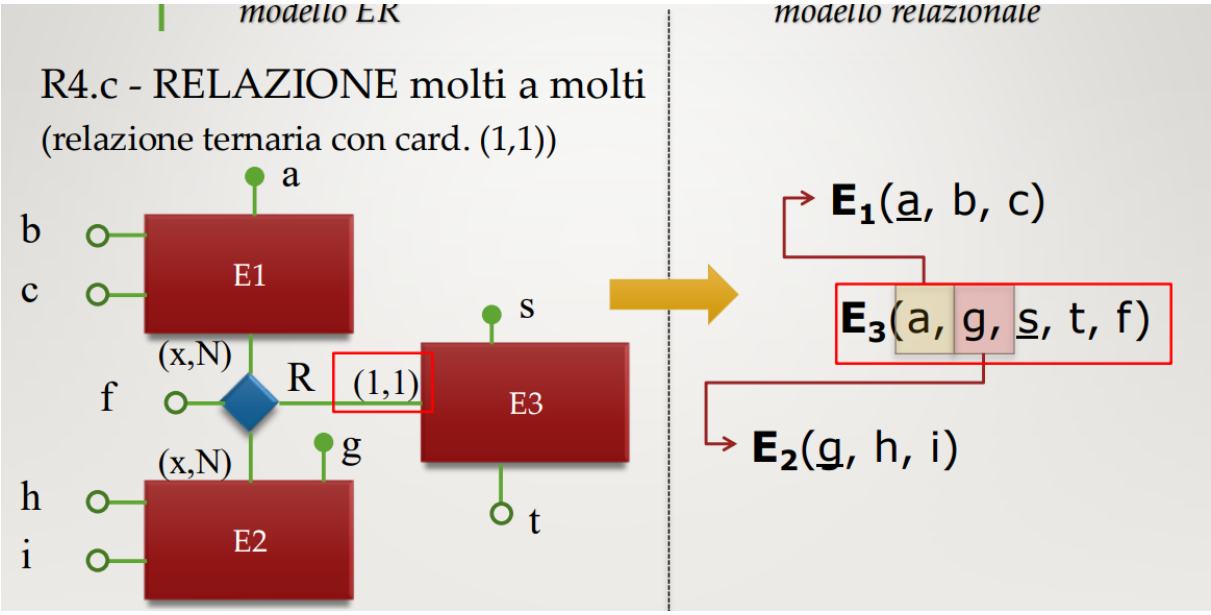



Figura 43: RELAZIONE molti a molti (relazione ternaria con card. (1,1))

# 6 Tecnologie NoSQL: Sistemi Document-Based

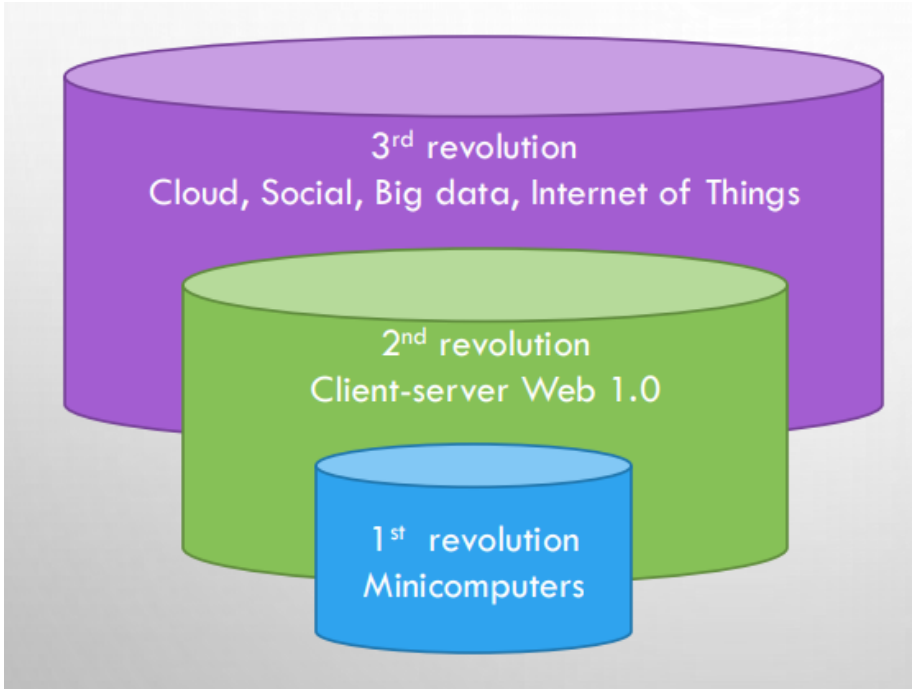



Figura 44: Evoluzione Database

La figura mostra l'evoluzione storica delle tecnologie di database attraverso tre grandi rivoluzioni, rappresentate come livelli sovrapposti. Ogni rivoluzione introduce nuovi modelli, nuovi paradigmi e nuove esigenze dovute ai cambiamenti tecnologici del periodo.

## Prima rivoluzione (1950–1960)

Alla base si trova la prima rivoluzione dei database, nata nell'epoca dei **minicomputer**. In questo periodo emergono i primi sistemi per la gestione dei dati, caratterizzati da strutture rigide e fortemente dipendenti dall'organizzazione fisica dei file. Le principali tecnologie di questo periodo sono:

- database **gerarchici**;
- database **reticolari** (network databases);
- file organizzati tramite **ISAM** (Indexed Sequential Access Method).

## Seconda rivoluzione (1970–1990)

Il secondo livello corrisponde alla diffusione del modello **relazionale**, che nasce negli anni '70 grazie ai lavori di E. F. Codd. In questa fase il paradigma relazionale si impone come standard grazie alla sua semplicità, astrazione e indipendenza dalla struttura fisica dei dati. È l'epoca dei sistemi **client-server** e del Web 1.0.

### Terza rivoluzione (2005–oggi)

La terza rivoluzione si sviluppa nell'era del **cloud**, dei **big data** e dell'**Internet of Things**. In questo contesto il volume, la varietà e la velocità dei dati aumentano notevolmente, rendendo necessarie nuove soluzioni oltre al modello relazionale classico. Accanto ai database tradizionali si diffondono:

- sistemi **NoSQL**;
- sistemi **NewSQL**;
- piattaforme di **Big Data**.

Questa rivoluzione rappresenta un ampliamento, non una sostituzione: i database relazionali restano fondamentali, ma vengono affiancati da tecnologie progettate per gestire grandi moli di dati distribuiti.

## 6.1 Approcci NoSQL

I modelli NoSQL nascono per rispondere ai limiti dei tradizionali database relazionali in scenari caratterizzati da grandi quantità di dati, alta eterogeneità e necessità di scalare orizzontalmente. Questi sistemi adottano strutture più flessibili e meno vincolate rispetto allo schema rigido delle tabelle relazionali, permettendo una gestione più efficiente di dati distribuiti su larga scala. Tra gli approcci più diffusi troviamo:

- **Sistemi Key-Value**: tutti i dati sono rappresentati tramite coppie (chiave, valore), dove la chiave identifica le istanze e il valore può avere una qualsiasi struttura, anche non omogenea. Un esempio di questo approccio è Hadoop.
- **Sistemi Document-Store**: i dati sono rappresentati come collezioni di documenti con struttura complessa e senza schema fisso. Esempi: **MongoDB**, **CouchBase**.
- **Column Store**: i dati sono rappresentati tramite record con struttura variabile. Esempi: **BigTable**, **HBase**, **Cassandra**.
- **Graph-Store**: i dati sono rappresentati come grafi, con nodi e archi. Esempi: **Neo4j**, **OrientDB**.

## 6.2 Sistemi Document-Store

Caratteristiche principali:

- i dati sono organizzati come collezioni di **documenti**;
- ogni documento può contenere dati complessi e nidificati;
- l'**incapsulamento** è fondamentale: tutte le informazioni relative a un'entità stanno nello stesso documento;
- non è richiesto uno schema fisso: i documenti di una collezione possono avere strutture diverse.

### Progettazione dei dati come documenti

Decisioni fondamentali:

- scelta della chiave di accesso;
- struttura interna dei documenti e livello di incapsulamento;

- rappresentazione delle relazioni tra documenti.

Le relazioni possono essere modellate tramite:

- **riferimenti** (ID dell'altro documento);
- **documenti nidificati**.

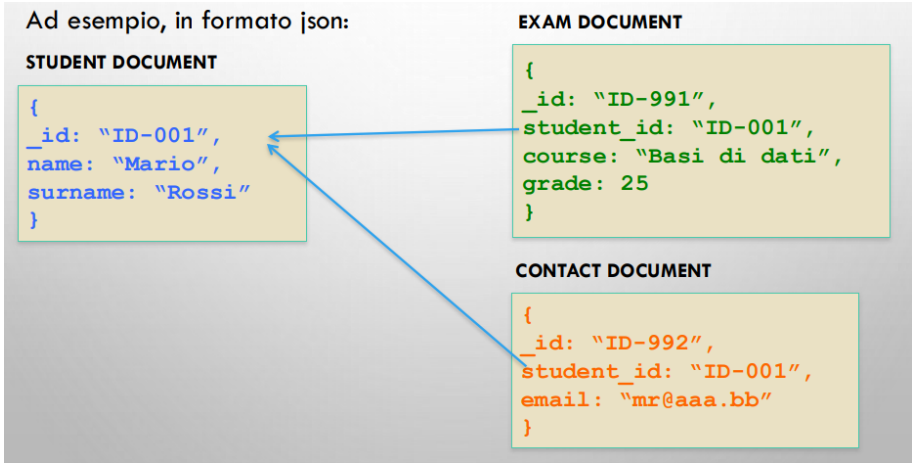



Figura 45: Formato JSON

## 6.3 Regole di traduzione di uno schema ER in collezioni di documenti

Assunzioni preliminari:

- si adotta un approccio **embedded**;
- lo schema ER è privo di generalizzazioni;
- relazioni ternarie o superiori eliminate;
- identificate collezioni $\{DOC_1, ..., DOC_N\}$.

### Regole fondamentali

1. Ogni collezione $DOC_i$ corrisponde a un'entità principale.
2. Ogni istanza dell'entità principale produce un documento.
3. Le entità non principali vengono incapsulate nella principale.
4. Le relazioni sono rappresentate tramite incapsulamento o riferimenti.

### 6.3.1 Prima Fase (entità principali)

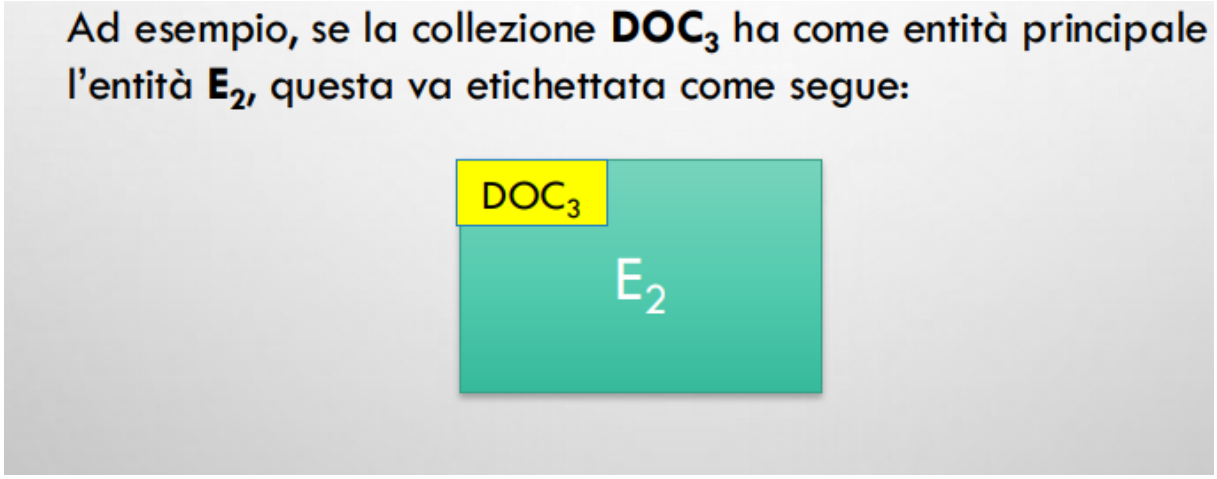



Figura 46: Individuazione entità principali

### 6.3.2 Seconda Fase (incapsulamento entità non principali)

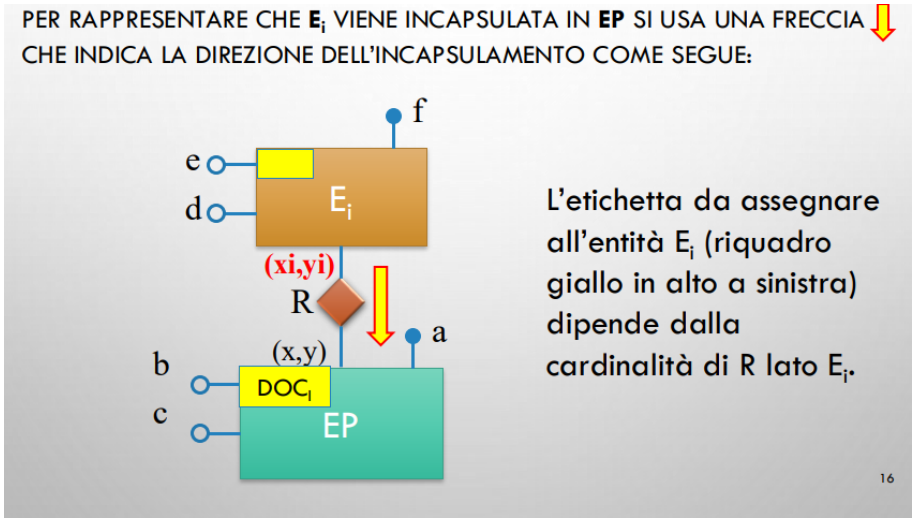



Figura 47: Incapsulamento

## Diversi Casi

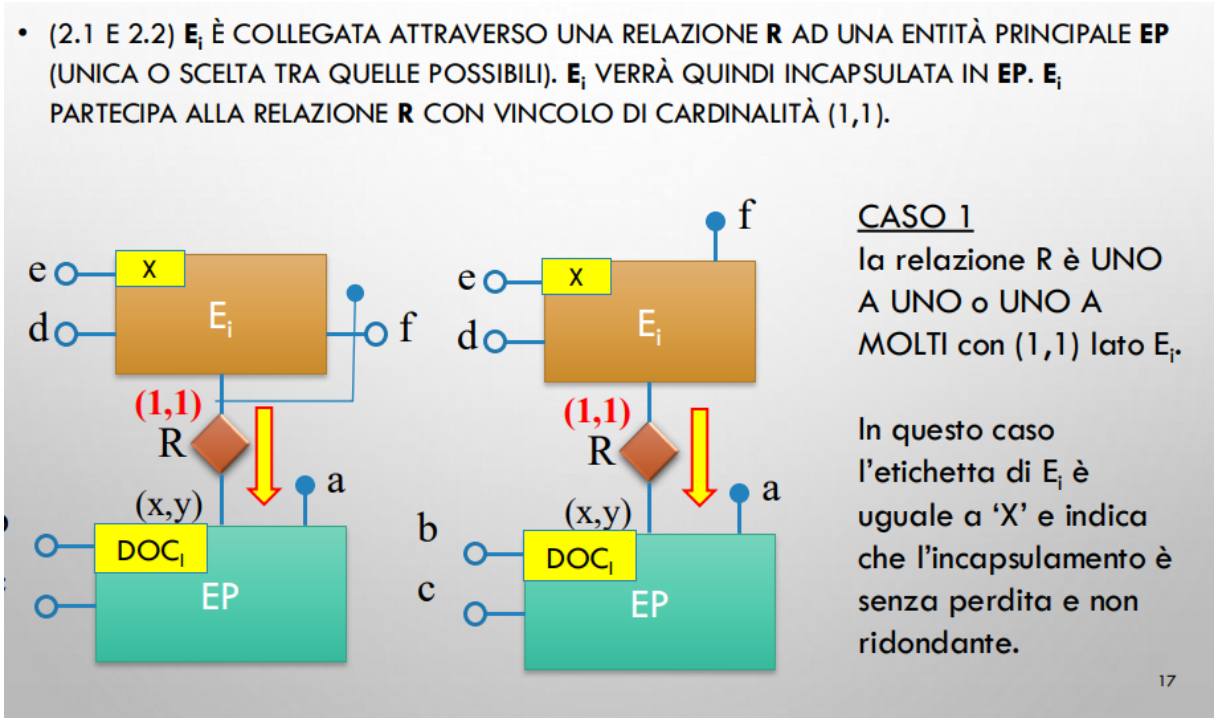



Figura 48: Uno a uno o uno a molti con (1,1)

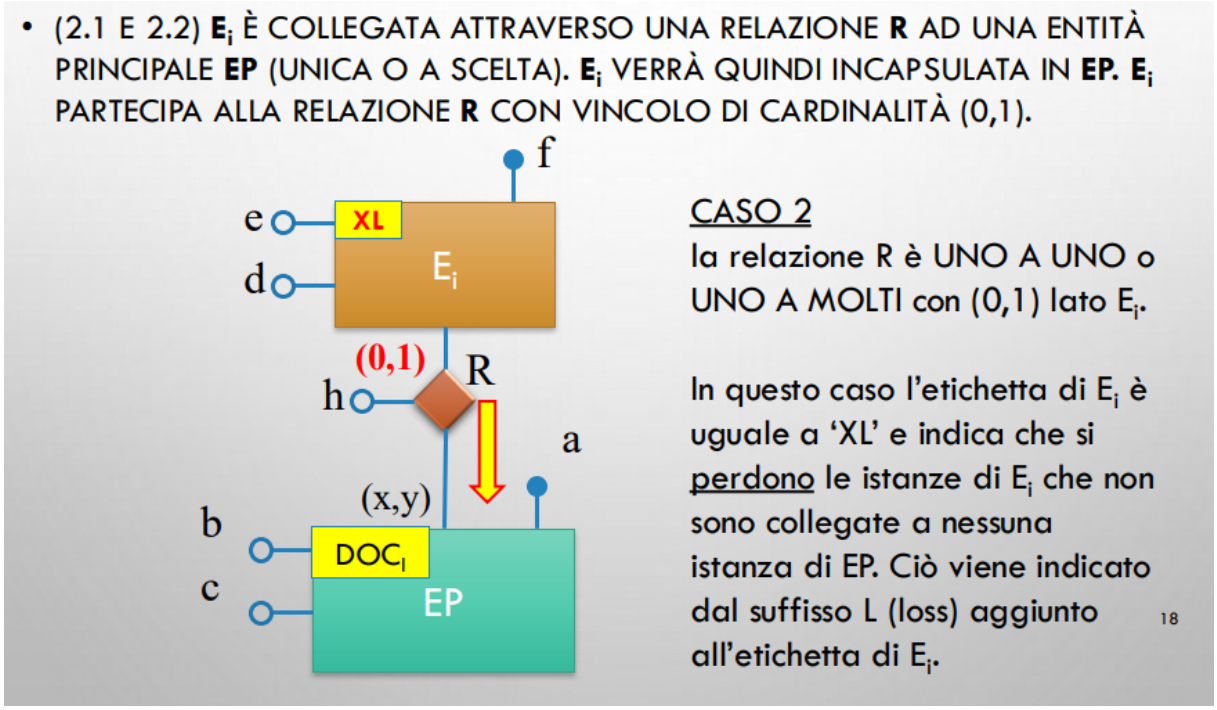
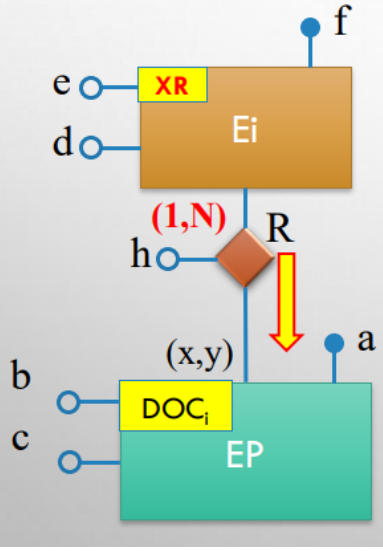



Figura 49: Uno a uno o uno a molti con (0,1)

**SECONDA FASE(INCAPSULAMENTO DELLE ENTITA' NON PRINCIPALI)**

- (2.1 E 2.2) **$E_i$** È COLLEGATA ATTRAVERSO UNA RELAZIONE **R** AD UNA ENTITÀ PRINCIPALE **EP** (UNICA O A SCELTA). **$E_i$** VERRÀ QUINDI INCAPSULATA IN **EP**. **$E_i$** PARTECIPA ALLA RELAZIONE **R** CON VINCOLO DI CARDINALITÀ (1,N).



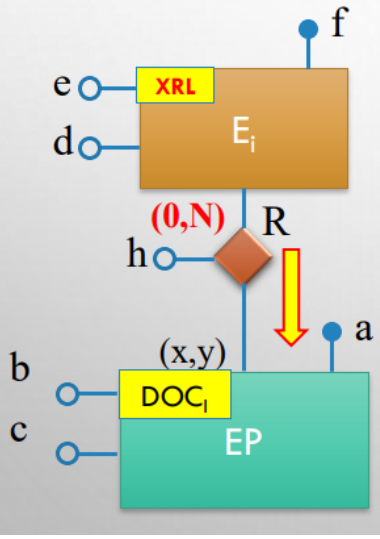

CASO 3
la relazione R è MOLTI A MOLTI o MOLTI A UNO con (1,N) lato $E_i$.

In questo caso le istanze di $E_i$ vengono rappresentate in modo ridondante in quanto sono collegate a più istanze di EP. Ciò viene indicato dal suffisso R (redundant) aggiunto all'etichetta di $E_i$.

19

Figura 50: Molti a molti o molti a uno con (1,N)

- (2.1 E 2.2) **$E_i$** È COLLEGATA ATTRAVERSO UNA RELAZIONE **R** AD UNA ENTITÀ PRINCIPALE **EP** (UNICA O A SCELTA). **$E_i$** VERRÀ QUINDI INCAPSULATA IN **EP**. **$E_i$** PARTECIPA ALLA RELAZIONE **R** CON VINCOLO DI CARDINALITÀ (0,N).




CASO 4
la relazione R è MOLTI A MOLTI o MOLTI A UNO con (0,N) lato $E_i$.

In questo caso le istanze di $E_i$ vengono rappresentate in modo ridondante in quanto sono collegate a più istanze di EP e si perdono quelle non collegate. Ciò viene indicato dall'aggiunta del suffisso RL all'etichetta di $E_i$.

20

Figura 51: Molti a molti o molti a uno con (0,N) lato Ei

### 6.3.3 Terza Fase (relazioni non strutturali)

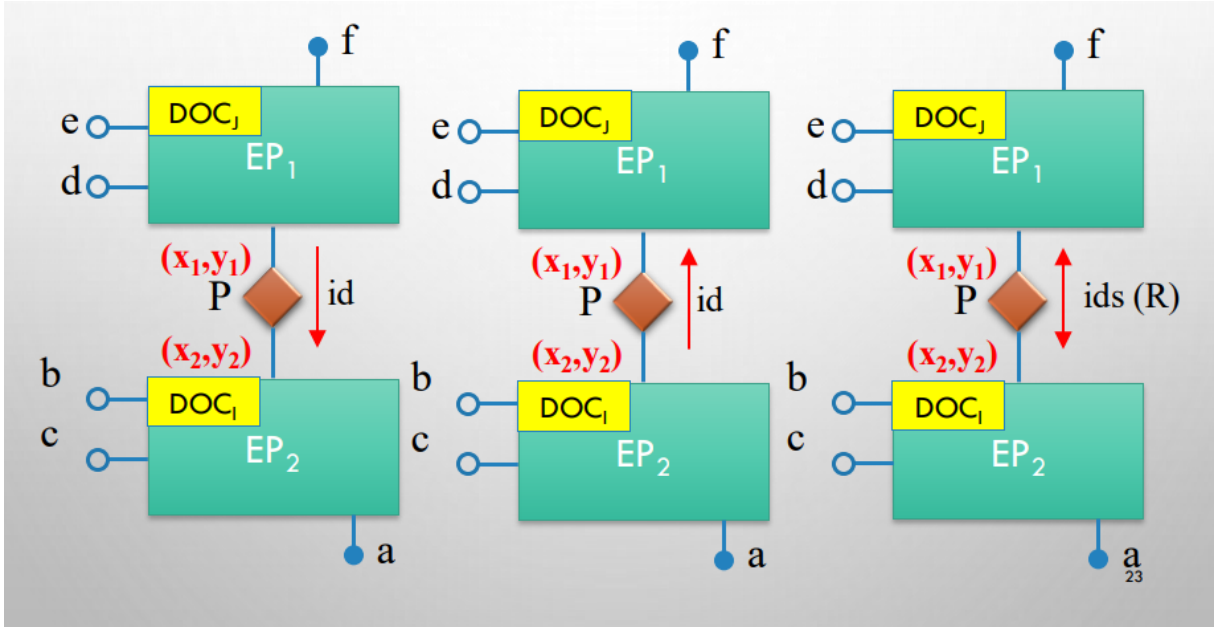



Figura 52: Etichettatura dello schema ER

#### Casi particolari

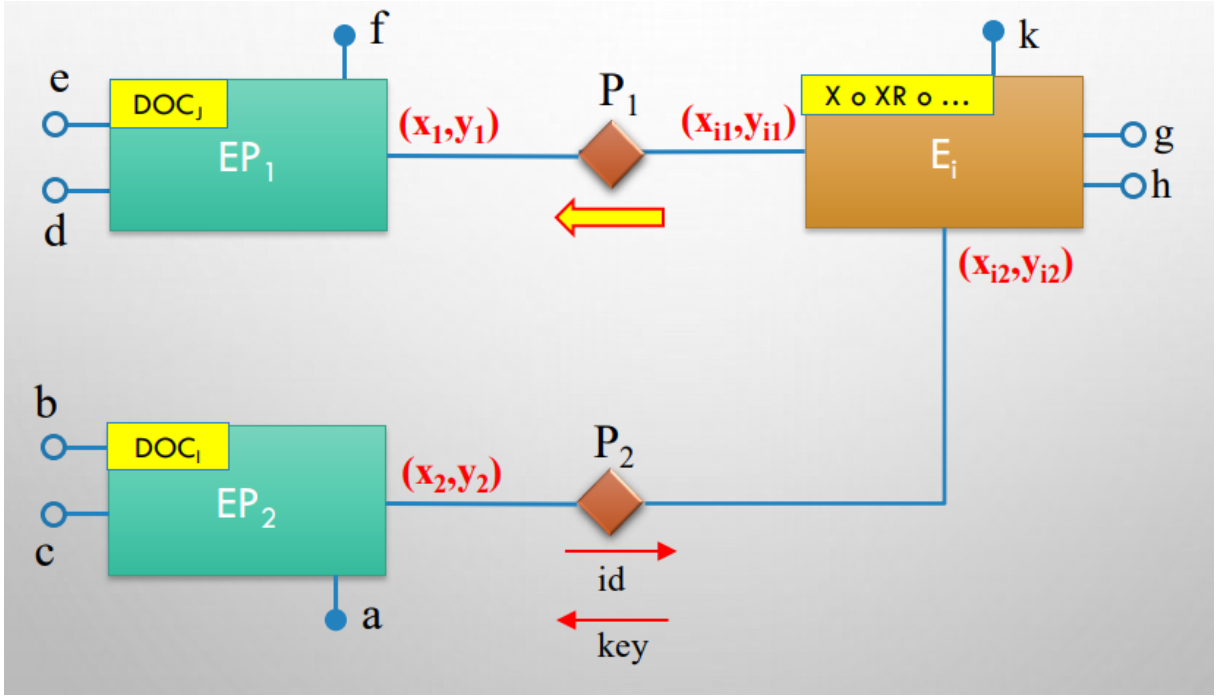



Figura 53: Caso Particolare A

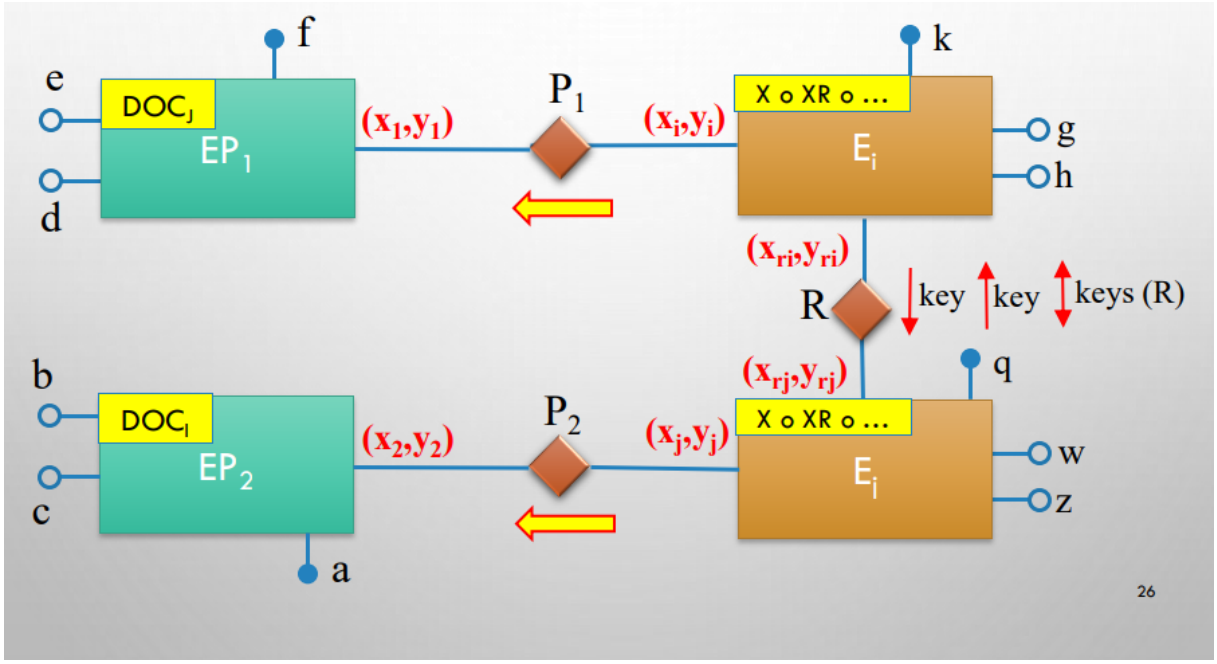



Figura 54: Caso Particolare B

## 6.4 Quarta fase

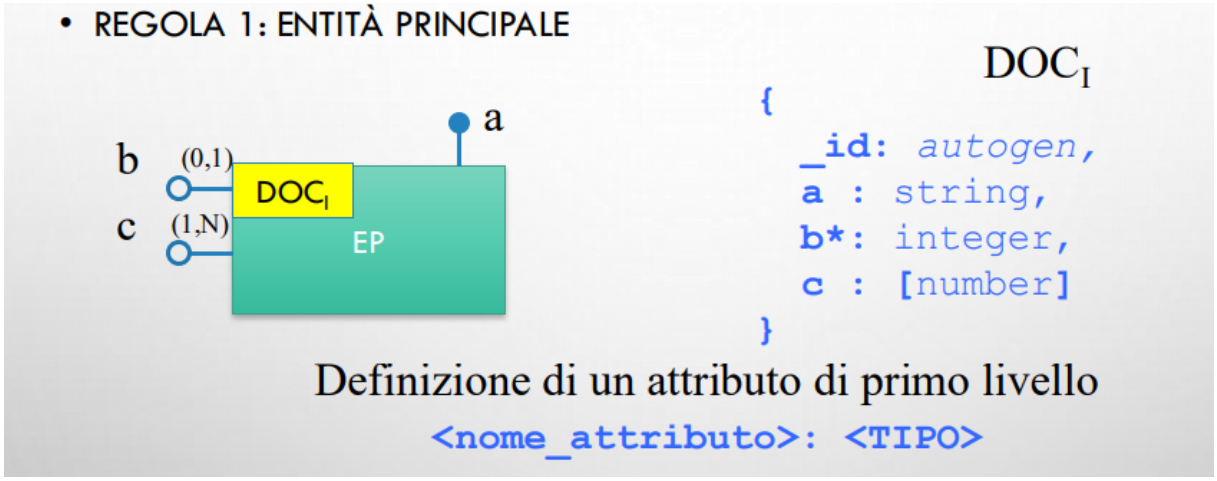
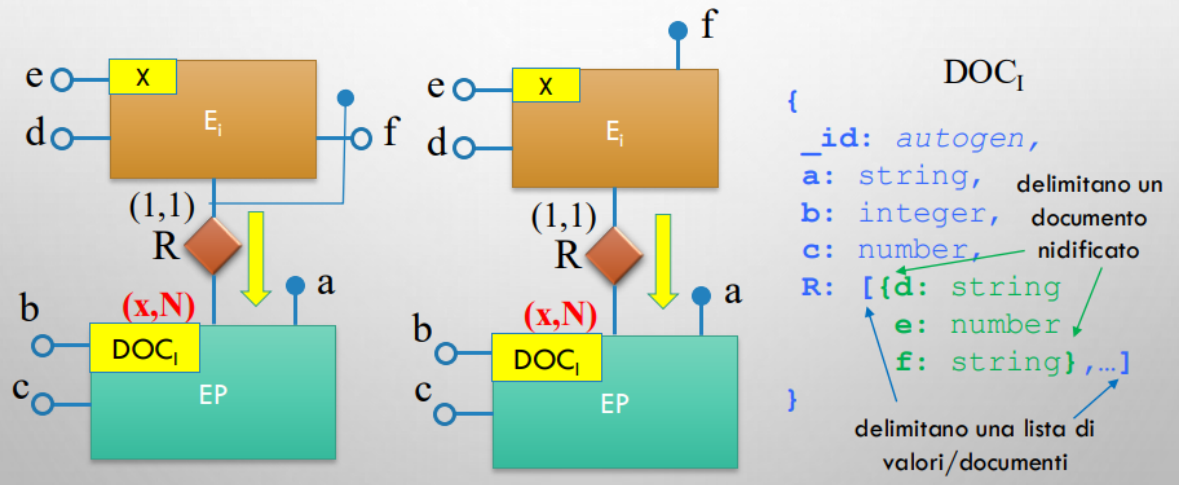



Figura 55: Entità Principale

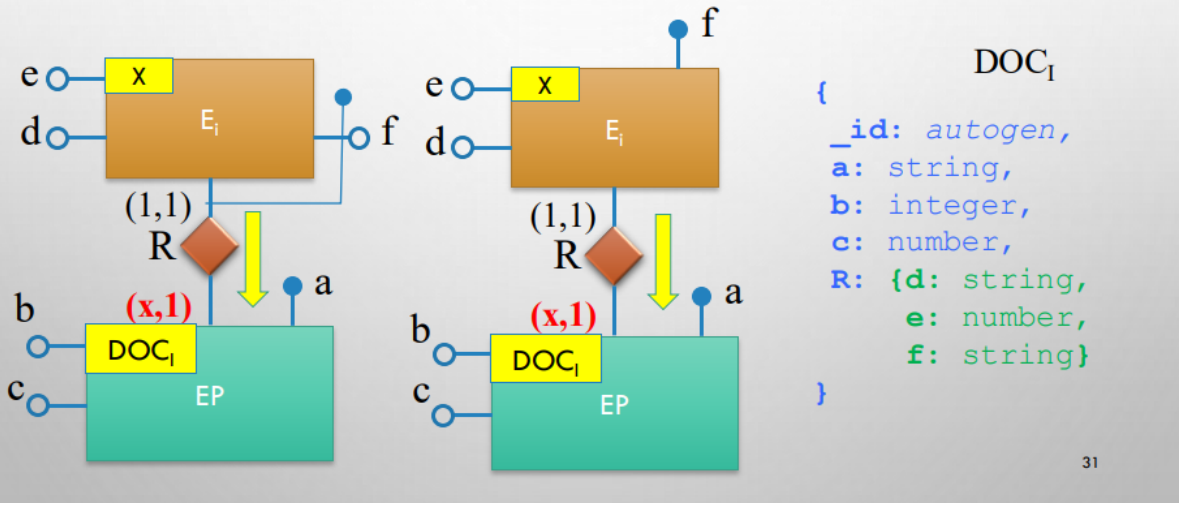
`<TIPO>` può essere: string, integer, number, boolean, date, time, timestamp.

L'attributo `_id` è sempre autogenerato dal sistema.

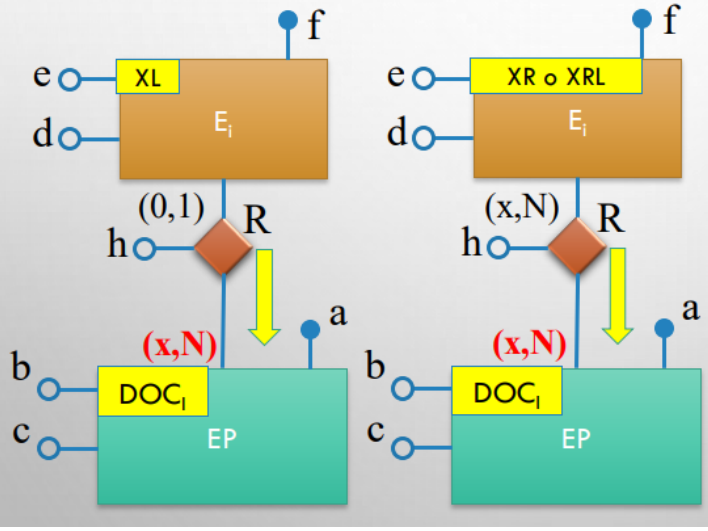
- REGOLA 2.1: ENTITÀ PRINCIPALE **EP** CON ENTITÀ DEBOLE **$E_i$** (O NON PRINCIPALE) DA INCAPSULARE. **$E_i$** PARTECIPA ALLA RELAZIONE **R** CON VINCOLO DI CARDINALITÀ (1,1) E EP CON VINCOLO DI CARDINALITÀ (X,N).




- REGOLA 2.2: ENTITÀ PRINCIPALE **EP** CON ENTITÀ DEBOLE **$E_i$** (O NON PRINCIPALE) DA INCAPSULARE. **$E_i$** PARTECIPA ALLA RELAZIONE **R** CON VINCOLO DI CARDINALITÀ (1,1) E **EP** CON VINCOLO DI CARDINALITÀ (X,1).



- REGOLA 3.1: ENTITÀ PRINCIPALE **EP** CON ENTITÀ DEBOLE **$E_i$** (O NON PRINCIPALE) DA INCAPSULARE. **$E_i$** PARTECIPA ALLA RELAZIONE **R** CON VINCOLO DI CARDINALITÀ (0,1) O (0,N) O (1,N) ED **EP** CON VINCOLO DI CARDINALITÀ (X,N).

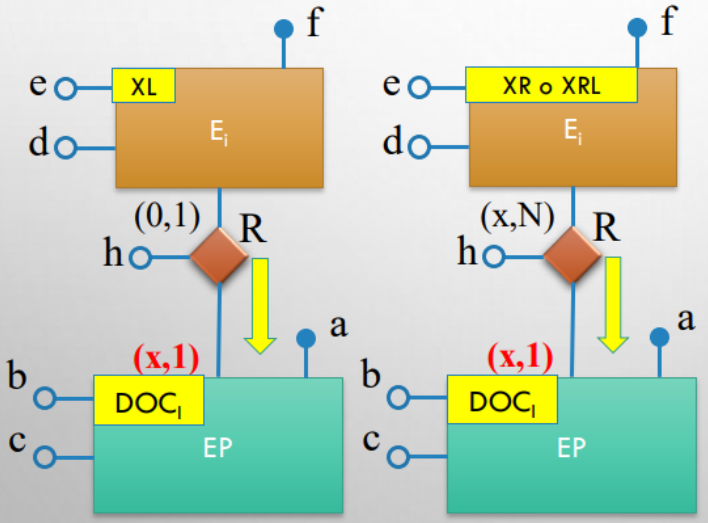



$DOC_I$

```
{
 _id: autogen,
 a: string,
 b: integer,
 c: number,
 R: [{Ei:{d: string,
          e: number,
          f: string}
     h: integer},…]
}
```

32

- REGOLA 3.2: ENTITÀ PRINCIPALE **EP** CON ENTITÀ DEBOLE **$E_i$** (O NON PRINCIPALE) DA INCAPSULARE. **$E_i$** PARTECIPA ALLA RELAZIONE **R** CON VINCOLO DI CARDINALITÀ (0,1) O (0,N) O (1,N) E **EP** CON VINCOLO DI CARDINALITÀ (X,1).

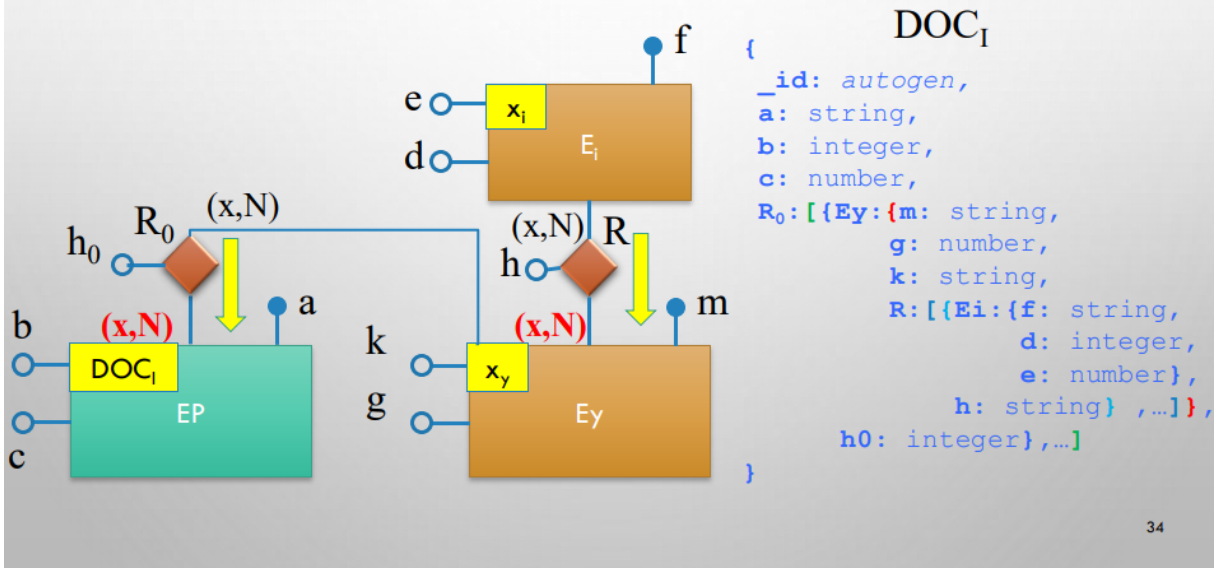



$DOC_I$

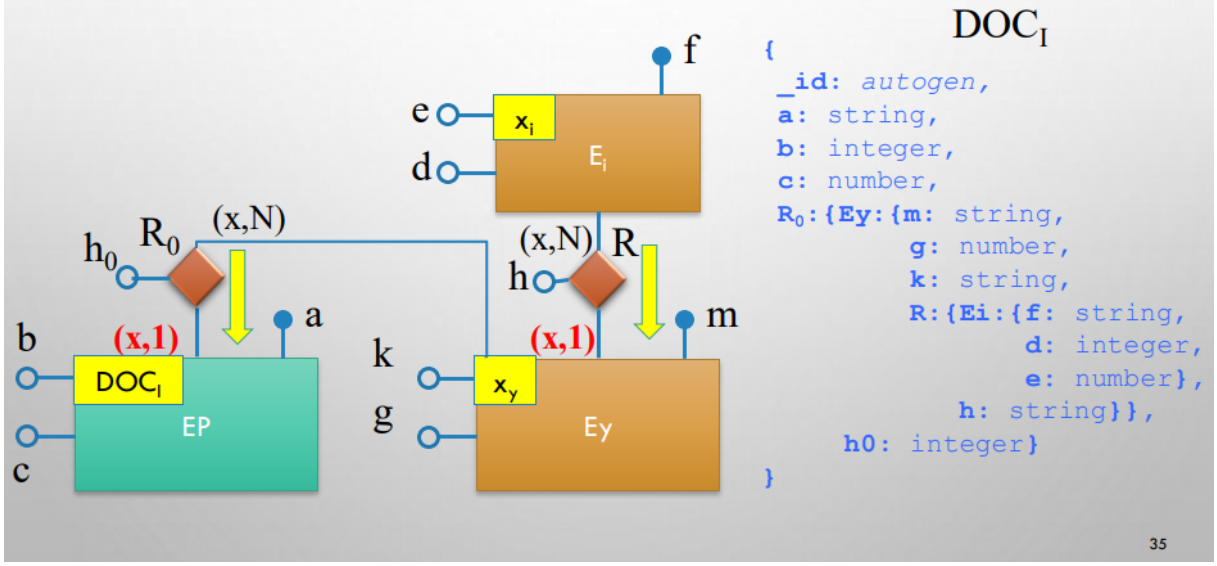
```
{
 _id: autogen,
 a: string,
 b: integer,
 c: number,
 R: {Ei:{d: string,
         e: number,
         f: string},
     h: integer}
}
```

33

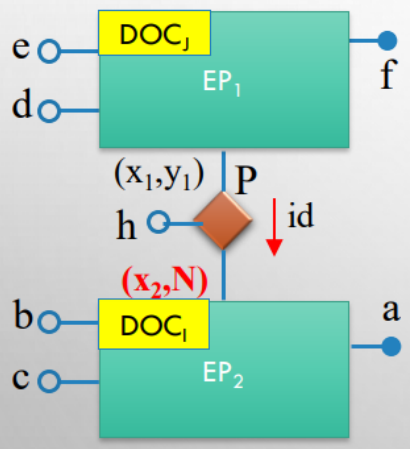
- REGOLA 4.1: **$E_i$** È COLLEGATA ATTRAVERSO UNA RELAZIONE **R** AD UNA ENTITÀ **$E_y$** (UNICA O A SCELTA) A SUA VOLTA COLLEGATA CON UN'ENTITÀ **EP**. **$E_i$** VIENE INCAPSULATA IN **$E_y$** E **$E_y$** VIENE INCAPSULATA IN EP. VINCOLI SU **$E_y$** E **EP** PARI A (0,N) O (1,N).




- REGOLA 4.2: **$E_i$** È COLLEGATA ATTRAVERSO UNA RELAZIONE **R** AD UNA ENTITÀ **$E_y$** (UNICA O A SCELTA) A SUA VOLTA COLLEGATA CON UN'ENTITÀ **EP**. **$E_i$** VIENE INCAPSULATA IN **$E_y$** E **$E_y$** VIENE INCAPSULATA IN **EP**. VINCOLI SU **$E_y$** E EP PARI A (0,1) O (1,1).



- REGOLA 5.1: RELAZIONE MOLTI A MOLTI TRA ENTITÀ PRINCIPALI **$EP_1$** E **$EP_2$**. **$EP_2$** PARTECIPA ALLA RELAZIONE **R** CON VINCOLO DI CARDINALITÀ (0,N) O (1,N) E INCAPSULO **R** SOLO IN **$EP_2$**.




$DOC_J$

```
{
 _id: autogen,
 f: string,
 e: integer,
 d: number
}
```

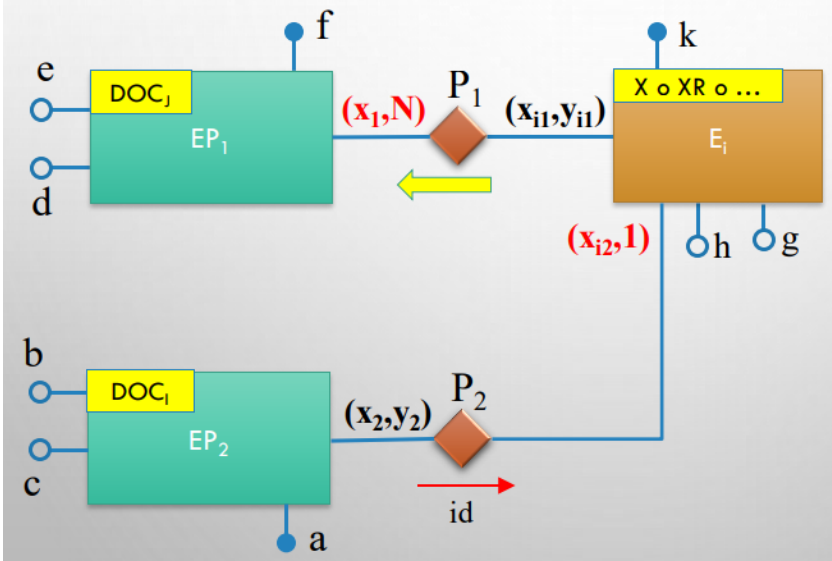
$DOC_I$

```
{
 _id: autogen,
 a: string,
 b: integer,
 c: number,
 P:[{Ep1_id:integer,
     h: number},…]
}
```

E' possibile anche il viceversa, vale a dire incapsulare P in $EP_1$, oppure incapsulare P in entrambi i documenti introducendo ridondanza.

36

- REGOLA 6.1: RELAZIONE BINARIA **P2** TRA UN'ENTITÀ NON PRINCIPALE **$E_i$**, DA INCAPSULARE IN **$EP_1$**, E UN'ENTITÀ PRINCIPALE **$EP_2$** CON VINCOLO (X,1) LATO **$E_i$**.




$DOC_J$

```
{
 _id: autogen,
 d: string,
 e: integer,
 f: number,
 P1:[{k: string,
      g: number,
      h: string,
      EP2_id: integer}
     ,…]
}
```

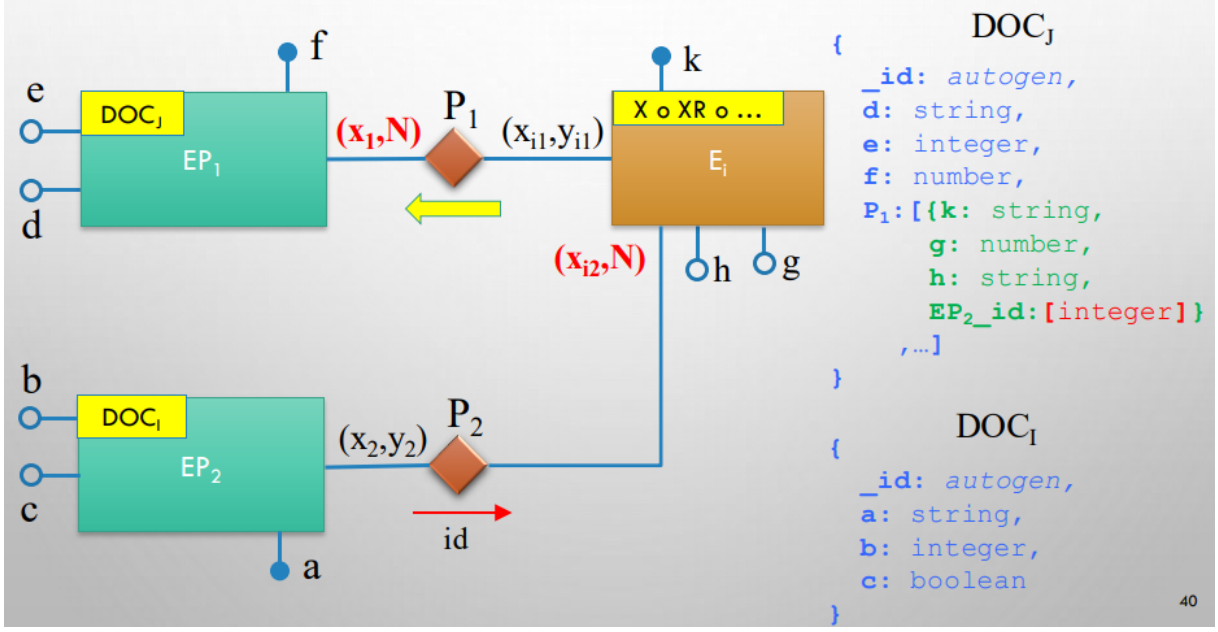
* se $x_{i2}=0$

$DOC_I$

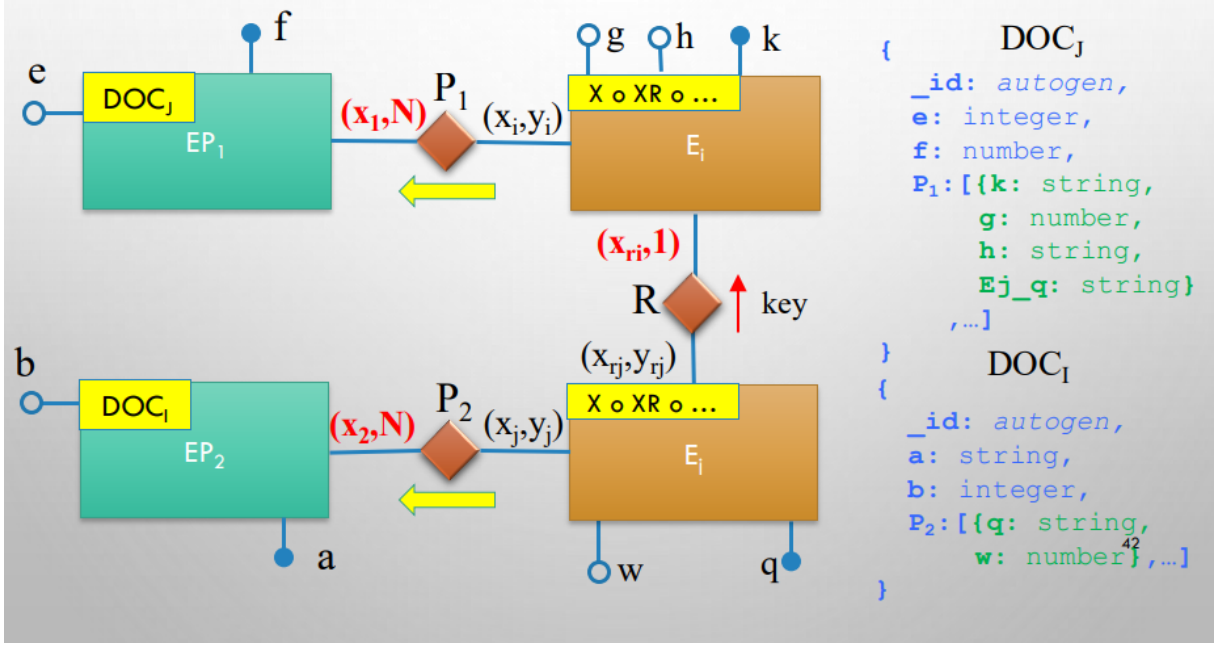
```
{
 _id: autogen,
 a: string,
 b: integer,
 c: boolean
}
```

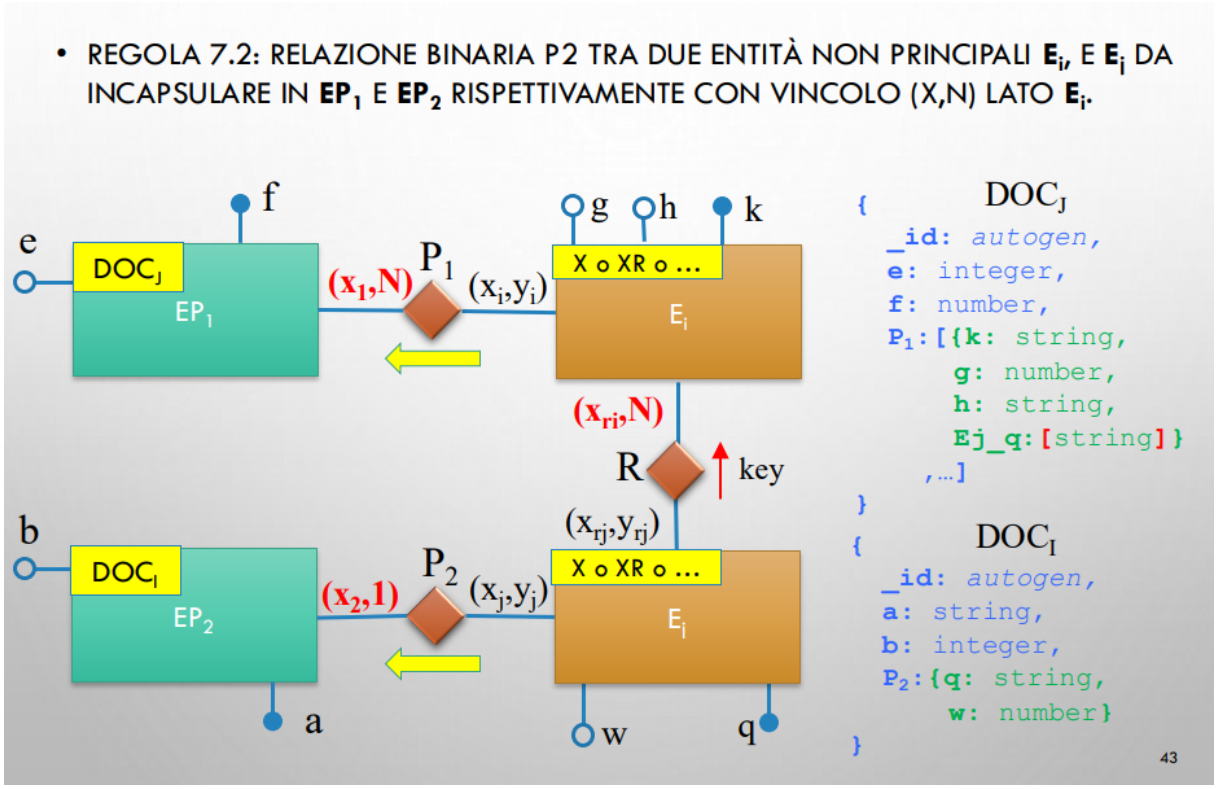
39

- REGOLA 6.2: RELAZIONE BINARIA **P2** TRA UN'ENTITÀ NON PRINCIPALE $\mathbf{E_i}$, DA INCAPSULARE IN $\mathbf{EP_1}$, E UN'ENTITÀ PRINCIPALE $\mathbf{EP_2}$ CON VINCOLO (X,N) LATO $\mathbf{E_i}$.




- REGOLA 7.1: RELAZIONE BINARIA **P2** TRA DUE ENTITÀ NON PRINCIPALI $\mathbf{E_i}$, E $\mathbf{E_j}$, DA INCAPSULARE IN $\mathbf{EP_1}$ E $\mathbf{EP_2}$ RISPETTIVAMENTE, CON VINCOLO (X,1) LATO $\mathbf{E_i}$.




• REGOLA 7.2: RELAZIONE BINARIA P2 TRA DUE ENTITÀ NON PRINCIPALI Ei, E Ej DA INCAPSULARE IN EP1 E EP2 RISPETTIVAMENTE CON VINCOLO (X,N) LATO Ei.
e
f
DOCJ
EP1
(x1,N)
P1
(xi,yi)
g
h
k
X o XR o ...
Ei
(xri,N)
R
key
(xrj,yrj)
b
DOCI
EP2
(x2,1)
P2
(xj,yj)
X o XR o ...
Ei
a
w
q
DOCJ
{
_id: autogen,
e: integer,
f: number,
P1:[{k: string,
g: number,
h: string,
Ej_q:[string]}
,...]
}
DOCI
{
_id: autogen,
a: string,
b: integer,
P2:{q: string,
w: number}
}
43

# 7 Algebra relazionale

L'**algebra relazionale** è un linguaggio formale procedurale che definisce un insieme di **operatori** applicabili a relazioni (tabelle) e che producono come risultato **nuove relazioni**. Essa costituisce il fondamento teorico dei linguaggi di interrogazione dei database relazionali, come SQL.

Ogni operatore dell'algebra relazionale è **chiuso**, ovvero restituisce sempre una relazione come risultato, permettendo la composizione di più operatori in un'unica espressione.

La **segnatura** di un operatore descrive:

- il numero di relazioni di input;
- l'eventuale presenza di parametri;
- il tipo del risultato prodotto.

Un **operatore unario** agisce su una singola relazione di input e produce una relazione di output:

$$Op_{\text{parametri}}(r_1) \;\rightarrow\; r_2$$

dove:

- $r_1$ è la relazione di input;
- $Op$ è l'operatore applicato;
- i *parametri* specificano il comportamento dell'operatore;
- $r_2$ è la relazione risultante.

Un **operatore binario** agisce su due relazioni di input e produce una nuova relazione:

$$r_1 \; Op_{\text{parametri}} \; r_2 \;\rightarrow\; r_3$$

dove:

- $r_1$ e $r_2$ sono le relazioni di input;
- $Op$ è l'operatore binario;
- i *parametri* definiscono le condizioni dell'operazione;
- $r_3$ è la relazione risultante.

## 7.1 Classificazione degli operatori

Gli operatori dell'algebra relazionale si classificano secondo due criteri principali:

### Classificazione funzionale

- Operatori insiemistici
- Operatori specifici
- Operatori di giunzione (join) tra relazioni

### Classificazione in base alla derivabilità

- Operatori di base

- Operatori derivati

## 7.2 Operatori insiemistici

Poiché le relazioni sono insiemi di tuple, è naturale applicare ad esse le operazioni dell'algebra degli insiemi. È tuttavia necessario ricordare che una relazione è un insieme di tuple **omogenee**, cioè definite sullo stesso insieme di attributi (stesso schema).

Di conseguenza, gli operatori insiemistici possono essere applicati esclusivamente a relazioni che hanno **lo stesso schema**.

Dati due relazioni $r_1$ e $r_2$ definite sullo stesso schema $X$, si definiscono i seguenti operatori:

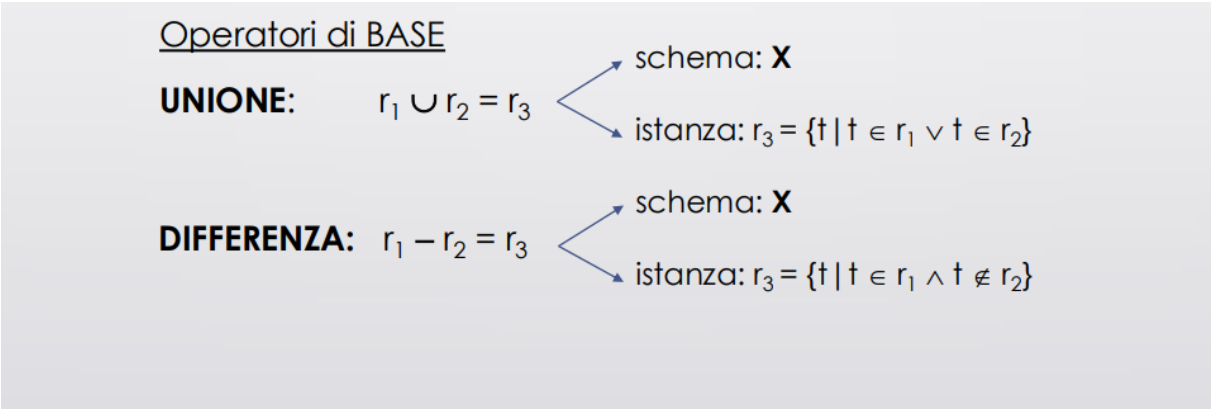



Figura 56: operazione di unione e differenza

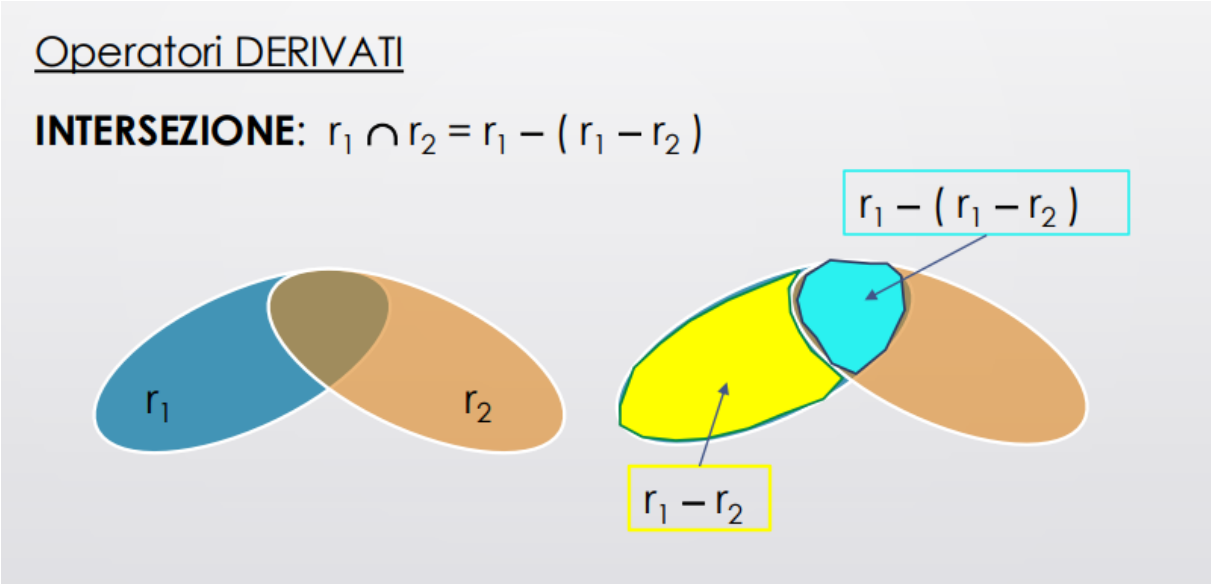



Figura 57: Intersezione

### 7.2.1 Cardinalità degli operatori insiemistici

La **cardinalità** di una relazione indica il numero di tuple contenute nella relazione risultante dall'applicazione di un operatore dell'algebra relazionale.

### Unione

L'operatore di unione combina le tuple di due relazioni eliminando le eventuali tuple duplicate. La cardinalità della relazione risultante dipende dal numero di tuple comuni tra le due relazioni.

Il valore minimo si ottiene quando una relazione è interamente contenuta nell'altra; in questo caso, la cardinalità dell'unione coincide con quella della relazione più grande. Il valore massimo si ottiene quando le due relazioni non hanno tuple in comune; in questo caso, la cardinalità dell'unione è pari alla somma delle cardinalità delle due relazioni.

$$\max(|r_1|, |r_2|) \;\leq\; |r_1 \cup r_2| \;\leq\; |r_1| + |r_2|$$

### Differenza

L'operatore di differenza restituisce le tuple appartenenti alla prima relazione che non sono presenti nella seconda relazione. La cardinalità della relazione risultante può variare da zero fino alla cardinalità della prima relazione.

Il valore minimo si ottiene quando tutte le tuple della prima relazione sono presenti anche nella seconda, producendo una relazione vuota. Il valore massimo si ottiene quando nessuna tupla della prima relazione è presente nella seconda, per cui tutte le tuple della prima relazione vengono mantenute.

$$0 \;\leq\; |r_1 - r_2| \;\leq\; |r_1|$$

### Intersezione

L'operatore di intersezione restituisce le tuple comuni a entrambe le relazioni. La cardinalità della relazione risultante dipende quindi dal numero di tuple condivise.

Il valore minimo si ottiene quando le due relazioni non hanno tuple in comune, producendo una relazione vuota. Il valore massimo si ottiene quando la relazione più piccola è interamente contenuta nella relazione più grande.

$$0 \;\leq\; |r_1 \cap r_2| \;\leq\; \min(|r_1|, |r_2|)$$

## 7.3 Operatori specifici

### 7.3.1 Operatore di rinominaizone

Dato una relazione $r$ definita sullo schema $X$, con

$$X = \{A_1, \ldots, A_n\},$$

si definiscono i seguenti operatori. L'operatore di **rinominazione** consente di modificare il nome degli attributi di una relazione senza alterarne le tuple. Sia

$$Y = \{B_1, \ldots, B_n\}$$

un insieme di attributi tale che $|Y| = |X|$.

La rinominazione è definita come:

$$\rho_{A_1 \to B_1, \ldots, A_n \to B_n}(r) = r_2$$

dove:

- lo schema della relazione risultante è $Y$;
- l'istanza $r_2$ contiene le stesse tuple di $r$;
- per ogni tupla $t \in r$ e per ogni $i \in \{1, \ldots, n\}$, vale $t[B_i] = t[A_i]$.

### Esempio

Schema relazionale:

- `CAPOLUOGO_PROV(Nome, Popolazione, CAP)`
- `COMUNI_MINORI(Nome, Abitanti, CAP)`

Per produrre la lista di tutti i comuni è necessario rendere compatibili gli schemi delle due relazioni. Si applica quindi l'operatore di rinominazione per uniformare il nome degli attributi:

$$\rho_{\text{Popolazione}\to\text{Abitanti}}(\text{CAPOLUOGO_PROV}) \cup \text{COMUNI_MINORI}$$

Dato una relazione $r$ definita sullo schema $X$, con

$$X = \{A_1, \ldots, A_n\},$$

si definiscono i seguenti operatori.

### 7.3.2 Operatore di Selezione

L'operatore di **selezione** consente di estrarre da una relazione le tuple che soddisfano una determinata condizione.

La selezione è definita come:

$$\sigma_F(r_1) = r_2$$

dove:

- lo schema della relazione risultante è lo stesso di $r_1$;
- l'istanza risultante è
$$r_2 = \{\, t \mid t \in r_1 \wedge F(t) \,\};$$
- $F(t)$ indica che la tupla $t$ rende vera la formula $F$.

La formula $F(t)$ è una formula proposizionale ottenuta combinando, mediante i connettivi logici $\wedge$, $\vee$ e $\neg$, condizioni atomiche del tipo:

$$A \;\theta\; B \quad \text{oppure} \quad A \;\theta\; c$$

dove:

- $A$ e $B$ sono attributi della relazione;
- $\theta \in \{=, \neq, >, \geq, <, \leq\}$;
- $c$ è una costante appartenente al dominio di $A$ o compatibile con esso.

Una formula del tipo $A\ \theta\ B$ è vera sulla tupla $t$ se $t[A]\ \theta\ t[B]$, mentre una formula del tipo $A\ \theta\ c$ è vera se $t[A]\ \theta\ c$.

Le formule composte assumono valori booleani determinati dalle tavole di verità dei connettivi logici utilizzati.

### Esempio

Schema relazionale:

- `TRENO(Numero, PartOra, PartMin, Cat, Dest, ArrOra, ArrMin)`
- `FERMATA(NumTreno, Stazione, Ora, Min)`

Si richiede di produrre la lista di tutti i treni non regionali che partono alle ore 12:00 o dopo le 12:00 e prima delle 16:00.

L'espressione in algebra relazionale è:

$$\sigma_{\text{PartOra}\geq 12\ \wedge\ \text{PartOra}<16\ \wedge\ \text{Cat}\neq\text{'REGIONALE'}}(\text{TRENO})$$

### 7.3.3 Proiezione

L'operatore di proiezione consente di ridurre una relazione eliminando alcuni attributi, mantenendo solo quelli di interesse. La riduzione avviene in senso verticale.

Sia $Y = \{A_1, \ldots, A_m\} \subseteq X$ un sottoinsieme degli attributi dello schema $X$ della relazione $r_1$.

La proiezione è definita come:

$$\pi_Y(r_1) = r_2$$

dove:

- lo schema della relazione risultante è $Y$;
- l'istanza risultante è
$$r_2 = \{\, t \mid \exists t' \in r_1 : t'[Y] = t \,\};$$
- $t'[Y]$ indica la tupla ottenuta da $t'$ considerando solo gli attributi appartenenti a $Y$;
- eventuali tuple duplicate vengono eliminate.

### Esempio

Schema relazionale:

- `TRENO(Numero, PartOra, PartMin, Cat, Dest, ArrOra, ArrMin)`
- `FERMATA(NumTreno, Stazione, Ora, Min)`

Si richiede di estrarre il **numero** e l'**ora di partenza** dei treni *Freccia Rossa* (categoria = `'FR'`) con destinazione *Milano Centrale*.

L'espressione in algebra relazionale è:

$$\pi_{\text{Numero,PartOra}}\big(\sigma_{\text{Dest='Milano Centrale'}\ \wedge\ \text{Cat='FR'}}(\text{TRENO})\big)$$

### 7.3.4 Cardinalità degli operatori specifici

La cardinalità indica il numero di tuple contenute nella relazione risultato di un'operazione di algebra relazionale.

**Selezione:** L'operatore di selezione $\sigma$ filtra le tuple di una relazione in base a una condizione logica, senza modificare gli attributi.

La cardinalità della relazione risultante soddisfa la seguente relazione:

$$0 \leq |\sigma_F(r_1)| \leq |r_1|$$

Questo significa che:

- la selezione può restituire nessuna tupla;
- può restituire tutte le tuple della relazione originale;
- non può mai aumentare il numero di tuple.

**Proiezione** L'operatore di proiezione $\Pi$ seleziona un sottoinsieme di attributi della relazione ed elimina eventuali tuple duplicate.

La cardinalità della relazione risultante è compresa tra:

$$\min(|r_1|, 1) \leq |\Pi_Y(r_1)| \leq |r_1|$$

In particolare:

- la proiezione può ridurre il numero di tuple a causa dell'eliminazione dei duplicati;
- nel caso migliore, la cardinalità rimane invariata.

Se l'insieme di attributi $Y$ è una superchiave per la relazione $r_1$, allora non si generano duplicati e vale:

$$|\Pi_Y(r_1)| = |r_1|$$

### 7.3.5 Operatori di giunzione (JOIN)

Dati due relazioni $r_1$ e $r_2$ con schema rispettivamente $X_1$ e $X_2$, gli operatori di *join* generano tuple nella relazione risultato a partire da coppie di tuple $(t_1, t_2) \in r_1 \times r_2$ che soddisfano una determinata condizione, detta *predicato di join*.

### 7.3.6 JOIN naturale

Il join naturale è un operatore di base in cui il predicato di join è definito automaticamente sugli attributi comuni alle due relazioni. Tale operatore è quindi dipendente dallo schema.

Lo schema della relazione risultante è:

$$\text{schema}(r_3) = X_1 \cup X_2$$

dove gli attributi comuni compaiono una sola volta.

L'istanza della relazione risultato è definita come:

$$r_3 = \{\, t \mid \exists t_1 \in r_1,\ \exists t_2 \in r_2 : t[X_1] = t_1 \wedge t[X_2] = t_2 \,\}$$

Il predicato di join è una condizione di uguaglianza sugli attributi comuni alle due relazioni.

### Esempio

Si consideri il seguente schema relazionale:

$$\text{DOCENTE}(\underline{CFDocente}, Nome, Cognome)$$
$$\text{CORSO}(Nome, \underline{CFDocente})$$

Si richiede di produrre l'insieme dei corsi riportando il nome del corso e il cognome del docente che lo tiene.

Poiché le informazioni richieste sono distribuite su due relazioni diverse, è necessario applicare un operatore di join. Il collegamento tra le relazioni avviene tramite l'attributo comune `CFDocente`.

Dal momento che l'attributo `Nome` della relazione `CORSO` deve essere rinominato in `NomeCorso`, si applica preliminarmente l'operatore di rinomina.

L'espressione di algebra relazionale è la seguente:

$$\Pi_{NomeCorso,\, Cognome}\Big(\rho_{Nome \to NomeCorso}(\text{CORSO}) \bowtie \text{DOCENTE}\Big)$$

L'operatore di join naturale combina le tuple delle due relazioni sulla base dell'uguaglianza dell'attributo comune `CFDocente`. Successivamente, la proiezione consente di ottenere esclusivamente gli attributi richiesti nel risultato finale.

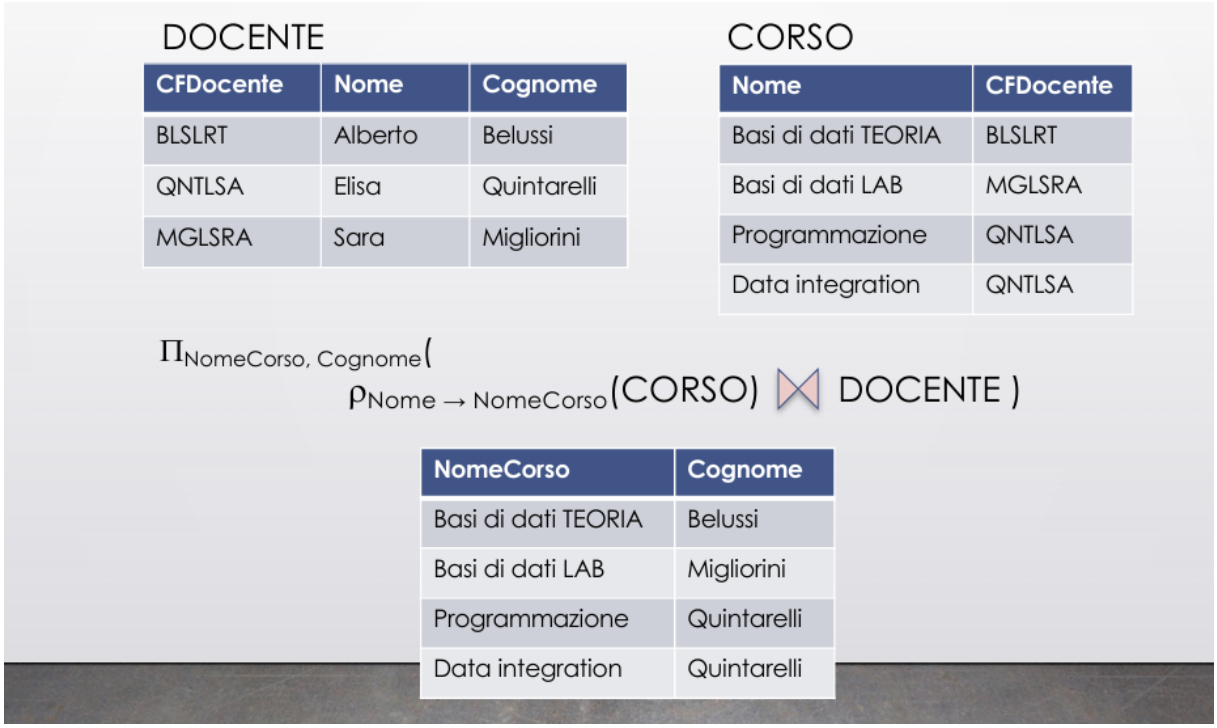



Figura 58: Esempio più visivo del JOIN naturale

### Proprietà del JOIN naturale

Il join naturale tra due relazioni $r_1$ e $r_2$ si dice *completo* se tutte le tuple delle due relazioni contribuiscono a generare almeno una tupla nella relazione risultato.

Formalmente, il join naturale è completo se valgono entrambe le seguenti condizioni:

$$\forall t_1 \in r_1 : \ \exists t \in r_1 \bowtie r_2 \text{ tale che } t[X_1] = t_1$$

$$\forall t_2 \in r_2 : \ \exists t \in r_1 \bowtie r_2 \text{ tale che } t[X_2] = t_2$$

Queste condizioni garantiscono che ogni tupla di $r_1$ e di $r_2$ compaia almeno una volta nel risultato del join.

Se il join naturale non è completo, le tuple che non contribuiscono alla generazione di alcuna tupla nel risultato si dicono **dangling tuples**. Il join naturale gode di alcune proprietà algebriche analoghe a quelle degli operatori classici dell'algebra.

**Commutatività:** Il join naturale è commutativo:

$$r_1 \bowtie r_2 = r_2 \bowtie r_1$$

**Associatività:** Il join naturale è associativo:

$$r_1 \bowtie (r_2 \bowtie r_3) = (r_1 \bowtie r_2) \bowtie r_3$$

**Relazioni con lo stesso schema:** Se le relazioni $r_1$ e $r_2$ hanno lo stesso schema, allora il join naturale coincide con l'intersezione:

$$r_1 \bowtie r_2 = r_1 \cap r_2$$

**Assenza di attributi comuni:** Se le relazioni $r_1$ e $r_2$ non hanno attributi in comune, allora il join naturale coincide con il prodotto cartesiano:

$$r_1 \bowtie r_2 = r_1 \times r_2$$

### 7.3.7 Cardinalità del JOIN naturale

In generale, la cardinalità del join naturale tra due relazioni $r_1$ e $r_2$ è compresa tra zero e il prodotto delle cardinalità delle due relazioni:

$$0 \leq |r_1 \bowtie r_2| \leq |r_1| \cdot |r_2|$$

Se il join naturale è completo, allora ogni tupla di entrambe le relazioni contribuisce a generare almeno una tupla nel risultato, e vale:

$$\max(|r_1|, |r_2|) \leq |r_1 \bowtie r_2| \leq |r_1| \cdot |r_2|$$

Se l'insieme degli attributi comuni $X_1 \cap X_2$ è una superchiave per la relazione $r_2$, allora a ogni tupla di $r_1$ può corrispondere al più una tupla di $r_2$, e quindi:

$$0 \leq |r_1 \bowtie r_2| \leq |r_1|$$

Se $X_1 \cap X_2$ è una superchiave per $r_2$ ed esiste un vincolo di integrità referenziale tra $X_1 \cap X_2$ (o una sua parte) di $r_1$ e $r_2$, allora ogni tupla di $r_1$ trova corrispondenza in $r_2$ e vale:

$$|r_1 \bowtie r_2| = |r_1|$$

### 7.3.8 THETA JOIN

Date due relazioni $r_1$ e $r_2$ di schema $X_1$ e $X_2$ rispettivamente, si introduce, oltre al *join naturale*, il seguente operatore. **Theta Join**: il predicato di join viene indicato esplicitamente ed è indipendente dallo schema delle relazioni.

#### Precondizione

Gli schemi delle relazioni devono essere disgiunti:

$$X_1 \cap X_2 = \emptyset$$

#### Definizione

Il theta join è definito come una selezione applicata al prodotto cartesiano:

$$r_1 \bowtie_\theta r_2 = \sigma_\theta(r_1 \times r_2)$$

dove:

- $\times$ è il prodotto cartesiano
- $\sigma_\theta$ è l'operatore di selezione
- $\theta$ è un predicato conforme alla sintassi della selezione

Il predicato $\theta$ può contenere confronti tra attributi delle due relazioni utilizzando operatori di confronto come $=, \neq, <, >, \leq, \geq$.

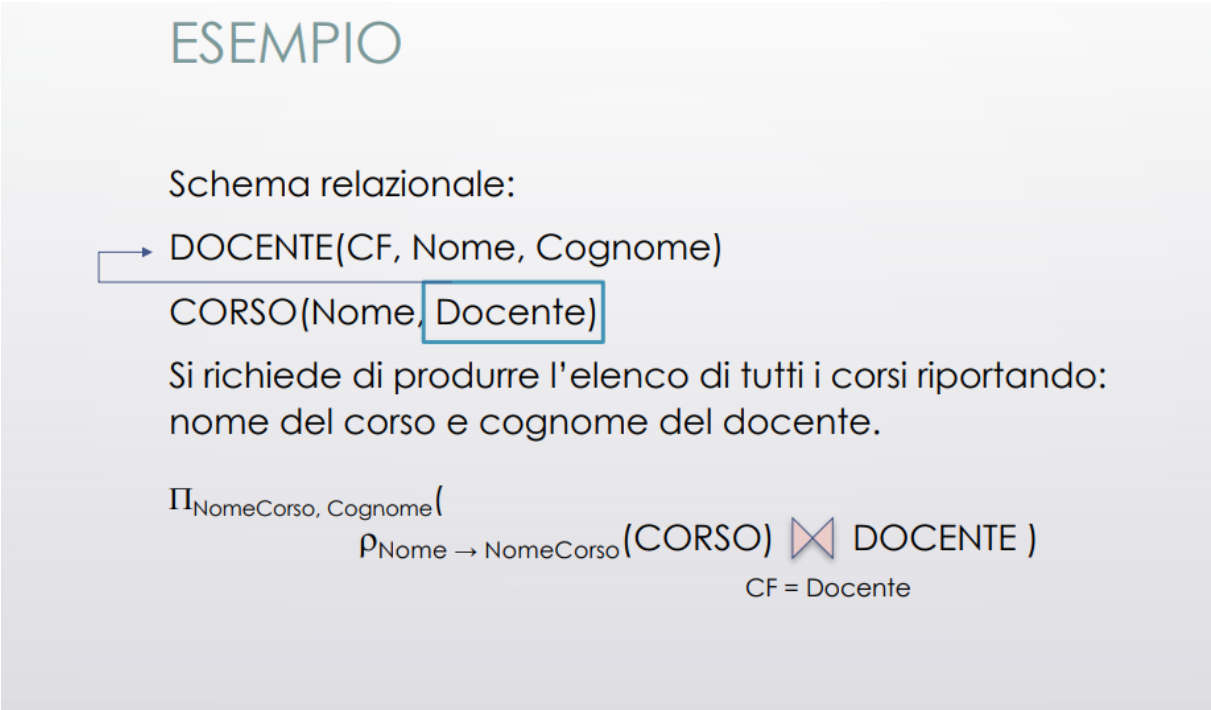



Figura 59: Esempio dell'uso del Theta Join

#### Proprietà del Theta Join

Il **theta join** si dice **equi-join** se la condizione $\theta$ è una congiunzione di uguaglianze:

$$r_1 \bowtie_{A_1=B_1 \wedge ... \wedge A_n=B_n} r_2$$

**N.B.: ATTENZIONE** non esiste un operatore misto che applica la logica del *join naturale* per gli attributi comuni e aggiunge ulteriori condizioni specificate esplicitamente tramite il predicato $\theta$.

**Teorema**

Date due relazioni $r_1$ e $r_2$ di schema $X_1$ e $X_2$ con

$$X_1 \cap X_2 = \{C_1, \ldots, C_m\} \quad \text{con } m > 0,$$

vale la seguente equivalenza:

$$r_1 \bowtie r_2 = \Pi_{X_1 \cup X_2} \left( r_1 \bowtie_{C_1=C'_1 \wedge ... \wedge C_m=C'_m} \rho_{C_1,...,C_m \to C'_1,...,C'_m}(r_2) \right)$$

**Ridenominazione** Data la relazione $r_1$ di schema

$$X_1 = \{A_1, \ldots, A_n\},$$

per cambiare nome a tutti gli attributi si può scrivere:

$$\rho_{A_1,...,A_n \to A'_1,...,A'_n}(r_1)$$

oppure, in forma breve:

$$\rho_{X \to X'}(r_1)$$

**N.B.: non sono ammesse** notazioni simili alla *dot notation* per indicare l'attributo di una tabella.

La notazione $T.A$ (per indicare l'attributo $A$ della tabella $T$) **NON è una notazione ammessa dall'algebra relazionale**.

## 7.4 Algebra con valori nulli

È opportuno estendere l'algebra relazionale affinché possa operare anche su relazioni che contengono valori nulli. In particolare, è necessario definire in modo esplicito il comportamento delle operazioni che coinvolgono il confronto tra attributi, poiché la presenza di valori nulli introduce ambiguità semantiche.

Le operazioni che richiedono una gestione specifica dei valori nulli sono:

- **Selezione**
- **Join naturale**

Le altre operazioni dell'algebra relazionale (proiezione, unione, differenza e prodotto cartesiano) si limitano a propagare i valori nulli eventualmente presenti nelle tuple di input, senza introdurre nuove problematiche semantiche.

**Motivazione:** La selezione e il join naturale si basano su confronti tra valori di attributi. In presenza di valori nulli, tali confronti non possono essere valutati secondo la logica booleana classica, ma richiedono una logica a tre valori (*vero*, *falso*, *sconosciuto*). Per questo motivo, è necessario estendere formalmente l'algebra relazionale per garantire un comportamento coerente e ben definito di queste operazioni.

### 7.4.1 Selezione

Le condizioni di selezione in presenza di valori nulli assumono i seguenti valori di verità:

- Sia $A\,\theta\,B$ una condizione atomica applicata a una tupla $t$. Se $t[A] = \text{NULL}$ oppure $t[B] = \text{NULL}$, allora l'espressione $t[A]\,\theta\,t[B]$ è valutata come **FALSO**.
- Sia $A\,\theta\,c$ una condizione atomica applicata a una tupla $t$, dove $c$ è una costante. Se $t[A] = \text{NULL}$, allora l'espressione $t[A]\,\theta\,c$ è valutata come **FALSO**.
- Sono introdotte le seguenti condizioni atomiche aggiuntive:

  $$A \text{ IS NULL} \qquad \text{e} \qquad A \text{ IS NOT NULL}.$$

  L'espressione $A$ IS NULL, valutata sulla tupla $t$, è **VERO** se $t[A] = \text{NULL}$, altrimenti è **FALSO**. Viceversa, l'espressione $A$ IS NOT NULL è **VERO** se $t[A] \neq \text{NULL}$, altrimenti è **FALSO**.

### 7.4.2 Join naturale

Nel join naturale, la condizione di uguaglianza sugli attributi comuni alle due relazioni è valutata come **FALSA** sulle tuple $t_1$ e $t_2$ se almeno uno degli attributi comuni di $t_1$ oppure di $t_2$ assume valore `NULL`.

Si osservi, in particolare, che il confronto

$$\texttt{NULL} = \texttt{NULL}$$

è sempre valutato come **FALSO**.

## 7.5 Ottimizzazione di espressioni DML

Ogni espressione DML (generalmente specificata tramite un linguaggio dichiarativo) ricevuta dal DBMS è sottoposta a un processo di elaborazione e ottimizzazione prima della sua esecuzione.

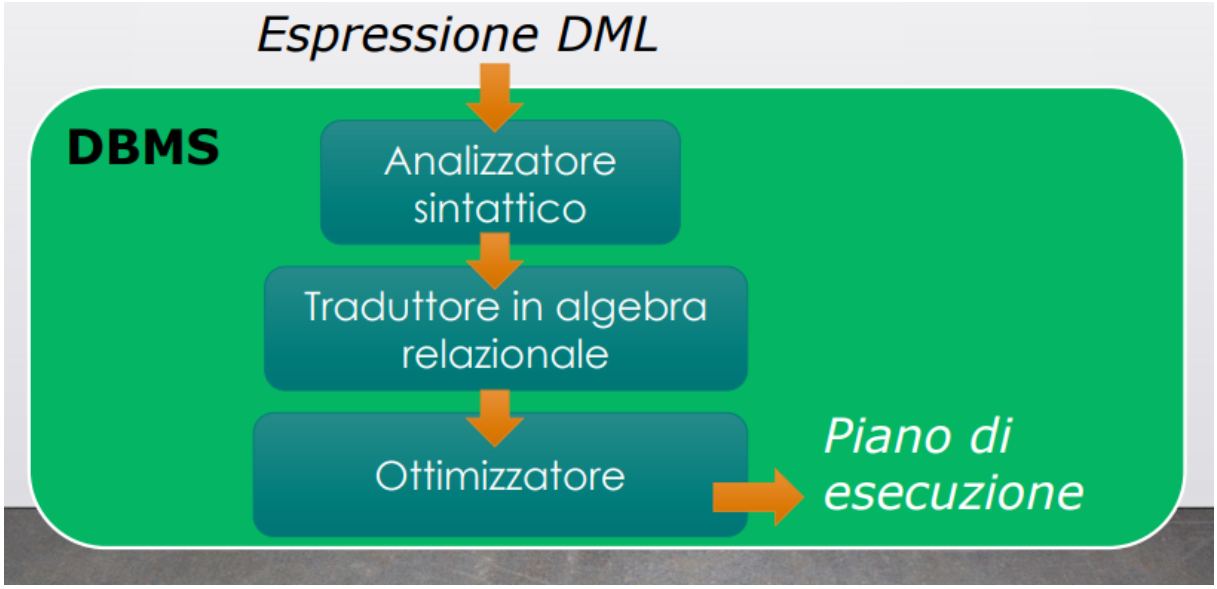



Figura 60: Ottimizzazione di espressioni DML

L'ottimizzatore genera un'espressione equivalente a quella di input, ma caratterizzata da un costo inferiore. Il costo viene stimato principalmente in termini di dimensione dei risultati intermedi, numero di accessi ai dati e operazioni eseguite.

Per ridurre il costo complessivo dell'esecuzione, l'ottimizzatore applica trasformazioni di equivalenza sull'espressione di input, con l'obiettivo di minimizzare la dimensione dei risultati intermedi e migliorare l'efficienza dell'elaborazione.

## 7.6 Equivalenza tra espressioni algebriche

### 7.6.1 Equivalenza dipendente dallo schema

Dato uno schema $R$, due espressioni algebriche $E_1$ ed $E_2$ si dicono *equivalenti rispetto allo schema* $R$, e si scrive:

$$E_1 \equiv_R E_2$$

se vale che

$$E_1(r) = E_2(r) \quad \text{per ogni istanza } r \text{ dello schema } R.$$

**Esempio:**

$$\Pi_{AB}(R_1) \bowtie \Pi_{AC}(R_2) \equiv_R \Pi_{ABC}(R_1 \bowtie R_2)$$

### 7.6.2 Equivalenza assoluta

L'equivalenza assoluta è indipendente dallo schema. Due espressioni $E_1$ ed $E_2$ si dicono *assolutamente equivalenti*, e si scrive:

$$E_1 \equiv E_2$$

se vale che

$$E_1 \equiv_R E_2 \quad \text{per ogni schema } R \text{ compatibile con } E_1 \text{ ed } E_2.$$

**Esempio:**

$$\Pi_{AB}(\sigma_{A>0}(R)) \equiv \sigma_{A>0}(\Pi_{AB}(R))$$

## 7.7 Trasformazioni di equivalenza

Sia $E$ un'espressione di schema $X$. Si definiscono le seguenti trasformazioni di equivalenza.

### Atomizzazione delle selezioni

$$\sigma_{F_1 \wedge F_2}(E) \equiv \sigma_{F_1}(\sigma_{F_2}(E))$$

Questa trasformazione è propedeutica ad altre trasformazioni. Da sola non produce un'ottimizzazione se non è seguita da ulteriori riscritture.

### 7.7.1 Idempotenza delle proiezioni

$$\Pi_Y(E) \equiv \Pi_Y(\Pi_Z(E)) \qquad \text{con } Z \subseteq X$$

Anche questa trasformazione è propedeutica e non comporta un'ottimizzazione se applicata isolatamente.

## 7.8 Trasformazioni di equivalenza con il join

Siano $E_1$ ed $E_2$ espressioni di schema rispettivamente $X_1$ e $X_2$.

### Anticipazione delle selezioni rispetto al join

$$\sigma_F(E_1 \bowtie E_2) \equiv E_1 \bowtie \sigma_F(E_2)$$

La trasformazione è applicabile solo se la condizione $F$ fa riferimento esclusivamente ad attributi di $E_2$.

### Anticipazione della proiezione rispetto al join

$$\Pi_{X_1 \cup Y}(E_1 \bowtie E_2) \equiv_R E_1 \bowtie \Pi_Y(E_2)$$

La trasformazione è applicabile solo se:

$$Y \subseteq X_2 \quad \text{e} \quad (X_2 - Y) \cap X_1 = \emptyset$$

## 7.9 Combinazione di anticipazione e idempotenza

Combinando l'anticipazione della proiezione con l'idempotenza delle proiezioni si ottengono le seguenti equivalenze:

$$\Pi_Y(E_1 \bowtie_F E_2) \equiv \Pi_Y\big(\Pi_{Y_1}(E_1) \bowtie_F \Pi_{Y_2}(E_2)\big)$$

$$\Pi_Y(E_1 \bowtie E_2) \equiv \Pi_Y\big(\Pi_{Y_1}(E_1) \bowtie \Pi_{Y_2}(E_2)\big)$$

dove:

- $Y_1 = (X_1 \cap Y) \cup J_1$
- $Y_2 = (X_2 \cap Y) \cup J_2$

- $J_1$ e $J_2$ sono gli attributi di $E_1$ ed $E_2$ coinvolti nel join. Nel caso di theta-join, tali attributi sono quelli presenti nella condizione $F$; nel caso di join naturale vale:

$$J_1 = J_2 = X_1 \cap X_2$$

TRENO(NumTreno, OrarioPart, Cat, Dest, OrarioArr)
FERMATA(NumTreno, Stazione, Orario)

**Q**: Trovare il numero e l'ora di partenza dei treni freccia rossa che fermano a Vicenza.

$$\Pi_{\text{NumTreno, OrarioPart}}(\sigma_{\text{Cat='FR'} \wedge \text{Stazione='Vicenza'}}(\text{TRENO} \bowtie \text{FERMATA}))$$

Figura 61: Esempio di applicazione delle trasformazioni

# Esempio di applicazione delle trasformazioni

TRENO(NumTreno, OrarioPart, Cat, Dest, OrarioArr)
FERMATA(NumTreno, Stazione, Orario)

$$\Pi_{\text{NumTreno, OrarioPart}}(\sigma_{\text{Cat='FR'} \wedge \text{Stazione='Vicenza'}}(\text{TRENO} \bowtie \text{FERMATA}))$$

**Atomizzazione delle selezioni**

$$\Pi_{\text{NumTreno, OrarioPart}}(\sigma_{\text{Cat='FR'}}(\sigma_{\text{Stazione='Vicenza'}}(\text{TRENO} \bowtie \text{FERMATA})))$$

**Anticipo delle selezioni**

$$\Pi_{\text{NumTreno, OrarioPart}}(\sigma_{\text{Cat='FR'}}(\text{TRENO}) \bowtie \sigma_{\text{Stazione='Vicenza'}}(\text{FERMATA}))$$

Figura 62: Esempio di applicazione delle trasformazioni

TRENO(NumTreno, OrarioPart, Cat, Dest, OrarioArr)
FERMATA(NumTreno, Stazione, Orario)

**Anticipo delle selezioni** (riportato dal lucido precedente)

$$\Pi_{\text{NumTreno, OrarioPart}}(\sigma_{\text{Cat='FR'}}(\text{TRENO}) \bowtie \sigma_{\text{Stazione='Vicenza'}}(\text{FERMATA}))$$

**Anticipo delle proiezioni**

$$\Pi_{\text{NumTreno, OrarioPart}}(\Pi_{\text{NumTreno, OrarioPart}}(\sigma_{\text{Cat='FR'}}(\text{TRENO})) \bowtie \Pi_{\text{NumTreno}}(\sigma_{\text{Stazione='Vicenza'}}(\text{FERMATA}))$$

Figura 63: Esempio di applicazione delle trasformazioni

## Ulteriori trasformazioni di equivalenza

Siano $E_1$ ed $E_2$ espressioni di schema rispettivamente $X_1$ e $X_2$. Si definisce la seguente trasformazione di equivalenza.

### Inglobamento di una selezione in un prodotto cartesiano

Questa trasformazione consente di inglobare una selezione applicata a un prodotto cartesiano in un join con condizione:

$$\sigma_F(E_1 \times E_2) \equiv E_1 \bowtie_F E_2$$

La trasformazione è applicabile solo se:

$$X_1 \cap X_2 = \emptyset$$

Si osservi che questa regola va applicata solo dopo aver verificato che non sia possibile anticipare le selezioni rispetto al join.

- Applicazione delle proprietà commutativa e associativa di: unione, prodotto cartesiano, intersezione.
- Applicazione della proprietà distributiva:

$$\sigma_F(E1 \cup E2) \equiv \sigma_F(E1) \cup \sigma_F(E2)$$

$$\sigma_F(E1 - E2) \equiv \sigma_F(E1) - \sigma_F(E2)$$

$$\Pi_Y(E1 \cup E2) \equiv \Pi_Y(E1) \cup \Pi_Y(E2)$$

$$E1 \bowtie (E2 \cup E3) \equiv (E1 \bowtie E2) \cup (E1 \bowtie E3)$$

Figura 64: Applicazione della proprietà distributiva

Applicazione di altre trasformazioni:

$$\sigma_{F1 \vee F2}(E) \equiv \sigma_{F1}(E) \cup \sigma_{F2}(E)$$

$$\sigma_{F1 \wedge F2}(E) \equiv \sigma_{F1}(E) \cap \sigma_{F2}(E) \equiv \sigma_{F1}(E) \bowtie \sigma_{F2}(E)$$

$$\sigma_{F1 \wedge \neg F2}(E) \equiv \sigma_{F1}(E) - \sigma_{F2}(E)$$

Figura 65: applicazione di altre trasformazioni

TRENO(NumTreno, Cat, OraPart, MinPart, OraArr, MinArr, Dest)

FERMATA(NumTreno, Stazione, Ora, Min)

**Q**: Trovare la destinazione dei treni non regionali che fermano a "Peschiera".

Dimensioni dei dati:

| TRENO: | FERMATA: |
|---|---|
| #attributi = 7 | #attributi = 4 |
| #tuple = 150 | #tuple = 1000 |
| #tuple treni regionali = 40 | #tuple staz peschiera = 50 |
| | #tuple staz peschiera e treno non regionale = 45 |

Figura 66: Esempio

# 8 Calcolo Relazionale

Il calcolo relazionale è un linguaggio di interrogazione dichiarativo: esso specifica le proprietà che devono essere soddisfatte dal risultato dell'interrogazione, senza descrivere le modalità operative per ottenerlo.

Esistono due versioni del calcolo relazionale:

- il calcolo relazionale sui domini;
- il calcolo relazionale sulle tuple.

**Caratteristiche del linguaggio** Nel calcolo relazionale sulle tuple, un'interrogazione è specificata mediante una formula logica con variabili libere. La formula viene interpretata sul contenuto della base di dati e le variabili assumono come valori le tuple delle relazioni presenti nella base di dati.

Il calcolo relazionale sulle tuple costituisce il fondamento teorico del linguaggio SQL nella sua forma *semplice*, ossia senza operatori aggregati e senza join esterni, includendo le interrogazioni nidificate.

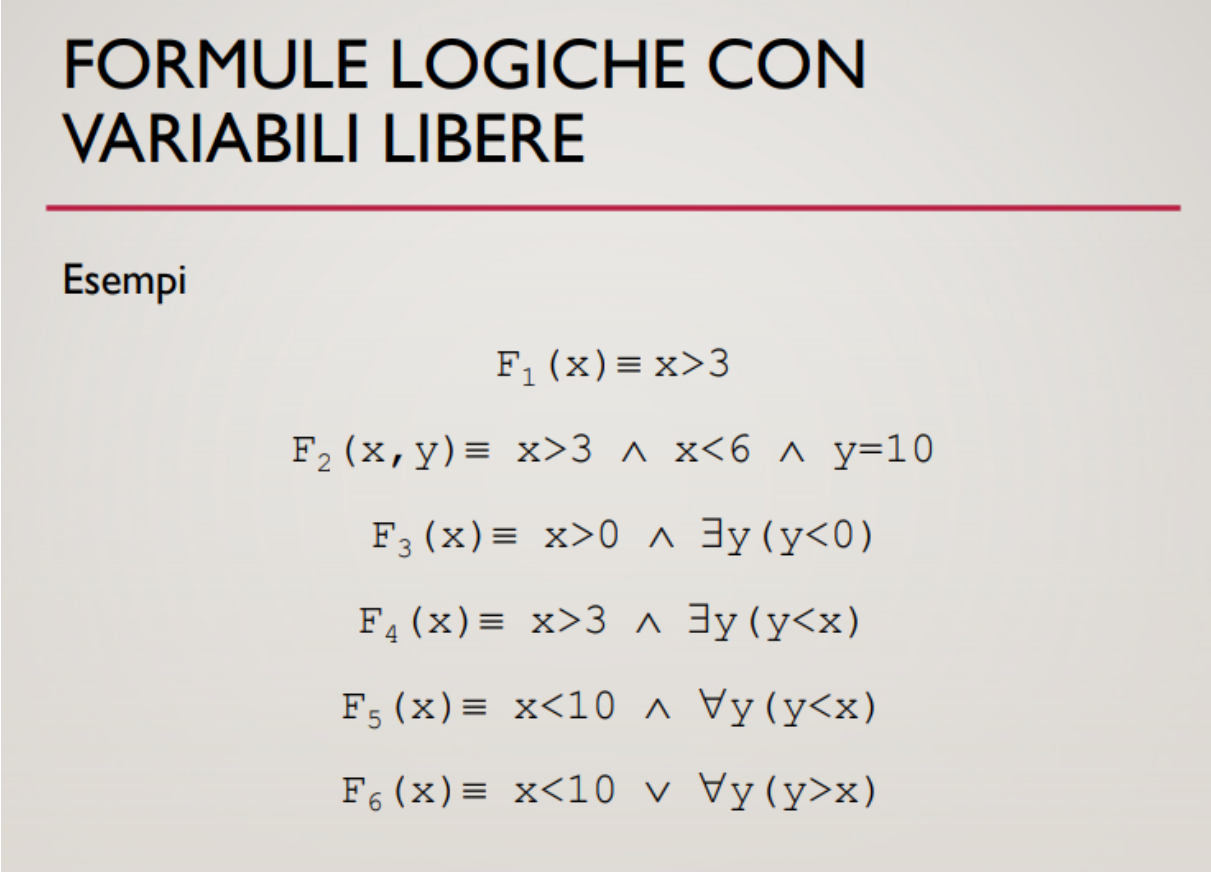



Figura 67: Esempi di formule logiche con variabili libere

**Interpretazione** L'interpretazione consiste nella scelta degli insiemi da cui prelevare i valori da assegnare alle variabili. Ad esempio, una formula può essere interpretata sull'insieme degli interi $\mathbb{Z}$ oppure sull'insieme dei numeri naturali $\mathbb{N}$.

$$F_1(x) \equiv x>3$$

$$F_2(x,y) \equiv x>3 \wedge x<6 \wedge y=10$$

$$F_3(x) \equiv x>0 \wedge \exists y(y<0)$$

$$F_4(x) \equiv x>3 \wedge \exists y(y<x)$$

$$F_5(x) \equiv x<10 \wedge \forall y(y<x)$$

$$F_6(x) \equiv x<10 \vee \forall y(y>x)$$

Figura 68: Formule logiche con variabili libere

**Sintassi** Le espressioni del calcolo relazionale sulle tuple hanno la seguente forma:

$$\{T \mid L \mid F\}$$

dove:

- **Target list** ($T$): definisce lo schema della relazione risultato e specifica come le tuple del risultato sono ottenute a partire dalle variabili libere.
- **Range list** ($L$): definisce le variabili libere e le associa alle relazioni della base di dati coinvolte nell'interrogazione.
- **Formula** ($F$): specifica la condizione logica che deve essere soddisfatta dalle variabili libere, ossia dalle tuple coinvolte nell'interrogazione.

## 8.1 Sintassi

**Sintassi Target List** La **target list** $T$ è una lista di elementi separati da virgole:

$$e_1, \ldots, e_n$$

dove ogni elemento $e_i$ può essere:

- $Y : x.(Z)$, dove $Y$ e $Z$ sono sequenze di attributi con la stessa cardinalità e $x$ è una variabile libera.
- $x.(Z)$, dove $Z$ è una sequenza di attributi e $x$ è una variabile libera (equivalente a $Z : x.(Z)$).
- $x.*$, dove $x$ è una variabile libera; in questo caso gli attributi sono tutti quelli della relazione associata alla variabile $x$ nella *range list* $L$.

**Sintassi range list** La **range list** $L$ è una lista di elementi separati da virgole:

$$r_1, \ldots, r_m$$

dove ogni elemento $r_i$ ha la seguente forma:

$$x_j(R)$$

dove $x_j$ è una variabile libera e $R$ è il nome di una relazione dello schema della base di dati.

Per ogni variabile libera della formula $F$ esiste uno e un solo elemento $r_i$ nella *range list* $L$.

**Sintassi della formula** La formula $F$ può essere:

**Formula atomica**

- $x.A\ \theta\ c$ oppure
- $x_1.A_1\ \theta\ x_2.A_2$

dove $x, x_1, x_2$ sono variabili, $A, A_1, A_2$ sono attributi, $c$ è una costante e $\theta$ è un operatore di confronto appartenente all'insieme:

$$\{=, \neq, >, <, \geq, \leq\}.$$

**Formula non atomica**

- se $f_1$ e $f_2$ sono formule, allora anche

  $$f_1 \wedge f_2, \quad f_1 \vee f_2, \quad \neg f_1, \quad \neg f_2$$

  sono formule;
- se $f$ è una formula, allora anche

  $$\forall x(R)\,(f) \quad \text{e} \quad \exists x(R)\,(f)$$

  sono formule.

**Notazione** Variabili libere e legate nelle formule:

- Nelle formule atomiche tutte le variabili sono libere.
- Nelle formule $\forall x\,(f)$ e $\exists x\,(f)$, la variabile $x$ è una variabile legata, mentre tutte le altre variabili mantengono lo stato (libere o legate) che avevano in $f$.
- Le congiunzioni, le disgiunzioni e la negazione non modificano l'insieme delle variabili libere o legate di una formula.

## 8.2 Semantica

**Semantica** Le formule vengono interpretate sull'istanza corrente della base di dati

$$db = \{r_1(x_1, \ldots, x_n), \ldots, r_m(x_1, \ldots, x_n)\}.$$

Ogni formula contiene un certo numero di variabili libere

$$f(x_1, \ldots, x_k).$$

Data una ennupla di tuple $(t_1, \ldots, t_k)$, tale ennupla rende vera la formula quando, sostituendo le tuple $t_1, \ldots, t_k$ alle variabili libere $x_1, \ldots, x_k$, la formula risulta soddisfatta.

Nel caso delle formule atomiche, una formula del tipo

$$x_{i,A} \, \theta \, x_{j,B}$$

è vera sulle tuple $(t_1, \ldots, t_k)$ se il confronto $t_{i,A} \, \theta \, t_{j,B}$ è soddisfatto; analogamente,

$$x_{i,A} \, \theta \, c$$

è vera se il confronto $t_{i,A} \, \theta \, c$ è soddisfatto, dove $\theta$ è un operatore di confronto.

Le formule non atomiche ottenute mediante i connettivi logici $\wedge$, $\vee$, $\neg$, $\Rightarrow$ sono vere secondo le usuali definizioni di verità dei connettivi logici.

Per quanto riguarda i quantificatori, la formula

$$\exists x_k \, (f)$$

è vera sulle tuple $(t_1, \ldots, t_{k-1})$ se esiste almeno una tupla $t_k$ nell'istanza di $R_k$ tale che, sostituendo $(t_1, \ldots, t_k)$ alle variabili $(x_1, \ldots, x_k)$, la formula $f$ risulta vera. Analogamente, la formula

$$\forall x_k \, (f)$$

è vera sulle tuple $(t_1, \ldots, t_{k-1})$ se per ogni tupla $t_k$ nell'istanza di $R_k$ la formula $f$ risulta vera quando si sostituiscono le tuple $(t_1, \ldots, t_k)$ alle variabili $(x_1, \ldots, x_k)$.

**Interpretazione di un'espressione del calcolo come interrogazione** Un'espressione del calcolo relazionale sulle tuple può essere interpretata come un'interrogazione della forma

$$Q = \{ \, Y_1 : x_1(Z_1), \ldots, Y_k : x_k(Z_k) \mid x_1(R_1), \ldots, x_n(R_n) \mid f(x_1, \ldots, x_n) \, \}.$$

La valutazione dell'interrogazione $Q$ produce una relazione risultato $R$ tale che:

- lo schema di $R$ è
$$Y = Y_1 \cup \cdots \cup Y_k;$$
- $R$ contiene tutte le tuple $t$ su $Y$ per le quali esiste una ennupla di tuple
$$(t_1, \ldots, t_n) \in R_1 \times \cdots \times R_n$$
che genera $t$ e che, sostituita alle variabili libere $(x_1, \ldots, x_n)$, rende vera la formula $f$.

## 8.3 Calcolo relazionale sulle tuple

Il calcolo relazionale sulle tuple non è equivalente né all'algebra relazionale né al calcolo relazionale sui domini, in quanto l'operatore di unione tra due relazioni non è esprimibile nel calcolo relazionale sulle tuple. Tuttavia, nel calcolo relazionale sulle tuple è possibile rappresentare sia l'intersezione sia la differenza tra due relazioni.

<u>Differenza</u> tra due relazioni $R_1$ e $R_2$ di schema $X = (A_1, \ldots, A_n)$

$$R_1 - R_2 \equiv \{x.* \mid x(R_1) \mid \neg\exists y(R_2)\ (x.A_1 = y.A_1 \wedge \ \ldots\ \wedge x.A_n = y.A_n)\}$$

<u>Intersezione</u> tra due relazioni $R_1$ e $R_2$ di schema $X = (A_1, \ldots, A_n)$

$$R_1 \cap R_2 \equiv \{x.* \mid x(R_1) \mid \exists y(R_2)\ (x.A_1 = y.A_1 \wedge \ \ldots\ \wedge x.A_n = y.A_n)\}$$

oppure

$$R_1 \cap R_2 \equiv \{x.* \mid x(R_1), y(R_2) \mid x.A_1 = y.A_1 \wedge \ \ldots\ \wedge x.A_n = y.A_n\}$$

Figura 69: differenza e intersezione

<u>Unione</u> tra due relazioni $R_1$ e $R_2$ di schema $X = (A_1, \ldots, A_n)$

$$R_1 \cup R_2 \equiv \{????\mid x(R_1), y(R_2) \mid true\}$$

<u>L'unione non è esprimibile in quanto non è possibile associare una variabile a più relazioni della base di dati.</u>

Figura 70: unione

## Esempio

TRENO(**Numero**, **Categoria**, **Partenza**, **Arrivo**, **Destinazione**)

FERMATA(**Treno**, **Stazione**, **Orario**)

Figura 71: Progettazione Logica

TRENO(**Num**, **Cat**, **Part**, **Arr**, **Dest**)
FERMATA(**Treno**, **Staz**, **Orario**)

$Q_0$: si richiede di restituire il numero e l'ora di partenza dei treni Freccia Rossa (Cat = 'FR') con destinazione 'Milano Centrale'

```
Q0 = {Numero,Orario_partenza: x.(Num,Part)|
        x(TRENO)|
        x.Cat='FR'∧ x.Dest='Milano Centrale'}
```

Figura 72: $Q_0$

TRENO(**Num**, **Cat**, **Part**, **Arr**, **Dest**)
FERMATA(**Treno**, **Staz**, **Orario**)

$Q_1$: Trovare l'elenco di tutte le fermate dei treni regionali (Cat='REG'), riportando nel risultato il numero del treno e la stazione di fermata.

```
Q1 = {Numero,Fermata: y.(Treno,Staz)|
        x(TRENO),y(FERMATA)|
        x.Cat='REG'∧ x.Num=y.Treno}
```

Figura 73: $Q_1$

TRENO(**Num**, **Cat**, **Part**, **Arr**, **Dest**)
FERMATA(**Treno**, **Staz**, **Orario**)

$Q_2$: Trovare l'elenco di tutti i treni Freccia Rossa che fermano a Venezia Mestre riportando: il numero del treno, l'orario di partenza e l'orario di arrivo a Venezia Mestre.

```
Q2 ≡ {Numero,Partenza: x.(Num,Part), Arrivo: y.(Orario)|
      x(TRENO),y(FERMATA)| x.Cat='FR' ∧ x.Num=y.Treno ∧
      y.Staz='Venezia Mestre'}
```

Figura 74: $Q_2$

TRENO(**Num**, **Cat**, **Part**, **Arr**, **Dest**)
FERMATA(**Treno**, **Staz**, **Orario**)

$Q_3$: Trovare la destinazione e l'orario di partenza dei treni che fermano a Padova.

```
Q3 ≡ {Destinazione,Partenza: x.(Dest,Part)|
      x(TRENO)| ∃y(FERMATA)(x.Num=y.Treno ∧
      y.Staz='Padova')}      oppure

Q3 ≡ {Destinazione,Partenza: x.(Dest,Part)|
      x(TRENO),y(FERMATA)| x.Num=y.Treno ∧
      y.Staz='Padova')}
```

Figura 75: $Q_3$

TRENO(**Num**, **Cat**, **Part**, **Arr**, **Dest**)
FERMATA(**Treno**, **Staz**, **Orario**)

$Q_4$: Trovare la destinazione e l'orario di partenza dei treni che non fermano a Padova.

```
Q3 ≡ {Destinazione,Partenza: x.(Dest,Part) |
      x(TRENO) | ¬∃y(FERMATA)(x.Num=y.Treno ∧
      y.Staz='Padova')}
```

Figura 76: $Q_4$

Il calcolo relazionale sulle tuple è equivalente al linguaggio di interrogazione adottato dallo standard SQL2, se si considera la forma base delle interrogazioni SQL, ovvero senza l'uso di operatori aggregati e di raggruppamenti. In tale contesto, ogni espressione del calcolo relazionale sulle tuple è direttamente traducibile in una interrogazione SQL di base.

In particolare, un'espressione del calcolo della forma

$$\{\, T \mid L \mid F \,\}$$

è in diretta corrispondenza con la forma base delle interrogazioni SQL:

`SELECT` $T$ `FROM` $L$ `WHERE` $F$.

La *target list* $T$ individua gli attributi da restituire nel risultato dell'interrogazione ed è mappata sulla clausola `SELECT`; la *range list* $L$ specifica le relazioni coinvolte ed è mappata sulla clausola `FROM`; infine, la formula $F$ esprime le condizioni di selezione ed è mappata sulla clausola `WHERE`.

# 9 Linguaggi di interrogazione nei sistemi NoSql

Nei sistemi NoSqlspesso non sono disponibili veri e propri linguaggi di interrogazione come nei sistemi relazionali. Solo i sistemi document-based forniscono strumenti simili all'algebra e al calcolo relazionale per l'accesso ai dati.

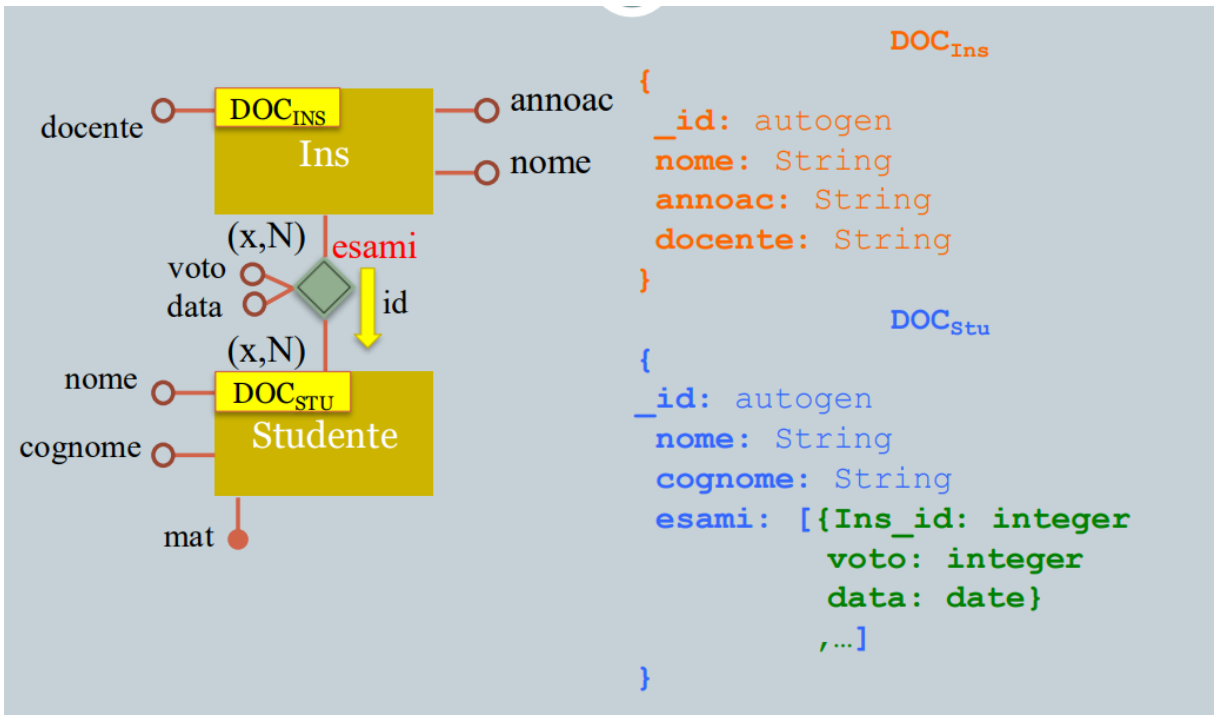



Figura 77: esempio sistemi document-store

La maggior parte del linguaggio di interrogazione di MongoDB è realizzato attraverso il metodo find.

```
db.studenti.insertMany([
{_id: "VR00010", nome: "Mario", cognome: "Rossi"
 esami: [{ins: "Basi di dati", voto: 22, data: "1/7/2016" },
    {ins: "Algebra", voto: 26, data: "5/7/2015" }]}
{_id: "VR00011", nome: "Maria", cognome: "Bianchi"
 esami: [{ins: "Algebra", voto: 30, data: "1/7/2016" }]}
]);
```

Figura 78: esempio con mongo db del document based di prima

- Trovare tutti gli studenti:

```
db.studenti.find( {} )
```

- Trovare tutti gli studenti di cognome "Rossi":

```
db.studenti.find( {cognome: "Rossi"} )
```

- Trovare tutti gli studenti di cognome "Rossi" e nome "Mario":

```
db.studenti.find( {cognome: "Rossi", nome: "Mario"} )
```

- Trovare tutti gli studenti di cognome "Rossi" o "Bianchi":

```
db.studenti.find( {cognome:{$in:["Rossi","Bianchi"]}}
)
```

Figura 79: Interrogazioni Semplici

| Name | Description |
|---|---|
| $and | Joins query clauses with a logical **AND** returns all documents that match the conditions of both clauses. |
| $not | Inverts the effect of a query expression and returns documents that do *not* match the query expression. |
| $nor | Joins query clauses with a logical **NOR** returns all documents that fail to match both clauses. |
| $or | Joins query clauses with a logical **OR** returns all documents that match the conditions of either clause. |

```
{$or: [{Cognome: "Rossi"},
       {Cognome: "Bianchi"}]}

{$and: [{Cognome: "Rossi"},
        {Nome: "Paolo"}]}
```

```
{$and: [{Esami.voto: {$gte: 21}},
        {Esami.voto: {$lte: 26}}]
```

| Name | Description |
|---|---|
| $eq | Matches values that are equal to a specified value. |
| $gt | Matches values that are greater than a specified value. |
| $gte | Matches values that are greater than or equal to a specified value. |
| $in | Matches any of the values specified in an array. |
| $lt | Matches values that are less than a specified value. |
| $lte | Matches values that are less than or equal to a specified value. |
| $ne | Matches all values that are not equal to a specified value. |
| $nin | Matches none of the values specified in an array. |

Figura 80: Operatori di Confronto

**Interrogazioni su dati incapsulati**

- Trovare tutti gli studenti che hanno registrato almeno un esame con voto 30:

```
db.studenti.find( {esami.voto: 30} )
```

- Trovare tutti gli studenti che hanno registrato almeno un esame con voto maggiore di 22:

```
db.studenti.find( {esami.voto: { $gt : 22 } } )
```

Figura 81: Esempio di interrogazioni con gli operatori

**Interrogazioni con proiezione**

- Trovare il cognome degli studenti di nome "Mario":

```
db.studenti.find( {nome: "Mario"}, {cognome: 1} )
```

- Trovare il cognome degli studenti di nome "Mario" escludendo il campo _id:

```
db.studenti.find( {nome: "Mario"},
                  {cognome: 1, _id: 0} )
```

- Trovare la matricola e i voti di tutti gli studenti:

```
db.studenti.find( { }, {_id: 1, esami.voto: 1} )
```

Figura 82: Interrogazioni con proiezione

- Trovare tutti gli studenti che hanno ottenuto un voto maggiore di 28 nell'appello del 30/10/2025:

```
db.studenti.find(
      { esami: { $elemMatch: { data: '30/10/2025',
                               voto: { $gt : 22 } } } }
```

Figura 83: condizioni multiple

# 10 Tecnologie - Transazioni

## 10.1 Tecnologia dei DBMS

Un **DBMS** (Database Management System), o *sistema di gestione di basi di dati*, è un sistema software progettato per gestire collezioni di dati che siano:

- **Grandi** (di dimensioni tali da richiedere gestione su memoria secondaria);
- **Condivise** (accessibili da più utenti o applicazioni contemporaneamente);
- **Persistenti** (i dati sopravvivono all'esecuzione dei programmi che li utilizzano).

Un DBMS garantisce:

- **Affidabilità** (integrità e recupero dei dati in caso di guasti);
- **Privatezza e sicurezza** (controllo degli accessi e gestione dei permessi).

Essendo un sistema software, un DBMS deve essere:

- **Efficiente**, ottimizzando l'uso delle risorse (tempo di risposta e spazio);
- **Efficace**, fornendo strumenti adeguati per la gestione e l'organizzazione dei dati.

Le prestazioni e la qualità complessiva del sistema dipendono in modo significativo dalla **corretta progettazione della base di dati**.

Un DBMS:

- Estende le funzionalità del *file system*;
- Gestisce collezioni di dati memorizzate in **memoria secondaria**;
- Fornisce meccanismi di accesso, manipolazione e controllo dei dati.

**Base di dati:** una collezione organizzata di dati, gestita e controllata da un DBMS.

## 10.2 Architettura su 3 livelli dei DBMS

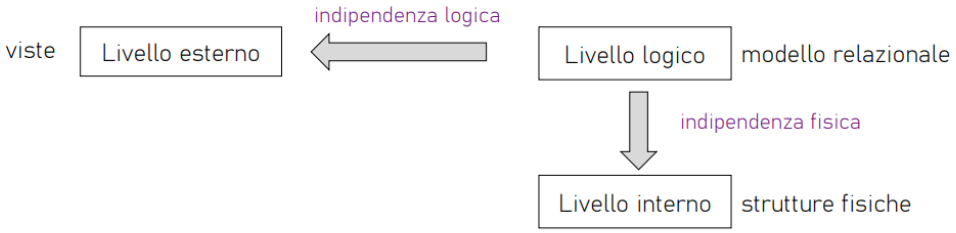



Figura 84: Architettura su 3 livelli dei DBMS

Un DBMS è organizzato secondo un'architettura a tre livelli: **esterno**, **logico** e **interno**. Il **livello esterno** descrive le viste degli utenti: ad esempio, uno studente può visualizzare solo i propri voti, mentre la segreteria può accedere ai dati di tutti gli studenti. Il **livello logico** definisce la struttura

concettuale della base di dati nel modello relazionale, ad esempio la tabella `Studenti(Matricola, Nome, Cognome, Corso)`, indipendentemente dalla memorizzazione fisica. Il **livello interno**, infine, descrive come i dati sono effettivamente memorizzati su disco, ad esempio in file organizzati in pagine con eventuali indici per ottimizzare le ricerche.

Questa separazione garantisce l'**indipendenza logica**, per cui modifiche allo schema logico non richiedono cambiamenti nelle viste esterne, e l'**indipendenza fisica**, per cui variazioni nell'organizzazione fisica dei dati non alterano lo schema logico.

## 10.3 Transazione

Una **transazione** è un'unità logica di lavoro eseguita da un programma applicativo che interagisce con una base di dati. È costituita da una sequenza di operazioni che devono essere considerate come un unico blocco indivisibile.

Una transazione deve garantire proprietà di **correttezza**, **robustezza** e **isolamento**, in modo da mantenere la consistenza della base di dati anche in presenza di errori o accessi concorrenti.

La caratteristica fondamentale di una transazione è il principio *all-or-nothing*: essa o termina con successo (**commit**) producendo effetti permanenti sulla base di dati, oppure viene annullata (**abort/rollback**) senza lasciare alcuna modifica.

```
<transazione> → begin transaction
                <programma>
                end transaction
<programma> → ⟨<istruzione> | commit work | rollback work⟩
              {⟨<istruzione> | commit work | rollback work⟩}
```

Figura 85: Sintassi di una transazione

Una **transazione** si dice *ben formata* se rispetta le seguenti condizioni:

- Inizia esplicitamente con un'istruzione `BEGIN TRANSACTION`.
- Termina con un'istruzione `END TRANSACTION`.
- Durante la sua esecuzione viene raggiunto un punto di terminazione esplicito, ovvero un'istruzione `COMMIT WORK` oppure `ROLLBACK WORK`.
- Dopo l'esecuzione di `COMMIT` o `ROLLBACK` non vengono effettuati ulteriori accessi o modifiche alla base di dati all'interno della stessa transazione.

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

Figura 86: Esempio transazione ben formata

### 10.3.1 Proprietà delle transazioni

Una transazione in un DBMS deve rispettare quattro proprietà fondamentali, note come proprietà **ACID**:

- **Atomicità (Atomicity)**: una transazione è un'unità di esecuzione indivisibile. Essa viene eseguita completamente oppure non viene eseguita affatto.

  Implicazioni:

  - Se una transazione viene interrotta prima del `commit`, il lavoro eseguito fino a quel momento deve essere annullato, ripristinando lo stato della base di dati precedente all'inizio della transazione.
  - Se una transazione viene interrotta dopo l'esecuzione del `commit` (commit completato con successo), il sistema deve garantire che gli effetti della transazione siano permanentemente registrati nella base di dati.

- **Consistenza (Consistency)**: l'esecuzione di una transazione non deve violare i vincoli di integrità della base di dati.

  Implicazioni:

  - **Verifica immediata**: il controllo dei vincoli viene effettuato durante l'esecuzione della transazione. Se un vincolo viene violato, viene annullata solo l'ultima operazione eseguita e il sistema restituisce all'applicazione una segnalazione di errore. L'applicazione può quindi reagire alla violazione.
  - **Verifica differita**: il controllo dei vincoli viene effettuato al momento del `commit`. Se un vincolo di integrità risulta violato, l'intera transazione viene abortita senza possibilità per l'applicazione di reagire alla violazione.

- **Isolamento (Isolation)**: l'esecuzione di una transazione deve essere indipendente dalla contemporanea esecuzione di altre transazioni.

  Implicazioni:

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

- **Persistenza (Durability)**: l'effetto di una transazione che ha eseguito il `commit` non deve andare perso.

  Implicazioni:

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

## 10.4 Architettura di un DBMS

Un DBMS è composto da diversi moduli che gestiscono le varie funzionalità necessarie all'esecuzione delle transazioni.

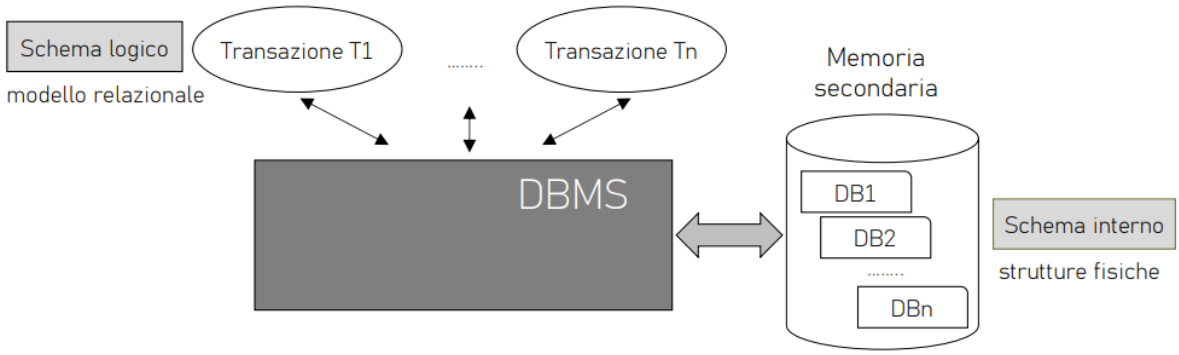



Figura 87: Architettura di riferimento di un DBMS

La figura rappresenta l'interazione tra le transazioni degli utenti, il DBMS e la memoria secondaria. Le transazioni (T1, ..., Tn) operano sullo **schema logico** del database secondo il modello relazionale e inviano le loro operazioni al DBMS. Il **DBMS** funge da intermediario, gestendo l'esecuzione delle transazioni e garantendo le proprietà di correttezza del sistema. I dati sono memorizzati nella **memoria secondaria** (DB1, DB2, ..., DBn) secondo lo **schema interno**, che descrive le strutture fisiche utilizzate per organizzare e memorizzare i dati.

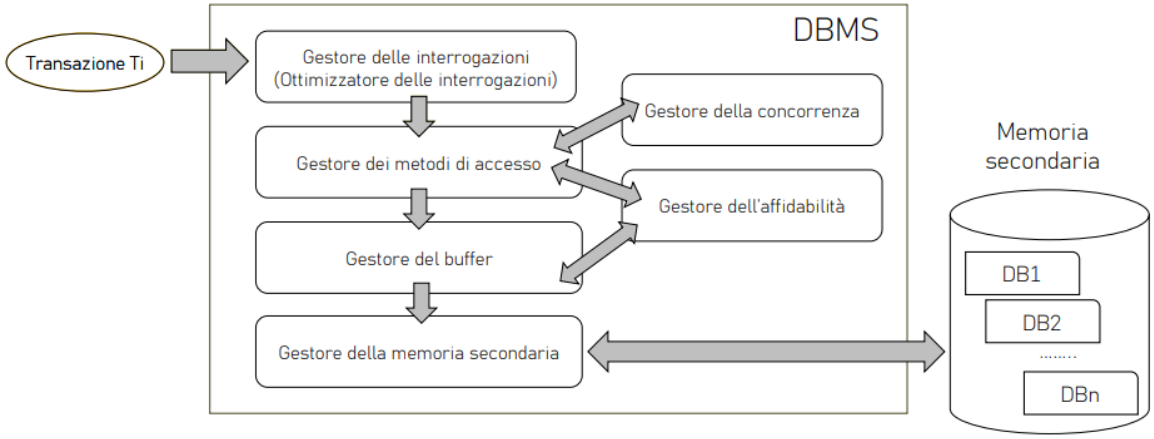



Figura 88: Moduli principali di un DBMS

La figura mostra i principali moduli che compongono l'architettura interna di un DBMS e il modo in cui collaborano durante l'esecuzione di una transazione.

Una transazione $T_i$ invia le proprie interrogazioni al **gestore delle interrogazioni** (query processor), che analizza e ottimizza le query per determinare il piano di esecuzione più efficiente. Il **gestore dei metodi di accesso** stabilisce come accedere ai dati, utilizzando strutture come indici o scansioni delle tabelle.

Il **gestore del buffer** si occupa della gestione delle pagine di dati in memoria principale, mentre il **gestore della memoria secondaria** gestisce l'accesso fisico ai dati memorizzati su disco.

Il **gestore della concorrenza** controlla l'esecuzione simultanea di più transazioni per evitare interferenze, mentre il **gestore dell'affidabilità** garantisce la correttezza delle operazioni anche in caso di guasti, assicurando il mantenimento delle proprietà ACID.

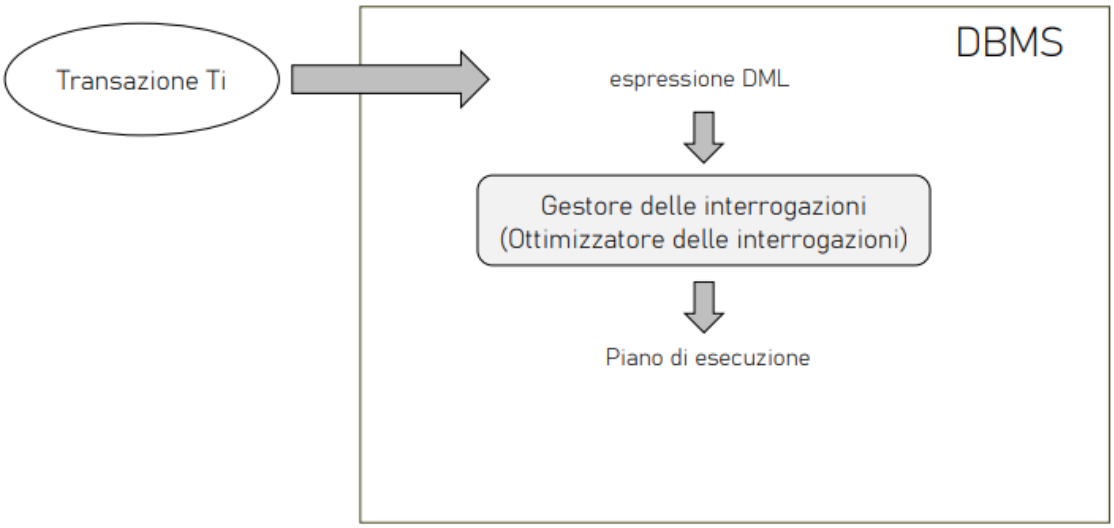



Figura 89: fase di elaborazione di una query

Una transazione $T_i$ invia al DBMS un'espressione del linguaggio DML. La query viene elaborata dal **gestore delle interrogazioni** (query processor), che analizza e ottimizza l'interrogazione. Il risultato di questo processo è un **piano di esecuzione**, ovvero una sequenza di operazioni che il DBMS esegue per ottenere il risultato richiesto in modo efficiente.

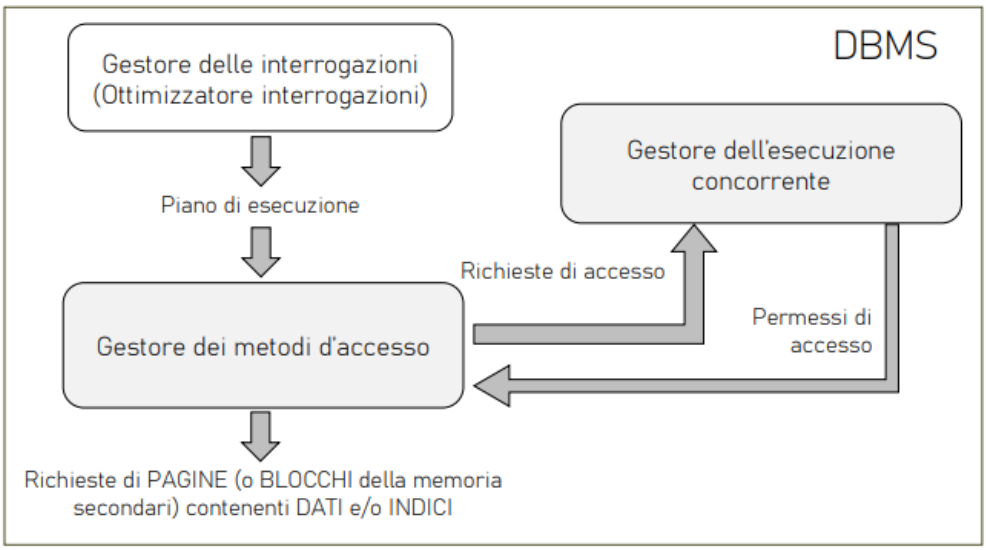



Figura 90: gestione delle interrogazioni

Il piano di esecuzione prodotto dal gestore delle interrogazioni viene eseguito dal gestore dei metodi di accesso, che richiede le pagine di dati o indici dalla memoria secondaria. Le richieste di accesso ai dati vengono controllate dal gestore dell'esecuzione concorrente, che verifica la compatibilità con le altre transazioni in esecuzione e concede i permessi di accesso. In questo modo il DBMS garantisce l'esecuzione corretta delle transazioni concorrenti.

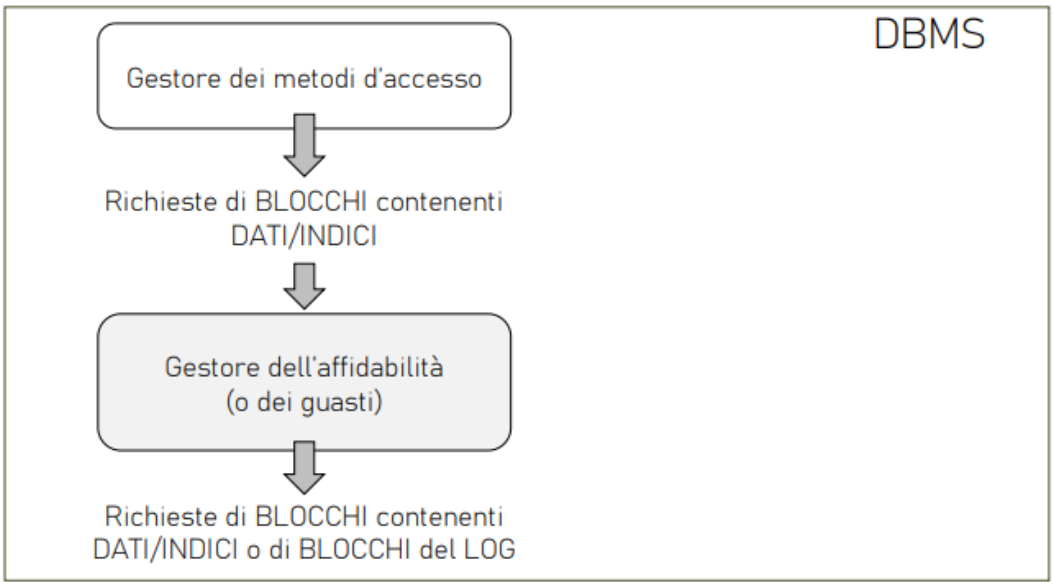



Figura 91: fase di recovery

Le richieste di accesso ai blocchi contenenti dati o indici, generate dal gestore dei metodi di accesso, vengono gestite dal **gestore dell'affidabilità** (recovery manager). Questo modulo ha il compito di garantire la correttezza del database anche in caso di guasti, utilizzando il **log delle transazioni**. Il log registra le operazioni eseguite dalle transazioni e permette al sistema di ripristinare lo stato corretto della base di dati attraverso operazioni di *redo* e *undo*.

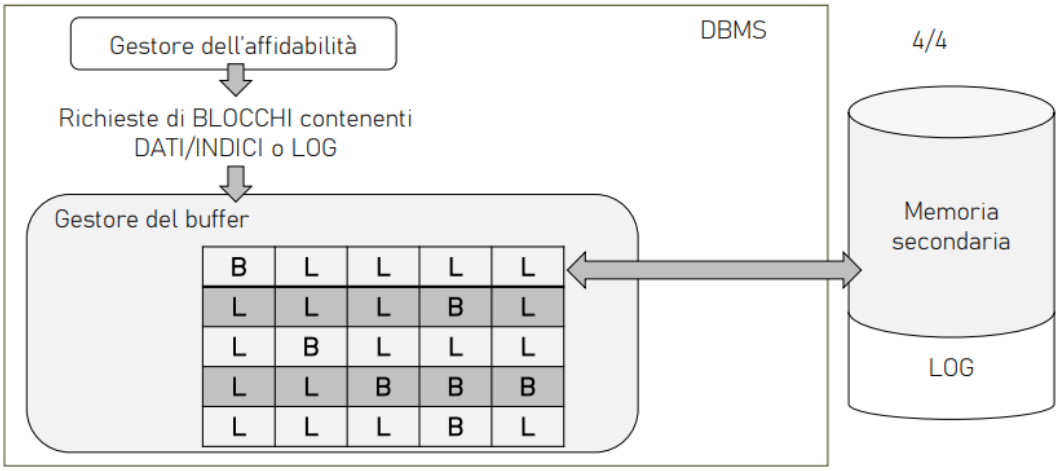



Figura 92: fase di gestione della memoria principale tramite il buffer manager quando accede ai dati sul disco

Le richieste di accesso ai blocchi contenenti dati, indici o record del log, generate dal gestore dell'affidabilità, vengono gestite dal **gestore del buffer**. Questo modulo mantiene in memoria principale una copia delle pagine utilizzate più frequentemente, riducendo gli accessi alla memoria secondaria. Quando necessario, il buffer manager carica nuove pagine dal disco o scrive sul disco le pagine modificate.

- Quali moduli contribuiscono a garantire le proprietà delle transazioni?

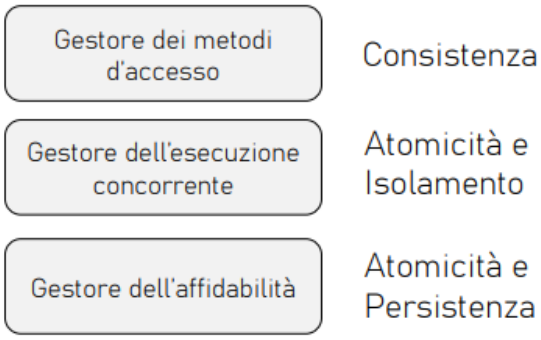



Figura 93:

# 11 Strutture fisiche e accesso ai dati

Per comprendere il funzionamento interno di un DBMS è necessario analizzare come i dati vengono effettivamente memorizzati e gestiti a livello fisico.

Una base di dati risiede in memoria secondaria per due motivi fondamentali. In primo luogo, per una questione di **dimensione**: le basi di dati sono generalmente molto grandi e non possono essere contenute interamente in memoria centrale. In secondo luogo, per garantire la **persistenza**: i dati devono sopravvivere all'esecuzione dei programmi e agli spegnimenti del sistema. A differenza della memoria centrale, che è volatile, la memoria secondaria consente di conservare le informazioni in modo permanente nel tempo.

La memoria secondaria presenta però alcune caratteristiche che influenzano direttamente il modo in cui i dati vengono gestiti. Essa non è direttamente accessibile dai programmi e i dati sono organizzati in **blocchi** (o pagine). Di conseguenza, non è possibile leggere una singola informazione in modo puntuale: quando si accede a un dato, è necessario caricare in memoria centrale l'intero blocco che lo contiene. Le uniche operazioni possibili sulla memoria secondaria sono quindi la **lettura** e la **scrittura** di blocchi.

Un aspetto cruciale è che tali operazioni hanno un costo molto elevato rispetto agli accessi alla memoria centrale, con una differenza di diversi ordini di grandezza. Per questo motivo, nella valutazione delle prestazioni di un DBMS non si considera il numero di operazioni di calcolo, ma il **numero di accessi alla memoria secondaria**, che rappresenta il vero collo di bottiglia del sistema.

Per ridurre il numero di accessi al disco interviene il **gestore del buffer** (buffer manager), che ha il compito di gestire la memoria centrale. Questo modulo mantiene in RAM i blocchi più utilizzati, evitando accessi ripetuti alla memoria secondaria. Quando un blocco non è presente in memoria, il buffer manager richiede il caricamento dal disco; analogamente, quando un blocco viene modificato, si occupa della sua scrittura in memoria secondaria.

Il **gestore della memoria secondaria** è invece responsabile dell'accesso fisico ai dati. Esso gestisce i file del database e si occupa delle operazioni di lettura e scrittura dei blocchi, interagendo direttamente con il file system del sistema operativo. In questo contesto, il gestore del buffer e quello della memoria secondaria collaborano strettamente: il primo richiede i blocchi necessari, mentre il secondo li recupera dal disco.

Infine, il **gestore dell'affidabilità** interviene per garantire la correttezza del sistema anche in presenza di guasti. Questo modulo mantiene un log delle operazioni eseguite e consente il ripristino dello stato corretto del database tramite operazioni di *redo* e *undo*, assicurando la persistenza delle transazioni completate. L'interazione tra memoria secondaria e memoria centrale avviene tramite il **gestore del buffer**, che si occupa di trasferire i dati dal disco alla RAM sotto forma di blocchi.

In particolare, il buffer manager carica uno o più blocchi dalla memoria secondaria in **pagine** della memoria centrale. Sebbene non vi sia una corrispondenza perfetta, si assume generalmente che una pagina in memoria centrale corrisponda a un blocco in memoria secondaria.

Il buffer rappresenta quindi un'area di memoria centrale condivisa tra più applicazioni, il cui obiettivo è ridurre il numero di accessi alla memoria secondaria.

Il gestore del buffer ha il compito di:

- caricare i blocchi dalla memoria secondaria alla memoria centrale;
- restituire riferimenti (puntatori) ai dati richiesti;

- scrivere i blocchi modificati dalla memoria centrale al disco.

Per ottimizzare le prestazioni, il buffer manager adotta diverse **politiche di gestione della memoria**, che riguardano sia le operazioni di lettura sia quelle di scrittura.

Nel caso della **lettura di un blocco**, si distinguono due situazioni:

- se il blocco è già presente nel buffer (*buffer hit*), non viene effettuato alcun accesso alla memoria secondaria e viene restituito direttamente il riferimento alla pagina in memoria centrale;
- se il blocco non è presente (*buffer miss*), esso viene caricato dal disco e successivamente reso disponibile.

Nel caso della **scrittura**, il DBMS può adottare una strategia di **scrittura differita** (*delayed write*): il blocco modificato viene aggiornato inizialmente solo in memoria centrale e la scrittura su disco viene posticipata. Questa scelta migliora le prestazioni riducendo il numero di operazioni di I/O, ma introduce il rischio di perdita dei dati in caso di guasto. Per questo motivo interviene il **gestore dell'affidabilità**, che tramite il log consente di ripristinare le informazioni corrette.

Le politiche di gestione del buffer si basano sul **principio di località**, secondo cui i dati recentemente utilizzati hanno un'alta probabilità di essere riutilizzati nel breve periodo.

Un'importante osservazione empirica è la cosiddetta **regola dell'80/20**: circa il 20% dei dati è responsabile dell'80% degli accessi. Questo implica che, mantenendo in memoria centrale una piccola parte dei dati più frequentemente utilizzati, è possibile ottenere un notevole miglioramento delle prestazioni.

Per questo motivo, politiche semplici come la LIFO (Last In First Out) risultano inefficaci, poiché non tengono conto del principio di località. Al contrario, strategie più evolute (come LRU, Least Recently Used) privilegiano il mantenimento in memoria dei dati utilizzati più di recente.

## 11.1 Rappresentazione logica del buffer

Il contenuto del buffer può essere rappresentato in maniera logica come una **matrice di pagine**, tutte aventi la stessa dimensione.

Ogni pagina del buffer può contenere un blocco $B$ proveniente dalla memoria secondaria. Si assume quindi, in prima approssimazione, una corrispondenza tra pagine in memoria centrale e blocchi in memoria secondaria.

Per ogni pagina del buffer che contiene un blocco, vengono mantenute due importanti **variabili di stato**:

- un **contatore** $i$, che indica quante transazioni stanno attualmente utilizzando quella pagina;
- un **bit di stato** $j$ (dirty bit), che indica se la pagina è stata modificata:
  - $j = 1$ se la pagina è stata modificata rispetto al contenuto in memoria secondaria;
  - $j = 0$ se la pagina è identica a quella presente su disco.

Il contatore $i$ è fondamentale per la gestione della concorrenza, in quanto impedisce la rimozione di una pagina ancora in uso da parte di una o più transazioni.

Il bit di stato $j$ è invece essenziale per la gestione delle scritture: una pagina modificata deve essere salvata su memoria secondaria prima di essere eventualmente rimossa dal buffer.

In Figura 94 è mostrata una possibile rappresentazione del buffer come matrice di pagine, in cui

alcune pagine contengono blocchi della memoria secondaria e sono associate ai rispettivi stati.

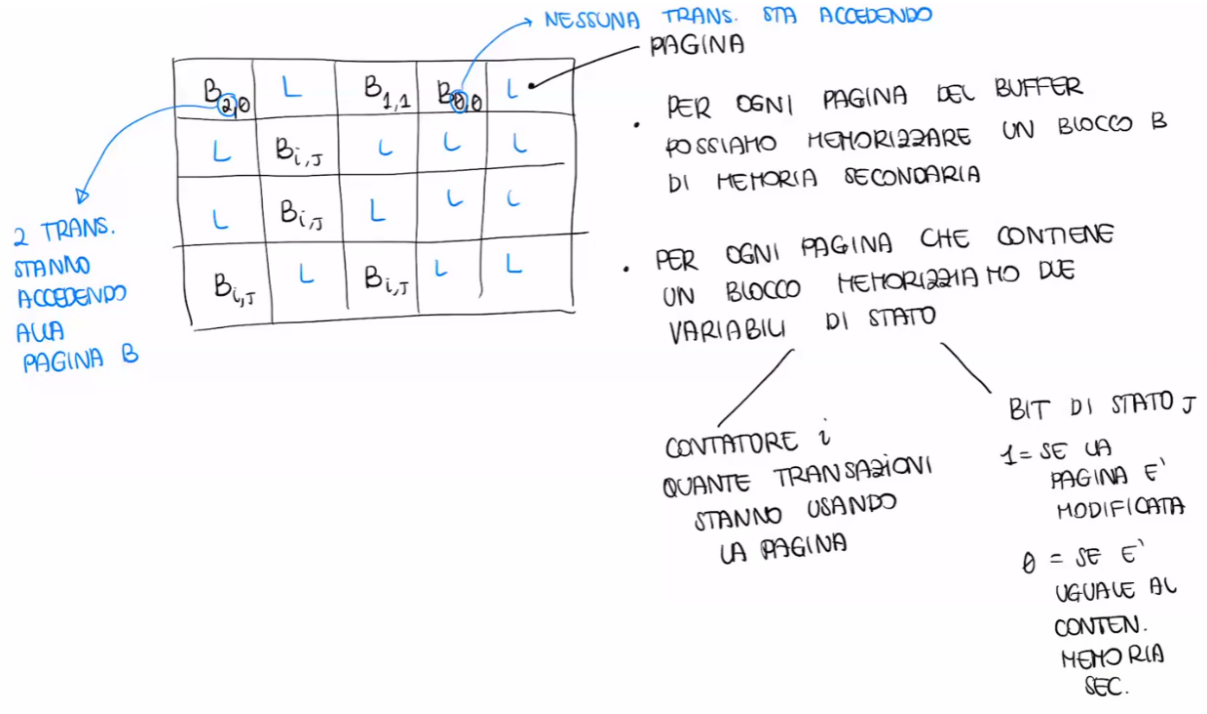



Figura 94: Rappresentazione logica del buffer

## 11.2 Primitive del gestore del buffer

Il gestore del buffer mette a disposizione un insieme di primitive utilizzate principalmente dal gestore dei metodi di accesso per richiedere e gestire i blocchi in memoria centrale.

1. **FIX**

   La primitiva FIX viene utilizzata per richiedere l'accesso a un blocco della memoria secondaria. Essa restituisce un puntatore alla pagina del buffer che contiene il blocco richiesto.

   Il comportamento può essere descritto nei seguenti casi:

   - **Caso 1: blocco già presente nel buffer (buffer hit)**
     Se il blocco è già caricato, il sistema restituisce semplicemente il puntatore alla pagina corrispondente.
   - **Caso 2: blocco non presente (buffer miss)**
     Il sistema deve caricare il blocco in una pagina del buffer.
     - Se esiste una **pagina libera** (indicata, ad esempio, con etichetta $L$ oppure con contatore $i = 0$), essa viene utilizzata.
     - Se non esistono pagine libere, è necessario scegliere una pagina da sostituire secondo una politica di rimpiazzamento, come:
       * **LRU** (Least Recently Used): rimuove la pagina meno recentemente utilizzata;
       * **FIFO** (First In First Out): rimuove la pagina caricata da più tempo.

   Prima di riutilizzare una pagina, si verifica il valore del **dirty bit** $j$:

  - se $j = 1$, la pagina deve essere scritta su disco tramite la primitiva *FLUSH*;
  - se $j = 0$, può essere riutilizzata immediatamente.

- **Caso 3: Gestione in assenza di pagine disponibili**
  Se tutte le pagine sono in uso (cioè con contatore $i > 0$), il DBMS può adottare due strategie:
  - **NO-STEAL**: la transazione viene sospesa finché non si libera una pagina;
  - **STEAL**: il sistema può sottrarre una pagina a un'altra transazione, anche se ancora in uso.

Dopo il caricamento del blocco, il contatore $i$ della pagina viene incrementato.

2. **UNFIX**

   La primitiva UNFIX indica che una transazione ha terminato di utilizzare un blocco. Di conseguenza, il contatore $i$ associato alla pagina viene decrementato.

3. **SETDIRTY**

   Questa primitiva viene utilizzata quando una transazione modifica il contenuto di una pagina. Il dirty bit $j$ viene posto a 1, indicando che la copia in memoria centrale è diversa da quella in memoria secondaria.

4. **FLUSH**

   La primitiva FLUSH consente di scrivere su memoria secondaria una pagina modificata. Questa operazione può avvenire in modo asincrono ed è utilizzata dal gestore del buffer per liberare spazio.

   Al termine dell'operazione, il dirty bit $j$ viene posto a 0.

5. **FORCE**

   La primitiva FORCE è una variante della FLUSH, utilizzata per garantire la scrittura immediata di una pagina su memoria secondaria. Essa è tipicamente richiesta dal gestore dell'affidabilità per assicurare la persistenza dei dati.

   Anche in questo caso, al termine dell'operazione il dirty bit $j$ viene posto a 0.

## 11.3 Persistenza, scritture differite e log

Per migliorare le prestazioni, il DBMS utilizza spesso una strategia di **scritture differite**, in cui le modifiche ai dati vengono inizialmente applicate solo in memoria centrale e scritte su memoria secondaria in un secondo momento.

Questa tecnica introduce il rischio di perdita dei dati in caso di guasto. Per garantire la **persistenza** e l'**atomicità** delle transazioni, interviene il **gestore dell'affidabilità**, che utilizza una struttura fondamentale chiamata **log**.

Il log è un archivio persistente, memorizzato su memoria stabile (resistente ai guasti), in cui vengono registrate tutte le operazioni eseguite dal DBMS.

All'interno del log si distinguono due categorie principali di record:

- **Record di transazione**, che descrivono le operazioni effettuate sulle basi di dati:
  - $B(T)$: *Begin*, inizio della transazione $T$;

  - $C(T)$: *Commit*, completamento corretto della transazione;
  - $A(T)$: *Abort*, annullamento della transazione;
  - $I(T, O, AS)$: *Insert*, inserimento dell'oggetto $O$ con stato finale (*After State*);
  - $U(T, O, BS, AS)$: *Update*, modifica dell'oggetto $O$ dallo stato iniziale (*Before State*) allo stato finale;
  - $D(T, O, BS)$: *Delete*, eliminazione dell'oggetto $O$, memorizzando lo stato precedente.
- **Record di sistema**, che descrivono eventi del DBMS:
  - **DUMP**: indica un salvataggio completo della base di dati;
  - **CHECKPOINT** $CK(T_1, \ldots, T_n)$: registra l'insieme delle transazioni attive in un determinato momento.

L'obiettivo del log è consentire il **ripristino** del database attraverso due operazioni fondamentali:

- **UNDO**: annulla gli effetti di una transazione non completata, riportando i dati allo stato precedente ($BS$);
- **REDO**: ripristina gli effetti di una transazione completata, riportando i dati allo stato finale ($AS$).

Entrambe le operazioni soddisfano la cosiddetta **proprietà di idempotenza**:

$$UNDO(UNDO(A)) = UNDO(A), \quad REDO(REDO(A)) = REDO(A)$$

Il **checkpoint** è una tecnica utilizzata per ottimizzare il processo di recovery. Esso viene eseguito periodicamente dal DBMS e consiste in:

- sospendere temporaneamente le operazioni;
- forzare la scrittura dei dati modificati su memoria secondaria;
- registrare nel log le transazioni attive in quel momento;
- riprendere la normale esecuzione.

Grazie al checkpoint, in caso di guasto non è necessario analizzare l'intero log, ma solo la parte successiva all'ultimo checkpoint. Le principali regole di scrittura del file di log sono fondamentali per garantire la correttezza del sistema e permettere le operazioni di recovery.

1. **Regola WAL (Write-Ahead Logging)**
   I record di log devono essere scritti sul log prima dell'esecuzione delle corrispondenti operazioni sui dati. In particolare, prima che una modifica venga applicata alla memoria secondaria, il relativo record deve essere già stato registrato nel log. Questa regola garantisce la possibilità di effettuare operazioni di **UNDO**, permettendo di annullare modifiche non correttamente completate.
2. **Regola di precedenza del commit**
   I record di log relativi a una transazione devono essere scritti in modo persistente prima che venga effettuato il commit della transazione stessa. Solo dopo che il record di commit è stato salvato stabilmente, la transazione può essere considerata completata. Questa regola garantisce la possibilità di effettuare operazioni di **REDO**, permettendo di ripristinare le modifiche di transazioni completate anche in caso di guasto.

## 11.4 Algoritmo di ripresa in caso di guasto

Il DBMS deve essere in grado di gestire diverse tipologie di guasti che possono compromettere la correttezza dei dati. In particolare, si distinguono due categorie principali:

1. **Guasto di sistema**
   Si verifica quando viene perso il contenuto della memoria centrale (RAM), mentre la memoria secondaria rimane intatta. Esempi tipici sono:
   - interruzione dell'alimentazione elettrica;
   - crash del sistema operativo;
   - errore hardware o software che causa il riavvio del sistema.

   In questo caso si applica un algoritmo detto di **ripresa a caldo** (*warm restart*), che utilizza il log per:
   - eseguire il **REDO** delle transazioni che avevano effettuato il commit ma le cui modifiche potrebbero non essere ancora state scritte su disco;
   - eseguire l'**UNDO** delle transazioni non completate al momento del guasto.
2. **Guasto di dispositivo**
   Si verifica quando viene perso il contenuto della memoria secondaria, ad esempio a causa di:
   - guasto del disco;
   - corruzione dei dati su file system;
   - perdita totale o parziale dei dati memorizzati su disco.

   In questo caso si applica un algoritmo detto di **ripresa a freddo** (*cold restart*), che consiste in:
   - ripristino della base di dati a partire da una copia di backup (dump);
   - successiva applicazione del log per eseguire le operazioni di REDO necessarie.

   La ripresa a freddo può essere vista come un'estensione della ripresa a caldo, in quanto include anche le operazioni di recupero da backup oltre all'utilizzo del log.

### 11.4.1 Algoritmo ripresa a caldo

L'algoritmo di ripresa a caldo consente di ripristinare uno stato consistente della base di dati dopo un guasto di sistema, utilizzando le informazioni contenute nel log.

1. **Individuazione del checkpoint**
   Si accede al file di log e lo si analizza a ritroso fino a individuare il più recente record di checkpoint $CK(T_1, \ldots, T_n)$.
2. **Inizializzazione delle strutture dati**
   Si inizializzano due insiemi:
   - **UNDO**: insieme delle transazioni da annullare;
   - **REDO**: insieme delle transazioni da ripetere.

   L'inizializzazione avviene nel seguente modo:
   - UNDO viene inizializzato con le transazioni attive presenti nel checkpoint $(T_1, \ldots, T_n)$;

- REDO viene inizializzato come insieme vuoto.

3. **Analisi in avanti del log**
   A partire dal checkpoint, si scorre il log in avanti applicando le seguenti regole per ogni record:
   - se si incontra $B(T)$, si aggiunge $T$ all'insieme UNDO;
   - se si incontra $C(T)$, si sposta $T$ da UNDO a REDO;
4. **Fase di UNDO**
   Si percorre il file di log all'indietro (dalla fine verso il checkpoint) e, per ogni operazione incontrata, si verifica se la transazione $T$ a cui appartiene è presente nell'insieme UNDO.

   Se la transazione appartiene a UNDO, si deve annullare l'operazione eseguendo l'operazione inversa, utilizzando le informazioni contenute nel log:
   - $I(T, O, AS) \rightarrow$ si esegue una **DELETE** dell'oggetto $O$;
   - $D(T, O, BS) \rightarrow$ si esegue una **INSERT** dell'oggetto $O$ ripristinando lo stato precedente ($BS$);
   - $U(T, O, BS, AS) \rightarrow$ si riporta l'oggetto $O$ allo stato precedente ($BS$).

   Il processo continua fino a raggiungere il record $B(T)$ della transazione più vecchia presente nell'insieme UNDO.
5. **Fase di REDO**
   Si scorre il log in avanti e si rieseguono le operazioni delle transazioni presenti nell'insieme REDO, utilizzando le informazioni di *After State* ($AS$), garantendo che tutte le modifiche delle transazioni concluse siano effettivamente applicate.

### 11.4.2 Algoritmo ripresa a freddo

L'algoritmo di ripresa a freddo viene applicato in caso di guasto della memoria secondaria e prevede il ripristino della base di dati a partire da una copia di backup (dump), seguito dall'utilizzo del log.

I passaggi sono i seguenti:

1. **Ripristino dal dump**
   Si accede all'ultima copia di backup della base di dati (dump) e si ripristina la parte deteriorata o l'intera base di dati, riportandola a uno stato consistente precedente al guasto.
2. **Individuazione del punto nel log**
   Si accede al file di log e si individua il punto corrispondente al dump utilizzato, da cui riprendere l'analisi.
3. **Riesecuzione delle operazioni (REDO)**
   Si scorre il log in avanti a partire dal dump e si rieseguono tutte le operazioni (REDO) necessarie per riportare la base di dati allo stato più aggiornato possibile, limitatamente alle informazioni disponibili.
4. **Applicazione della ripresa a caldo**
   Una volta ricostruito lo stato della base di dati, si applica l'algoritmo di ripresa a caldo per garantire la consistenza finale, eseguendo eventuali operazioni di UNDO e REDO sulle transazioni non completamente gestite.

### Esercizio 1. Ripresa a caldo

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

Figura 95: Dati

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

PASSO 1: RISALGO IL LOG FINO ALL'ULTIMO CK

Figura 96: PASSO 1 : INDIVIDUAZIONE CK

- PASSO 1: RISALGO IL LOG
- PASSO 2: INIZIALIZZARE UNDO e REDO

UNDO = {T2, T3, T4} ⇝ INIZIALIZZATO con TRANS. CK
REDO = { }

Figura 97: PASSO 2: INIZIALIZZAZIONE DELLE STRUTTURE DATI

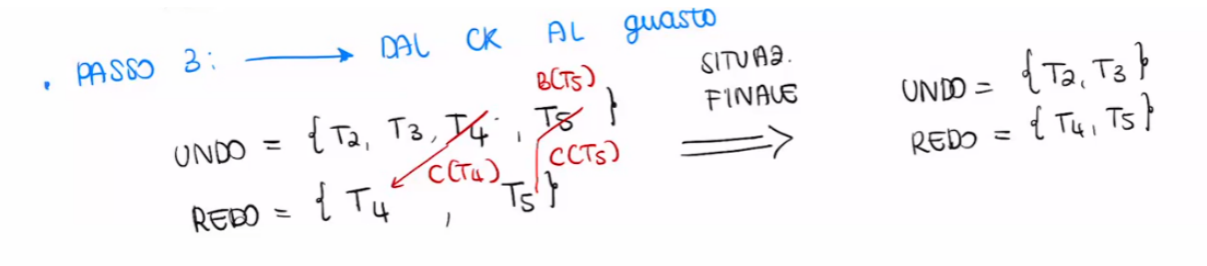



Figura 98: PASSO 3 : ANALISI IN AVANTI DEL LOG

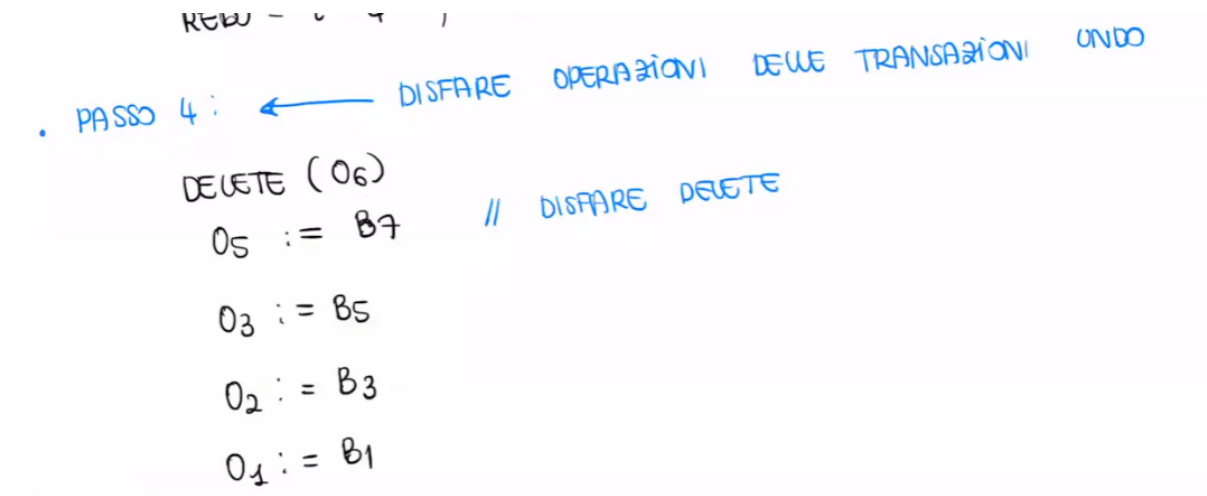



Figura 99: PASSO 4: FASE UNDO

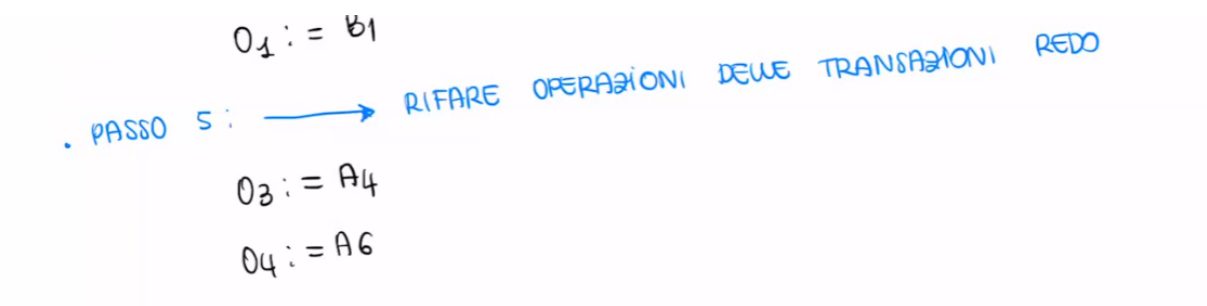



Figura 100: PASSO 5 : FASE DI REDO

### Esercizio 2

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

Figura 101: dati

PASSO 1: ← CK

PASSO 2: INIZIALIZZARE UNDO, REDO

UNDO = {T1, T4, T5, T6}

REDO = {}

Figura 102: PASSI 1 E 2

PASSO 3: →

UNDO = {T1, ~~T4~~, T5, T6, T7, T8}

REDO = {T4}

PASSO 4: ← UNDO

O3 := B7

O6 := B6

O5 := B5

O4 := B4

DELETE(O1)

Figura 103: PASSO 3 E 4

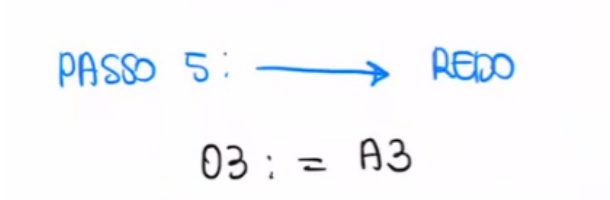



Figura 104: PASSO 5

## 12 Gestore metodi d'accesso

Il **gestore dei metodi di accesso**, per operare correttamente, deve conoscere una serie di informazioni fondamentali:

- l'**organizzazione delle tuple all'interno dei blocchi**;
- come sono **organizzati i blocchi al loro interno**.

In particolare, è necessario comprendere come è strutturata una **pagina** (o blocco) in memoria secondaria.

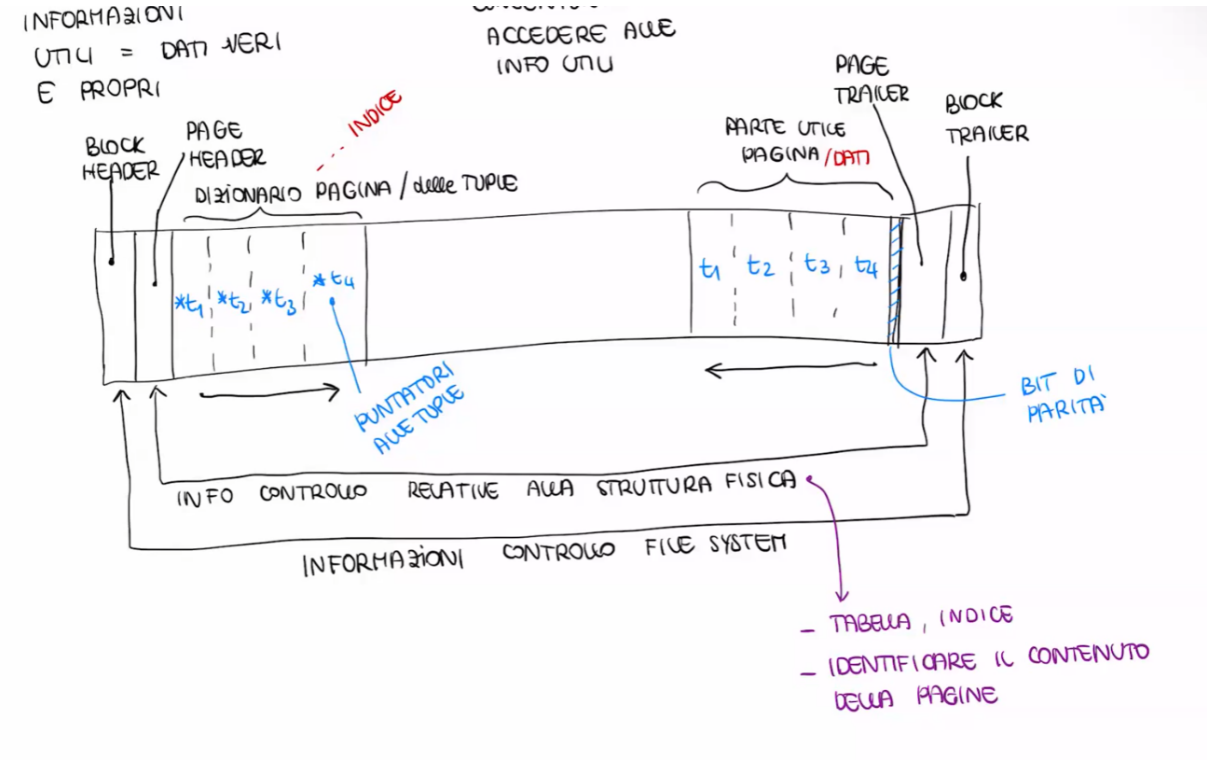



Figura 105: Struttura pagina

Una pagina contiene due tipologie principali di informazioni:

- **informazioni utili** (dati veri e propri), ovvero le tuple memorizzate;
- **informazioni di controllo**, che permettono di gestire e accedere correttamente ai dati.

Le informazioni di controllo sono fondamentali, poiché consentono al DBMS di localizzare, interpretare e manipolare i dati presenti nella pagina.

All'inizio e alla fine del blocco sono presenti strutture di controllo legate al file system:

- **Block Header**: contiene informazioni di gestione del blocco;
- **Block Trailer**: contiene informazioni di controllo aggiuntive (ad esempio per verifiche di integrità).

Sono inoltre presenti informazioni di controllo relative alla **struttura fisica**, che permettono di:

- identificare a quale tabella appartengono i dati;

- determinare se il blocco contiene dati o indici;
- interpretare correttamente il contenuto della pagina.

All'interno della pagina si distinguono due aree principali di memoria, che crescono in direzioni opposte:

- il **dizionario della pagina** (o slot directory), che contiene puntatori alle tuple;
- la **parte utile della pagina**, che contiene effettivamente le tuple (dati).

Il dizionario della pagina può essere visto come una sorta di indice interno, che consente di accedere rapidamente alle tuple presenti nella pagina.

Un elemento importante è il **bit di parità**, utilizzato per verificare l'integrità dei dati e rilevare eventuali corruzioni delle informazioni.

Il dizionario delle tuple è necessario soprattutto quando le tuple hanno **lunghezza variabile**.

- Nel caso di tuple a **lunghezza fissa**, non è necessario un dizionario, poiché la posizione delle tuple può essere calcolata direttamente.
- Tuttavia, questa soluzione è inefficiente, in quanto può causare **spreco di spazio**.
- Nel caso di tuple a **lunghezza variabile**, è invece necessario un dizionario, che memorizza per ogni tupla l'**offset di inizio** all'interno della pagina.

In questo modo, il DBMS può accedere direttamente alle tuple senza dover scansionare l'intero blocco, migliorando significativamente le prestazioni.

## 12.1 Operazioni sulle pagine

Il gestore dei metodi di accesso deve supportare diverse operazioni sulle pagine (blocchi), tra cui inserimento, cancellazione e accesso ai dati.

### 12.1.1 Inserimento

Nel caso di inserimento di una nuova tupla, si possono verificare tre situazioni:

- **Spazio contiguo sufficiente**
  È il caso più semplice: la tupla viene inserita nella parte utile della pagina e il **dizionario della pagina** viene aggiornato con il nuovo puntatore (offset) alla tupla.
- **Spazio insufficiente**
  Se non esiste spazio sufficiente all'interno della pagina, è necessario interagire con il file system per allocare un **nuovo blocco**, nel quale verrà inserita la tupla.
- **Spazio sufficiente ma non contiguo**
  In questo caso è necessario effettuare una **riorganizzazione della pagina** (compattazione), in modo da rendere lo spazio libero contiguo. Dopo la riorganizzazione, l'inserimento avviene come nel caso semplice.

### 12.1.2 Cancellazione

La cancellazione di una tupla è sempre possibile e generalmente non richiede una riorganizzazione immediata della pagina.

In particolare:

- viene rimosso il riferimento alla tupla dal dizionario;
- lo spazio occupato dalla tupla viene marcato come libero.

Eventuali operazioni di compattazione possono essere eseguite in un secondo momento per ottimizzare lo spazio.

### 12.1.3 Accesso

L'accesso ai dati può avvenire in due modalità principali:

- **Accesso a una tupla**
  Se è noto l'offset della tupla, si utilizza il **dizionario della pagina** per accedere direttamente alla posizione corretta. In caso di accesso a più tuple, può essere conveniente effettuare una **scansione sequenziale** della pagina.
- **Accesso a un attributo**
  Per accedere a un attributo specifico, è necessario prima individuare la tupla. Una volta identificata, si accede all'attributo tramite un meccanismo basato sugli **offset** interni alla tupla, che permettono di localizzare i singoli campi.

## 12.2 Criteri di organizzazione delle tuple nei blocchi

Le tuple all'interno dei blocchi possono essere organizzate secondo diversi criteri, in base a come vogliamo accedere ai dati.

Le principali modalità sono:

- **Organizzazione sequenziale**: le tuple sono memorizzate una dopo l'altra, tipicamente nell'ordine di inserimento.
- **Accesso calcolato (hash)**: la posizione delle tuple è determinata da una funzione di hash, che permette un accesso diretto.
- **Strutture ad albero**: i dati sono organizzati tramite indici (es. B+-tree), per rendere più efficienti le ricerche.

## 12.3 Organizzazione sequenziale dei file

Consideriamo ora il caso più semplice: l'**organizzazione sequenziale**.

I dati sono salvati in file composti da blocchi logicamente consecutivi. Possiamo avere due varianti:

- **Sequenziale disordinata (seriale)** → le tuple sono nell'ordine di inserimento
- **Sequenziale ordinata** → le tuple sono ordinate secondo una chiave

### 12.3.1 Sequenziale disordinata (seriale)

È la struttura più semplice e più intuitiva.

**Inserimento** L'inserimento è molto veloce:

- si prende l'ultimo blocco del file;
- si inserisce la tupla lì dentro (se c'è spazio);

- si fa in genere un solo accesso al disco.

**Ricerca** La ricerca è il punto debole:

- bisogna scorrere tutti i blocchi;
- quindi serve una scansione sequenziale;
- nel caso peggiore: $N$ accessi alla memoria secondaria.

**Cancellazione**

- prima bisogna trovare la tupla;
- poi si elimina;
- non si compatta subito lo spazio.

**Modifica**

- si trova la tupla;
- se cambia dimensione (tuple variabili), potrebbe non starci più;
- quindi si fa: cancellazione + inserimento.

**Problema principale** Queste operazioni (soprattutto cancellazioni e modifiche) creano **frammentazione** e spreco di spazio.

Per questo motivo il DBMS può eseguire delle **riorganizzazioni periodiche**, che servono a:

- compattare i dati;
- recuperare spazio;
- migliorare le prestazioni.

### 12.3.2 Sequenziale ordinata

La **struttura sequenziale ordinata** è un file sequenziale in cui le tuple sono memorizzate in ordine rispetto a una **chiave di ordinamento**.

In generale, questa chiave può essere diversa dalla **chiave primaria** ed è scelta per ottimizzare determinate operazioni di accesso.

**Caratteristiche principali**

- le tuple sono mantenute **ordinate** secondo la chiave;
- i blocchi sono logicamente consecutivi;
- è possibile sfruttare l'ordinamento per rendere più efficienti le ricerche.

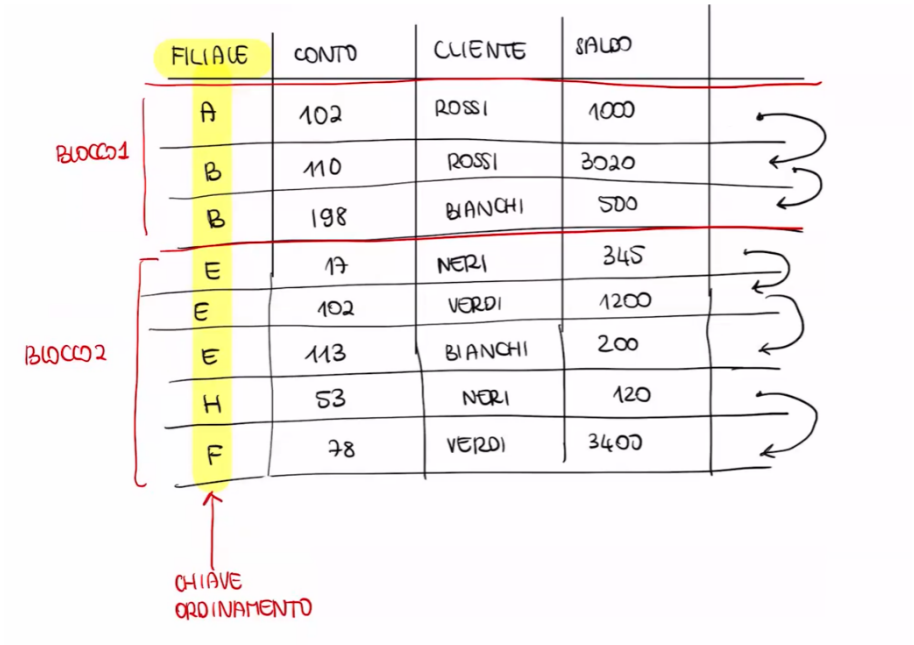



Figura 106: Esempio

- **Vantaggio**
  Rende efficienti tutte le operazioni che possono sfruttare l'**ordinamento**, come ad esempio le ricerche per intervallo (es. ricerca per filiale).
- **Svantaggio**
  Le operazioni di aggiornamento (inserimento e modifica) devono mantenere l'ordinamento, con un conseguente aumento del costo computazionale.

**Operazioni principali** **1. Inserimento**

- si individua il blocco $b$ in cui effettuare l'inserimento, cioè il blocco che contiene la tupla precedente nell'ordinamento;
- una volta individuato il blocco $b$, si verifica se è presente spazio sufficiente:
  - **se c'è spazio** → inserimento semplice, mantenendo l'ordinamento;
  - **se non c'è spazio** → si crea un **nuovo blocco** in cui inserire la tupla (eventualmente aggiornando i collegamenti tra i blocchi).

**2. Cancellazione**

- si individua il blocco $b$ che contiene la tupla;
- si elimina la tupla;
- si aggiornano eventuali **puntatori** o strutture di supporto.

Le operazioni di cancellazione possono portare a una **frammentazione dello spazio**, rendendo necessario effettuare operazioni di riorganizzazione.

**3. Riorganizzazione**

Per mantenere efficiente l'utilizzo dello spazio, il DBMS può eseguire periodicamente operazioni di **riorganizzazione** del file.

Queste operazioni si basano su **coefficienti di riempimento** (*fill factor*), scelti dal DBMS in base a statistiche sul carico di lavoro.

In particolare:

- il DBMS non riempie completamente i blocchi, ma lascia una certa quantità di spazio libero;
- questo spazio serve per facilitare inserimenti futuri senza dover creare nuovi blocchi o spostare troppe tuple;
- i coefficienti di riempimento sono determinati analizzando la frequenza di inserimenti, cancellazioni e modifiche.

In questo modo si ottiene un compromesso tra:

- **utilizzo dello spazio**;
- **efficienza delle operazioni di aggiornamento**.

## 12.4 Indici

Gli **indici** sono strutture dati utilizzate per rendere più efficiente l'accesso ai dati nei file sequenziali.

L'obiettivo principale è quello di **velocizzare l'accesso casuale** ai dati, evitando la scansione sequenziale dell'intero file.

L'accesso avviene tramite una **chiave di ricerca**, che può essere costituita da uno o più attributi della relazione.

In particolare:

- la chiave di ricerca è l'insieme di attributi utilizzato per effettuare le operazioni di ricerca;
- non coincide necessariamente con la chiave primaria;
- può essere scelta in base alle query più frequenti.

Gli indici si distinguono principalmente in due categorie:

- **Indici primari**
  La chiave di ricerca coincide con la **chiave di ordinamento** del file.
- **Indici secondari**
  La chiave di ricerca è **diversa** dalla chiave di ordinamento del file.

La scelta tra indice primario e secondario dipende dal tipo di accesso ai dati e dalle operazioni più frequenti sul database.

### 12.4.1 Struttura degli indici primari

Nel caso degli **indici primari**, poiché la chiave di ricerca coincide con la chiave di ordinamento, le tuple con lo stesso valore di chiave sono memorizzate in modo contiguo.

Ogni record dell'indice è una coppia:

$$\langle V_i, P_i \rangle$$

dove:

- $V_i$ è il valore della chiave di ricerca;

- $P_i$ è un puntatore alla **prima tupla** che contiene quel valore.

È sufficiente un solo puntatore, poiché le tuple con lo stesso valore sono consecutive nel file.

### 12.4.2 Indici primari densi e sparsi

Gli indici primari possono essere di due tipi:

**Indice primario denso**

- contiene un record di indice per **ogni valore** della chiave di ricerca;
- permette un accesso molto preciso;
- occupa più spazio.

**Indice primario sparso**

- contiene un record di indice per **ogni blocco** del file;
- ogni voce punta alla prima tupla del blocco;
- occupa meno spazio, ma richiede una scansione all'interno del blocco.

**Esempio** Consideriamo la tabella mostrata in figura 106.

- **Indice primario denso**
  L'indice contiene una coppia per ogni valore della chiave di ricerca:

  $$A,\ B,\ E,\ H,\ M$$

  Ogni valore è associato a un puntatore alla prima tupla corrispondente.
- **Indice primario sparso**
  L'indice contiene una coppia per ogni blocco.

  Nell'esempio, se i dati sono distribuiti in due blocchi:

  - una voce per il primo blocco (es. $A$);
  - una voce per il secondo blocco (es. $E$).

  Ogni voce punta alla prima tupla del blocco.

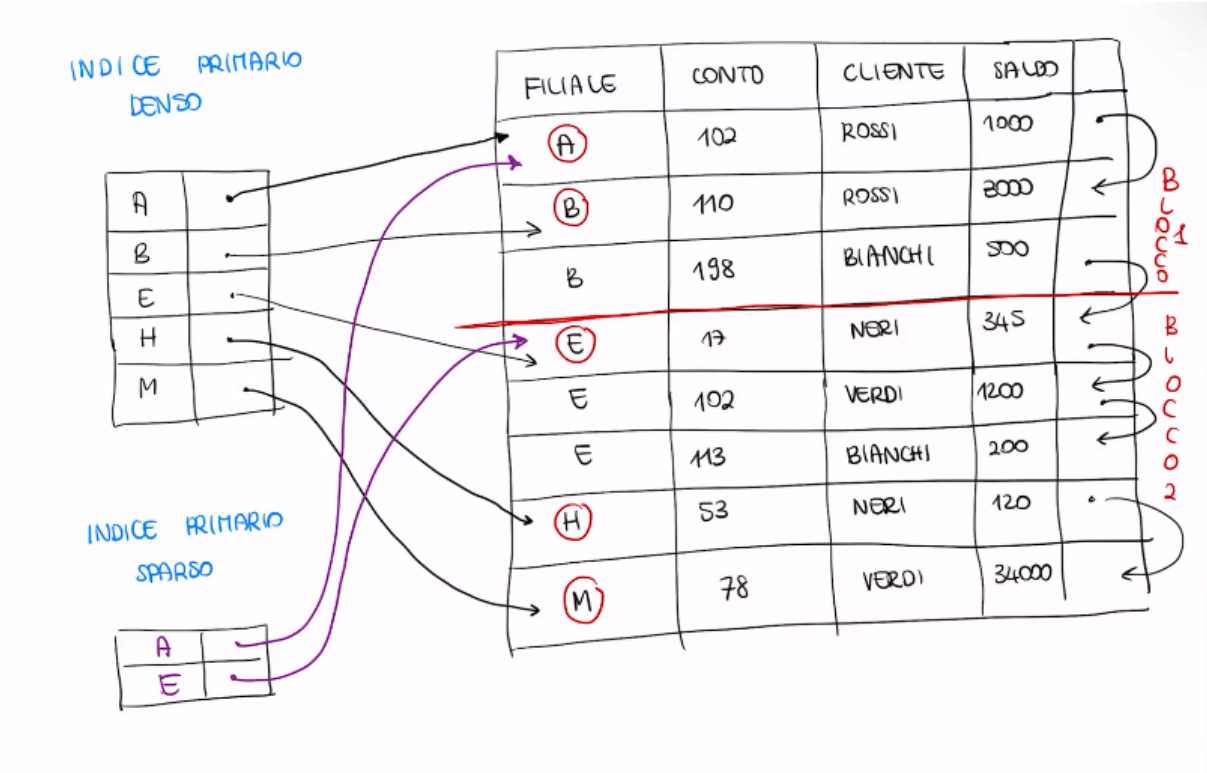



Figura 107: Esempio indice primario denso e indice primario sparso

### 12.4.3 Operazioni sugli indici primari

**Ricerca Indice primario denso**

Data una chiave $K$:

- si cerca $K$ nell'indice;
- **se $K$ non è presente**:
  - la ricerca termina;
  - costo: 1 accesso all'indice;
- **se $K$ è presente**:
  - si trova la coppia $\langle K, P_K \rangle$;
  - si segue il puntatore $P_K$;
  - si accede al blocco dati;
  - costo: 1 accesso all'indice + 1 accesso al blocco dati.

**Indice primario sparso**

Data una chiave $K$:

- $K$ potrebbe non essere presente nell'indice anche se presente nei dati;
- si effettua una scansione dell'indice per trovare la coppia $\langle K', P_{K'} \rangle$ tale che:

$$K' \leq K \text{ ed è il valore più grande possibile}$$

- si segue il puntatore $P_{K'}$ per accedere al blocco dati;
- all'interno del blocco si cerca la tupla (eventualmente scansione sequenziale).
- costo: 1 accesso all'indice + 1 accesso al blocco dati.

**Inserimento** Nel caso di inserimento di una nuova tupla, si procede innanzitutto come nel caso del file sequenziale (ordinato o disordinato).

Successivamente, è necessario **aggiornare anche l'indice**. Si distinguono due casi:

- **Indice denso**
  Si inserisce una nuova coppia nell'indice solo se la chiave di ricerca della tupla inserita è **nuova** (cioè non già presente nell'indice).
- **Indice sparso**
  Si inserisce una nuova coppia nell'indice solo se l'inserimento della tupla comporta la creazione di un **nuovo blocco**.

La presenza di un indice introduce quindi un **costo aggiuntivo** nelle operazioni di aggiornamento.

Per questo motivo, nella pratica non è conveniente creare un indice per ogni attributo della tabella, ma è necessario trovare un **compromesso** tra:

- numero di indici mantenuti;
- efficienza delle operazioni di accesso.

**Cancellazione** Nel caso di cancellazione di una tupla:

- si elimina la tupla dal file sequenziale (ordinato o disordinato);
- si aggiorna l'indice.

Anche qui si distinguono due casi:

- **Indice denso**
  Si elimina la coppia dall'indice solo se la tupla cancellata era l'**ultima** con quella chiave $K$.
- **Indice sparso**
  Si elimina la coppia dall'indice se la tupla cancellata era la **prima del blocco** (cioè quella puntata dall'indice).

  In tal caso:

  - si deve aggiornare l'indice inserendo una nuova coppia;
  - la nuova coppia punterà alla nuova prima tupla del blocco.

### 12.4.4 Indici secondari

Gli **indici secondari** sono indici in cui la **chiave di ricerca è diversa dalla chiave di ordinamento** del file.

In questo caso, le tuple con lo stesso valore di chiave di ricerca **non sono memorizzate in modo contiguo** nel file sequenziale.

Per questo motivo, la struttura dell'indice è diversa rispetto agli indici primari.

Ogni record dell'indice è una coppia:

$$\langle V_i, P_i \rangle$$

dove:

- $V_i$ è un valore della chiave di ricerca;
- $P_i$ è un puntatore a un **bucket** contenente più puntatori.

Il bucket contiene i puntatori a tutte le tuple del file sequenziale che hanno valore di chiave pari a $V_i$.

**Esempio** Consideriamo la tabella mostrata in figura 106.

Se utilizziamo come chiave di ricerca l'attributo *cliente*, otteniamo valori come:

Rossi, Bianchi, Neri, Verdi

Per ciascun valore:

- **Rossi** → il puntatore $P_i$ punta a un bucket contenente due puntatori, uno per ciascuna tupla con cliente = Rossi;
- **Bianchi** → il bucket contiene i puntatori a tutte le tuple con cliente = Bianchi;
- analogamente per **Neri** e **Verdi**.

La struttura è illustrata nella figura seguente.

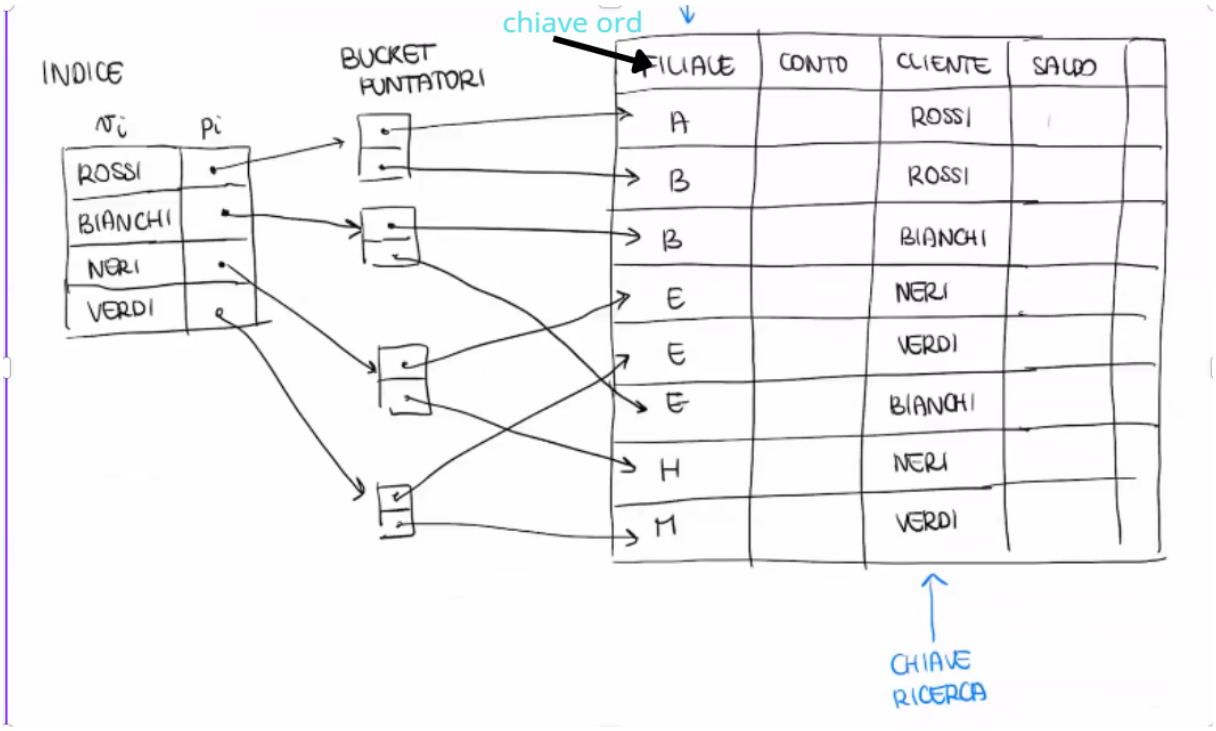



Figura 108: Esempio di indice secondario con bucket di puntatori

### 12.4.5 Operazioni sugli indici secondari

**Ricerca** Data una chiave di ricerca $K$:

- si effettua una **ricerca sequenziale sull'indice** per trovare $K$;

- poiché l'indice secondario è **denso**, se $K$ non è presente nell'indice allora non esistono tuple con chiave $K$;
- se $K$ è presente:
  - si trova la coppia $\langle K, P_K \rangle$;
  - si accede al **bucket** $B$ puntato da $P_K$;
  - il bucket contiene puntatori a tutte le tuple con chiave $K$;
  - si accede al file dati seguendo i puntatori contenuti nel bucket.

Costo:

$$1 \text{ accesso all'indice} + 1 \text{ accesso al bucket} + n \text{ accessi ai blocchi dati}$$

dove $n$ è il numero di puntatori contenuti nel bucket $B$.

**Inserimento, cancellazione e modifica** Le operazioni di inserimento, cancellazione e modifica avvengono in modo analogo al caso degli **indici primari densi**, con la differenza che è necessario aggiornare anche il **bucket**.

- **Inserimento**:
  - si inserisce la tupla nel file dati;
  - si aggiorna l'indice:
    - se la chiave $K$ è nuova $\rightarrow$ si crea una nuova coppia nell'indice con un nuovo bucket;
    - altrimenti $\rightarrow$ si aggiunge un puntatore nel bucket esistente.
- **Cancellazione**:
  - si elimina la tupla dal file dati;
  - si rimuove il puntatore dal bucket;
  - se il bucket diventa vuoto $\rightarrow$ si elimina anche la coppia dall'indice.
- **Modifica**:
  - se cambia la chiave di ricerca $\rightarrow$ si esegue una cancellazione seguita da un inserimento;
  - altrimenti $\rightarrow$ si aggiorna solo la tupla nel file dati.

## 12.5 Strutture di accesso avanzate

Per migliorare le prestazioni di accesso agli indici, è possibile definire strutture dati più efficienti rispetto agli indici sequenziali. Le principali sono:

- **Strutture ad albero**: indici $B^+$-tree;
- **Strutture ad accesso calcolato**: strutture hash.

L'obiettivo comune è **minimizzare gli accessi alla memoria secondaria** necessari per accedere all'indice.

## 12.6 $B^+$-tree

Il **$B^+$-tree** è una struttura ad albero con le seguenti caratteristiche:

- ogni nodo viene memorizzato in una **pagina** della memoria secondaria;
- i legami tra i nodi sono **puntatori a pagina**, ovvero riferimenti a blocchi in memoria secondaria;
- è caratterizzato da **pochi livelli** e **molti nodi foglia**: non si tratta di un albero binario, poiché la complessità delle operazioni dipende dall'altezza dell'albero, ed è quindi preferibile avere un albero largo piuttosto che profondo;
- è un **albero bilanciato**: non esistono percorsi preferenziali, il che garantisce che inserimenti e cancellazioni non alterino le prestazioni di accesso ai dati.

### 12.6.1 Struttura dei nodi

Distinguiamo due tipologie di nodi: i **nodi foglia** e i **nodi intermedi**. Introduciamo il parametro **fan-out**, indicato con $n$.

**Nodo foglia** Un nodo foglia con fan-out $n$:

- può contenere fino a $n-1$ valori ordinati di chiavi di ricerca e fino a $n$ puntatori ai dati;

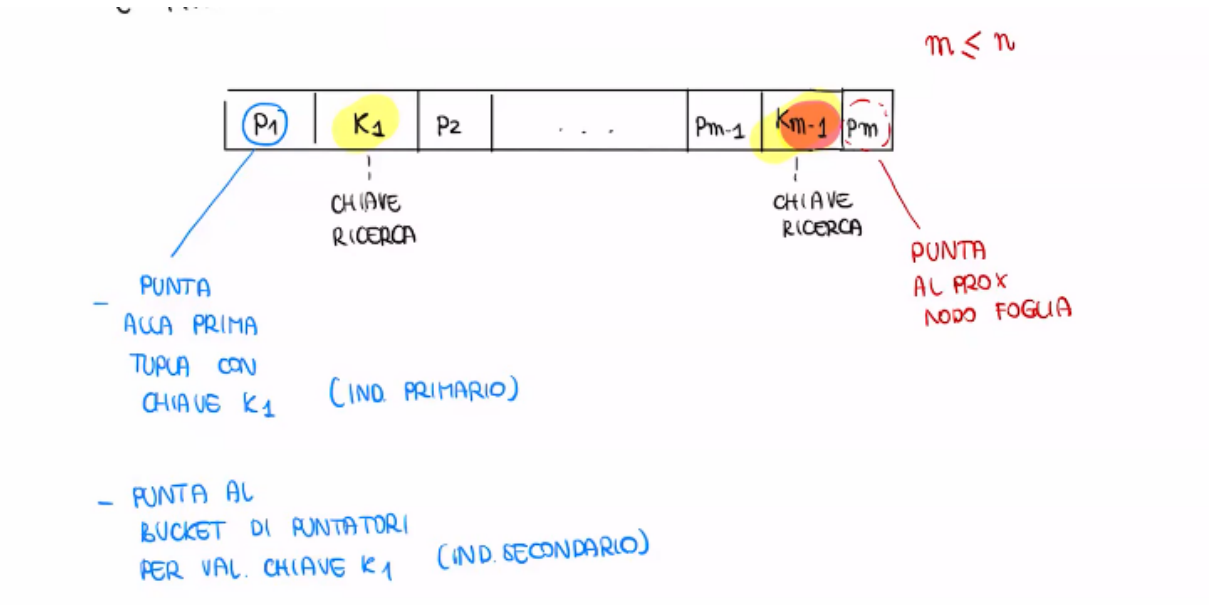



Figura 109: Nodo Foglia

I nodi foglia rispettano un **vincolo di ordinamento**:

- le chiavi **all'interno** di ogni nodo foglia sono ordinate;
- i nodi foglia sono **ordinati tra loro**.

Formalmente, dati due nodi foglia $L_i$ e $L_j$ con $i < j$, per ogni chiave $K_t \in L_i$ e per ogni chiave $K_s \in L_j$:

$$K_t < K_s$$

**Nodo intermedio** Un nodo intermedio con fan-out $n$ ha una struttura simile al nodo foglia, ma con un significato diverso:

$$P_1 \mid K_1 \mid P_2 \mid \cdots \mid P_{m-1} \mid K_{m-1} \mid P_m$$

I puntatori $P_i$ non puntano a dati, ma a **sottoalberi**, con il seguente significato:

- $P_1$ punta al sottoalbero contenente le tuple con chiave $K < K_1$;
- $P_i$ (per $1 < i < m$) punta al sottoalbero contenente le tuple con chiave $K_{i-1} \leq K < K_i$;
- $P_m$ punta al sottoalbero contenente le tuple con chiave $K \geq K_{m-1}$.

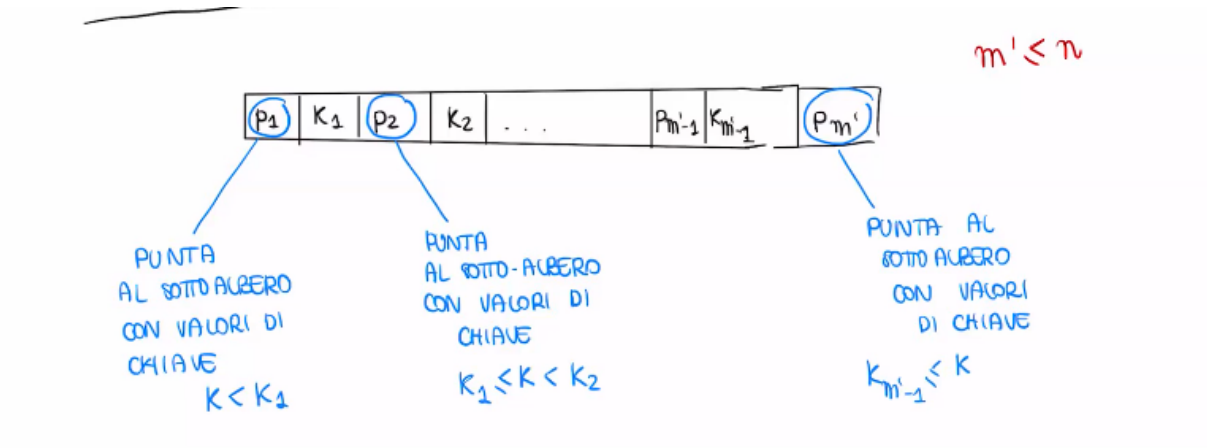



Figura 110: Nodo intermedio

**Vincoli di riempimento** I nodi del B$^+$-tree devono rispettare dei **vincoli di riempimento**, che garantiscono che l'albero rimanga bilanciato e che lo spazio non venga sprecato.

- **Nodo foglia** (fan-out $n$): il numero di valori chiave #chiave è vincolato come segue:

$$\left\lceil \frac{n-1}{2} \right\rceil \leq \#\text{chiave} \leq n-1$$

- **Nodo intermedio** (fan-out $n$): il numero di puntatori #puntatori è vincolato come segue:

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

### 12.6.2 Esempio B$^+$-tree

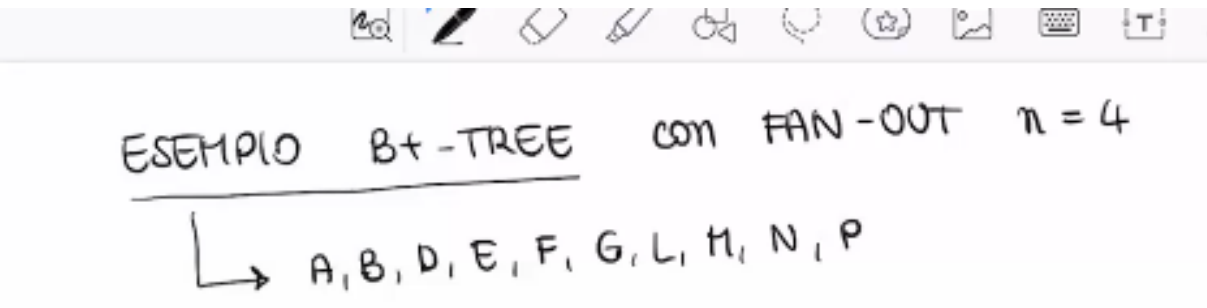



Figura 111: Esempio 1 - Contenuto

Con fan-out $n = 4$, i vincoli di riempimento diventano:

- **Nodo foglia**:
$$\left\lceil \frac{4-1}{2} \right\rceil \leq \#\text{chiave} \leq 4-1 \quad \Rightarrow \quad 2 \leq \#\text{chiave} \leq 3$$

- **Nodo intermedio**:
$$\left\lceil \frac{4}{2} \right\rceil \leq \#\text{puntatori} \leq 4 \quad \Rightarrow \quad 2 \leq \#\text{puntatori} \leq 4$$

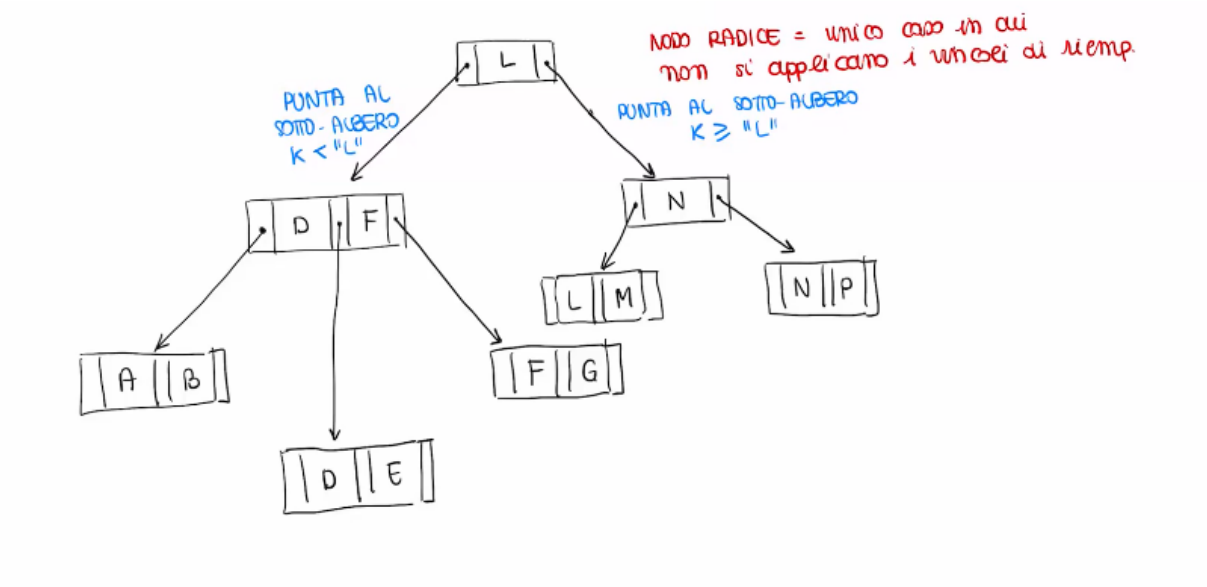



Figura 112: Esempio 2 : Creazione Albero

Il nodo radice è l'unico nodo per cui **non si applicano i vincoli di riempimento**.

Valgono tuttavia le stesse regole dei nodi intermedi per quanto riguarda il significato dei puntatori: data una chiave $L$ nella radice:

- il puntatore sinistro punta al sottoalbero contenente le tuple con chiave $K < L$;
- il puntatore destro punta al sottoalbero contenente le tuple con chiave $K \geq L$.

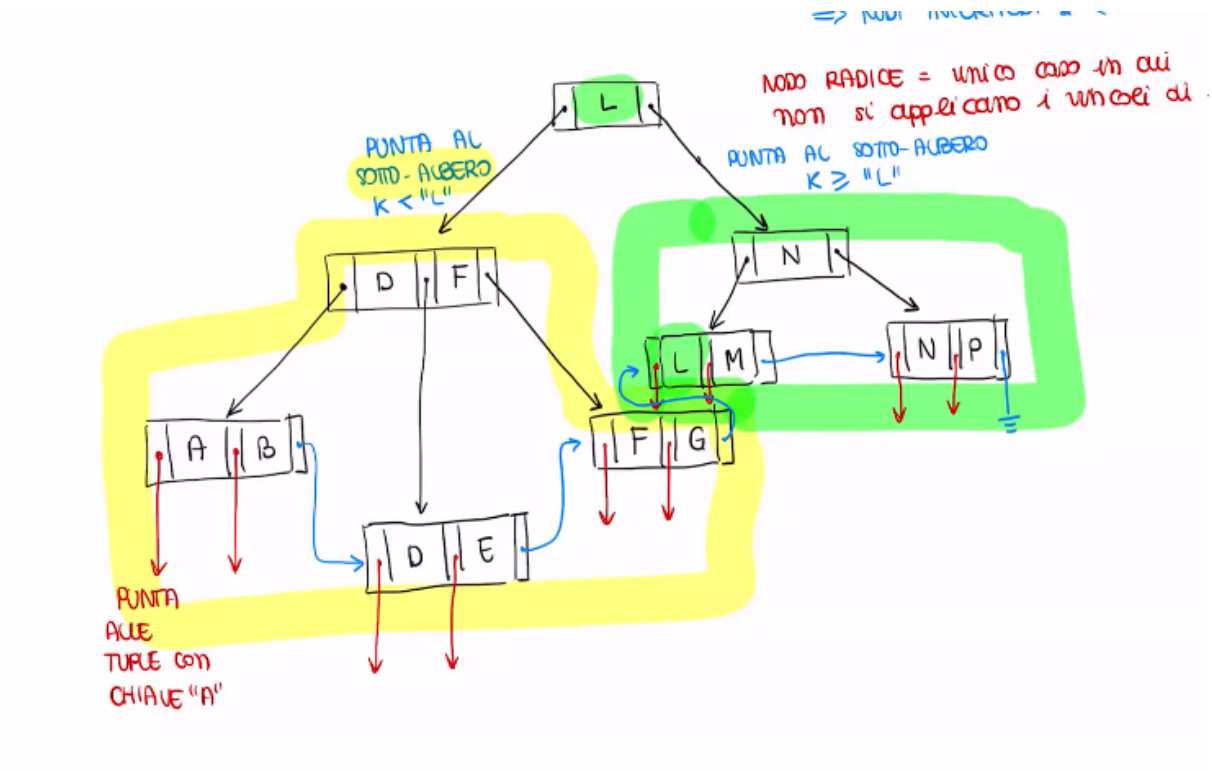



Figura 113: Esempio 3 : Albero con Riempimento minimo

I nodi foglia contengono le chiavi ordinate e i relativi puntatori ai dati: ad esempio, il nodo foglia contenente le chiavi A e B B ha i puntatori rossi che puntano alle tuple corrispondenti nel file dati. L'ultimo puntatore di ciascun nodo foglia (in blu) punta al nodo foglia successivo, realizzando una **lista concatenata** tra tutti i nodi foglia. Questo esempio rispetta il vincolo di riempimento minimo.

## Esempio con riempimento massimo

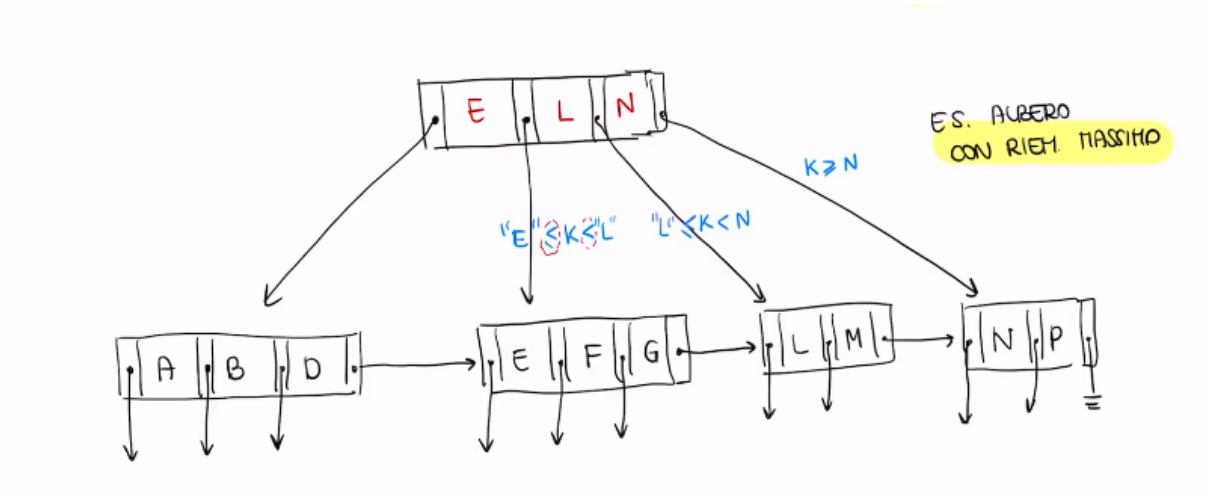



Figura 114: Esempio 1 : Albero con Riempimento massimo

È possibile costruire un B$^+$-tree alternativo sugli stessi dati utilizzando un **riempimento massimo**. In questo caso i nodi foglia sono:

$$[A,\ B,\ D] \quad [E,\ F,\ G] \quad [L,\ M] \quad [N,\ P]$$

Tutti i nodi foglia rispettano i vincoli di riempimento. In questo caso non è necessario introdurre un livello intermedio: è sufficiente un **nodo radice** che punti direttamente ai quattro nodi foglia tramite 4 puntatori, il che è compatibile con il vincolo di riempimento dei nodi intermedi (#puntatori $\leq 4$).

### 12.6.3 Ricerca con chiave K

La ricerca di un valore di chiave K nel B$^+$-tree avviene scendendo dall'albero verso le foglie, seguendo i puntatori appropriati, fino a raggiungere il nodo foglia che contiene il valore cercato. L'algoritmo è il seguente:

1. **Cercare nel nodo corrente** il più piccolo valore di chiave maggiore di K:

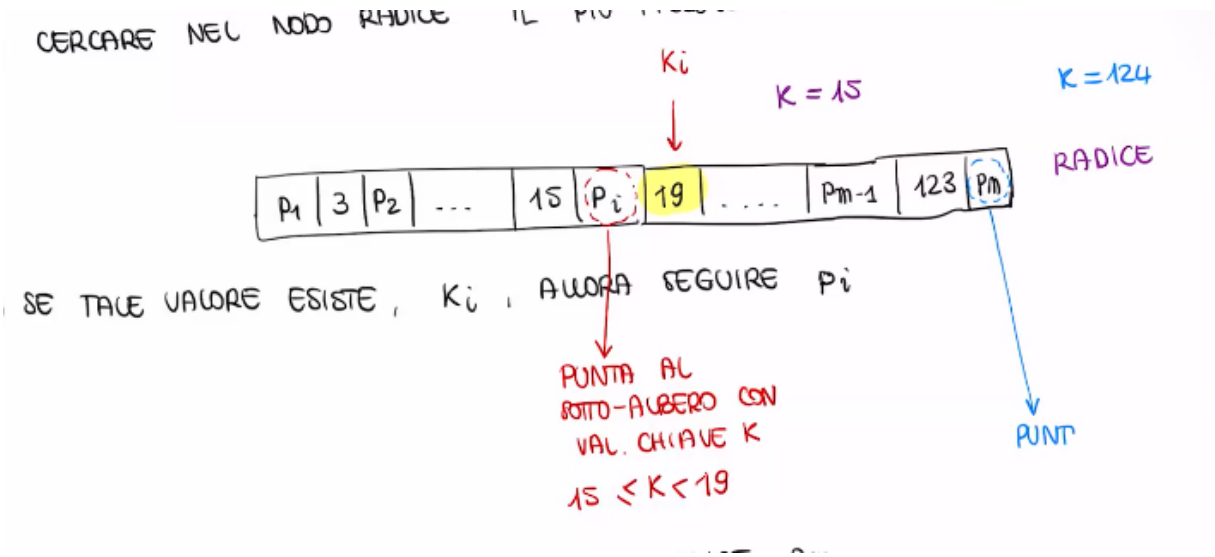



Figura 115: Ricerca nel nodo: $K = 15$, il più piccolo valore di chiave maggiore di $K$ è $K_i = 19$

2. **Se tale valore $K_i$ esiste**, seguire il puntatore $P_i$, il quale punta al sottoalbero contenente i valori di chiave $K$ tali che:
$$K_{i-1} \leq K < K_i$$
Ad esempio, con $K = 15$ e $K_i = 19$: $15 \leq K < 19$.

   **Se tale valore non esiste**, seguire $P_m$, che punta al sottoalbero con valori di chiave $K \geq K_{m-1}$.

3. **Se si raggiunge un nodo foglia**, cercare il valore $K$ al suo interno e seguire il corrispondente puntatore $P_i$:

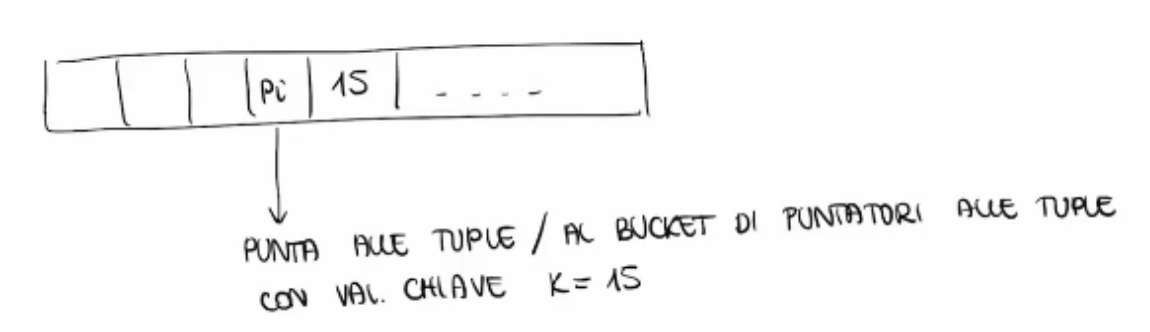



Figura 116: Nodo foglia: $P_i$ punta alle tuple (o al bucket di puntatori alle tuple) con valore di chiave $K = 15$

**Se non si è ancora raggiunto un nodo foglia**, ripetere dal passo 1 e 2.

### Costo della Ricerca

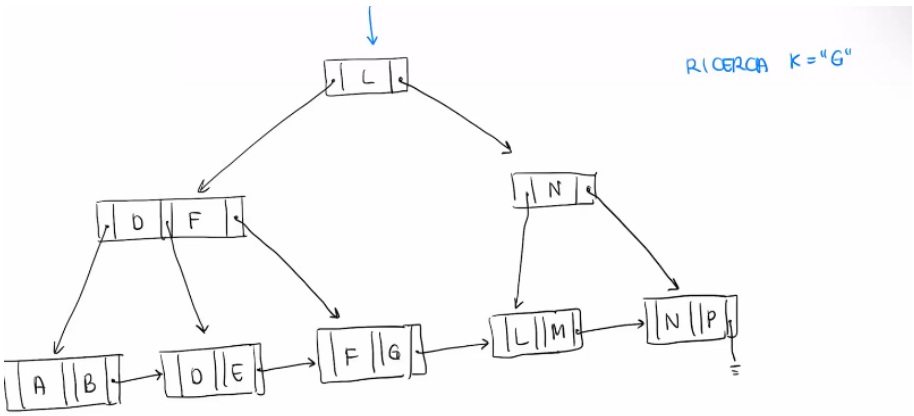



Figura 117: Esempio ricerca per K = 'G'

Consideriamo la ricerca di K=G nell'albero dell'esempio precedente.

1. Nel nodo radice, il più piccolo valore di chiave maggiore di G è L: si segue quindi il puntatore al sottoalbero sinistro, contenente i valori con chiave K < L.
2. Si raggiunge un nodo intermedio: poiché non si è ancora a una foglia, si ripete il passo 1.
3. Si raggiunge il nodo foglia: il valore $G$ è presente; si segue il corrispondente puntatore verso le tuple con valore di chiave $K = G$.

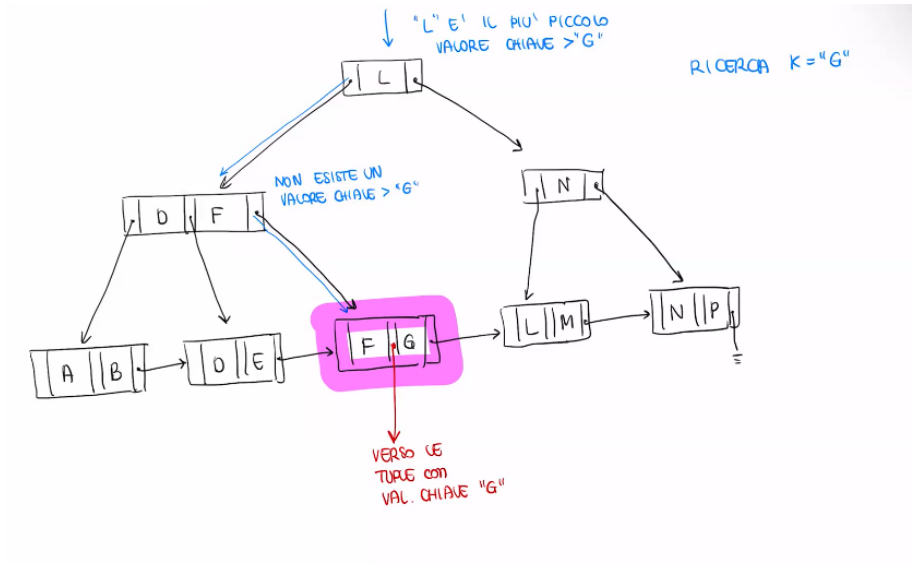



Figura 118: Ricerca su chiave G

**Il Costo della Ricerca :** è misurato in termini di **accessi alla memoria secondaria**. Poiché ogni nodo occupa un blocco, il costo è pari alla **profondità dell'albero**.

La profondità dipende dal numero di nodi foglia $NF$, che a sua volta dipende dal fan-out $n$ e dal numero di valori chiave da memorizzare.

Nel **caso pessimo** (riempimento minimo), ogni foglia contiene $\left\lceil \frac{n-1}{2} \right\rceil$ valori chiave, quindi il numero massimo di foglie è:

$$NF_{max} = \frac{\#\text{valori chiave}}{\left\lceil \frac{n-1}{2} \right\rceil}$$

La profondità dell'albero è quindi:

$$\text{profondità} \leq 1 + \log_{\lceil n/2 \rceil} NF_{max}$$

dove il termine 1 tiene conto della radice.

### 12.6.4 Inserimento

**Passo 1.** Ricerca del nodo foglia $NF$ in cui il valore $K$ deve essere inserito.

**Passo 2.** Se $K$ è già presente in $NF$:

- **Indice primario**: nessuna azione;
- **Indice secondario**: aggiornare il bucket dei puntatori.

Altrimenti, inserire $K$ in $NF$ rispettando l'ordinamento:

- **Indice primario**: inserire $K$ e il puntatore alla prima tupla;
- **Indice secondario**: inserire $K$ e creare un nuovo bucket che sarà puntato dall'indice.

**Passo 3 (Split).** Se in $NF$ esistono già $n-1$ valori di chiave (violazione del vincolo di riempimento massimo):

- creare due nodi foglia $NF_1$ e $NF_2$;
- inserire i primi $\left\lceil \frac{n-1}{2} \right\rceil$ valori di chiave in $NF_1$;
- inserire i rimanenti valori in $NF_2$;
- aggiornare il nodo padre; se anche il padre viola il vincolo di riempimento massimo, propagare lo split verso l'alto, creando un nuovo livello (e quindi una nuova radice) se necessario.

**Esempio di inserimento con split** Consideriamo un B$^+$-tree con fan-out $n = 5$ e supponiamo di voler inserire $K = 15$.

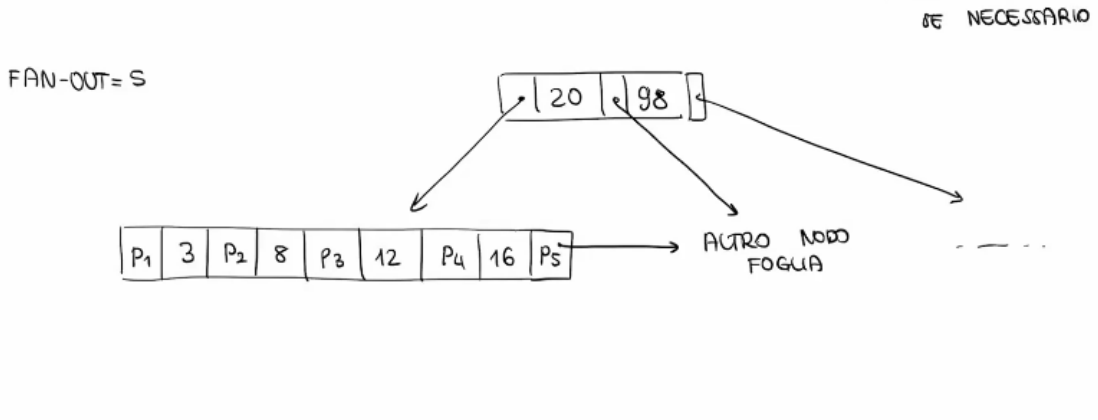



Figura 119: Stato iniziale: fan-out $n = 5$, nodo foglia contenente i valori $3, 8, 12, 16$

Il valore 15 dovrebbe essere inserito tra 12 e 16, ma il nodo foglia conterrebbe 5 valori di chiave, violando il vincolo di riempimento massimo ($\leq n - 1 = 4$). È necessario effettuare uno **split**.

Si creano due nuovi nodi foglia $NF_1$ e $NF_2$:

$$NF_1 = \left\lceil \frac{n-1}{2} \right\rceil = 2 \text{ valori di chiave}$$

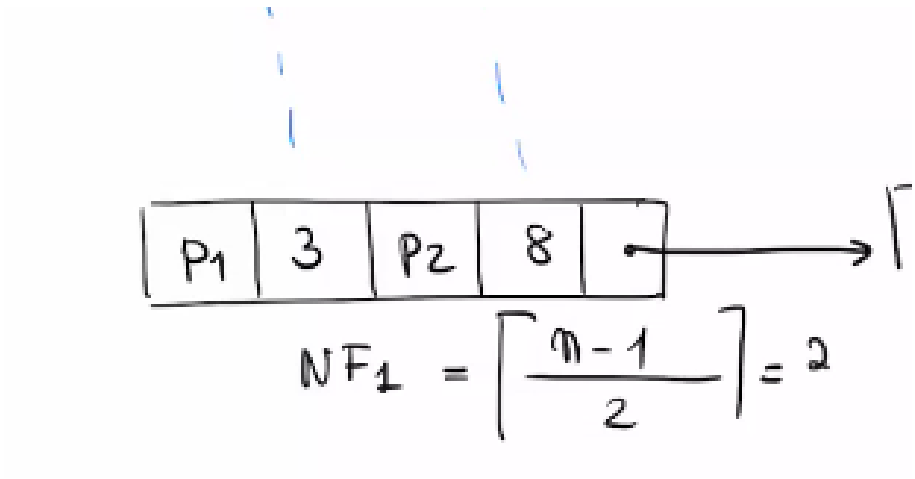



Figura 120: Split: $NF_1$ contiene i valori $3, 8$; $NF_2$ contiene i valori $12, 15, 16$

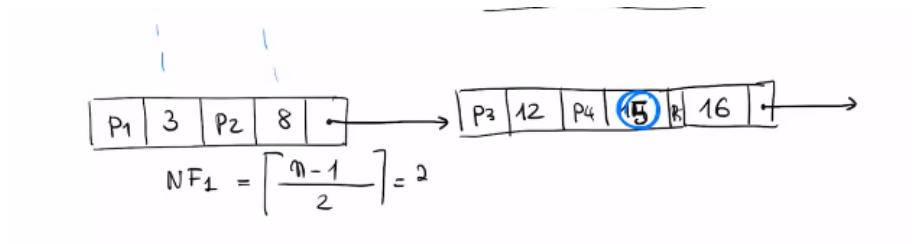



Figura 121: I due nodi foglia sono collegati tramite il puntatore al nodo foglia successivo

Il nodo padre viene aggiornato: viene inserita la chiave 12 nel padre, con il puntatore sinistro che punta al sottoalbero con valori $K < 12$ e il puntatore destro che punta al sottoalbero con valori $12 \leq K < 20$.

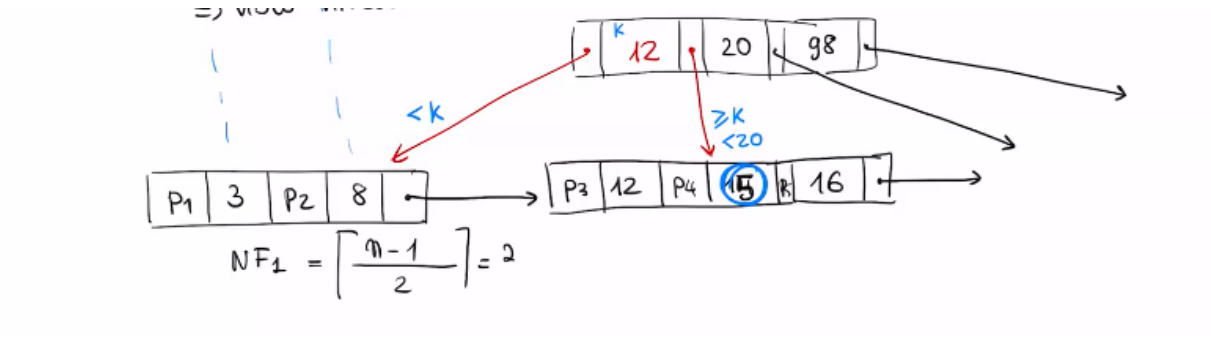



Figura 122: Aggiornamento del padre dopo lo split: inserimento della chiave 12

### 12.6.5 Cancellazione

**Passo 1.** Ricercare il nodo foglia $NF$ in cui si trova il valore di chiave $K$.

**Passo 2.** Cancellare $K$ e il suo puntatore da $NF$.

Se dopo la cancellazione si verifica:

$$\#\text{chiave} < \left\lceil \frac{n-1}{2} \right\rceil$$

si viola il vincolo di riempimento minimo. In tal caso è necessario eseguire un **merge** di $NF$ con un nodo fratello:

- si individua il nodo fratello candidato;
- si genera un unico nodo foglia; se non esiste un fratello che, unito a $NF$, rispetta il vincolo di riempimento massimo, si esegue prima il merge tra i due e poi uno split;
- si aggiorna il nodo padre;
- se anche il padre viola il vincolo di riempimento minimo, si propaga il merge verso l'alto.

**Esempio di cancellazione con merge** Consideriamo il B$^+$-tree risultante dall'esempio precedente e supponiamo di voler eliminare $K = 3$.

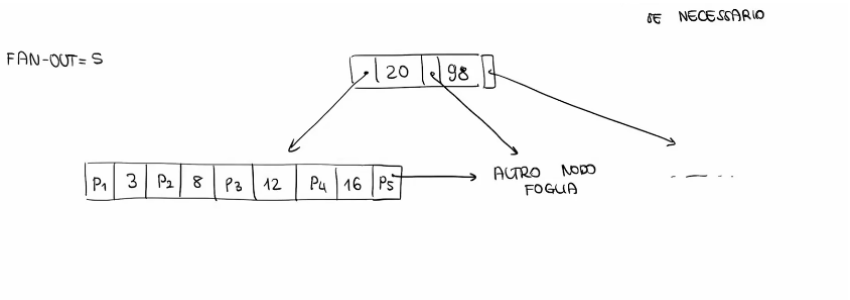



Figura 123: Stato iniziale: fan-out $n = 5$, nodo foglia contenente i valori $3, 8, 12, 16$

Dopo la cancellazione, $NF_1$ contiene un solo valore di chiave, violando il vincolo di riempimento minimo (con fan-out $n = 5$, si richiede #chiave $\geq 2$).

Si esegue quindi un **merge** di $NF_1$ con il fratello $NF_2$, ottenendo un unico nodo foglia contenente i valori $8, 12, 15, 16$:

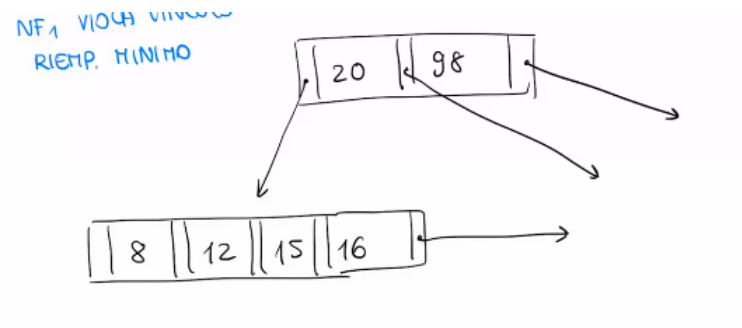



Figura 124: Risultato del merge: unico nodo foglia con valori $8, 12, 15, 16$; il padre viene aggiornato perdendo il puntatore a $NF_1$

### Esercizio B$^+$-tree

**Testo** Costruire un B$^+$-tree con fan-out $n = 5$ a partire dai seguenti nodi foglia:

$$(A, B, C, D) \quad (F, G, M, N) \quad (O, P) \quad (S, T) \quad (W, Z)$$

Successivamente, inserire il valore $H$. Infine, eliminare il valore $Z$

**Soluzione** **Vincoli di riempimento** con $n = 5$:

- Nodo foglia (NF): $\left\lceil \frac{n-1}{2} \right\rceil \leq \#\text{chiave} \leq n - 1 \;\Rightarrow\; 2 \leq \#\text{chiave} \leq 4$
- Nodo intermedio (NI): $\lceil \frac{n}{2} \rceil \leq \#\text{puntatori} \leq n \;\Rightarrow\; 3 \leq \#\text{puntatori} \leq 5$

**Passo 1: costruzione dell'albero iniziale.**

I nodi foglia sono collegati in lista concatenata:

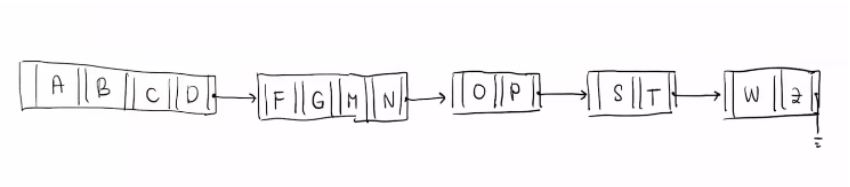



Figura 125: Nodi foglia collegati in lista concatenata

Abbiamo 5 nodi foglia, quindi servono 5 puntatori al livello superiore. Poiché $5 \leq n = 5$, è sufficiente un unico nodo radice con 4 chiavi e 5 puntatori.

I valori di chiave nella radice sono:

$$K_1 = F, \quad K_2 = O, \quad K_3 = S, \quad K_4 = W$$

Ogni $K_i$ rappresenta il primo valore del nodo foglia corrispondente, in modo che il puntatore $P_i$ punti al sottoalbero con valori $K_{i-1} \leq K < K_i$.

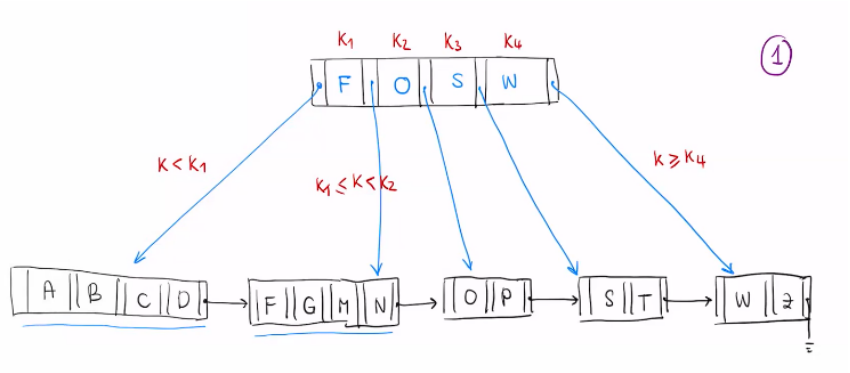



Figura 126: B$^+$-tree iniziale con unico nodo radice

**Passo 2: inserimento di $H$.**

Partendo dalla radice, il più piccolo valore di chiave maggiore di $H$ è $O$: si segue il puntatore corrispondente e si raggiunge il nodo foglia $(F, G, M, N)$. Inserendo $H$ si otterrebbe $(F, G, H, M, N)$, che contiene 5 valori di chiave, violando il vincolo di riempimento massimo ($\leq 4$).

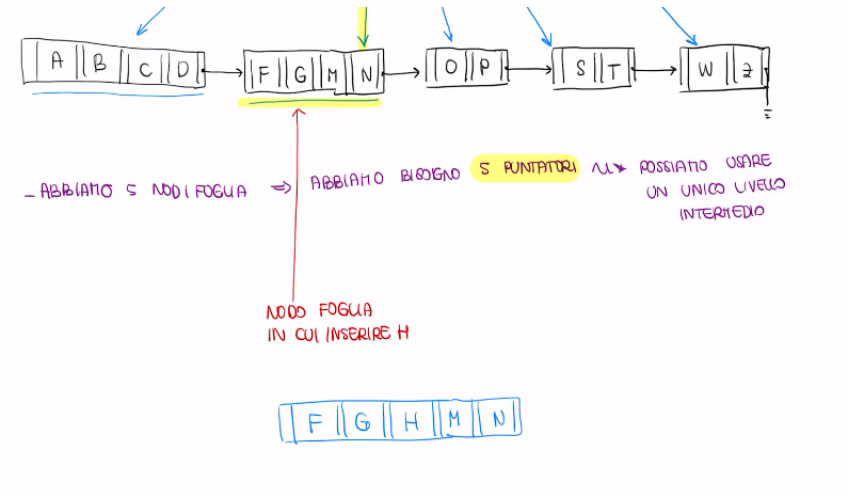



Figura 127: Il nodo foglia $(F, G, H, M, N)$ viola il vincolo di riempimento massimo

Si esegue uno **split**: si creano due nodi $NF_1$ e $NF_2$:

$$NF_1 = (F, G) \qquad NF_2 = (H, M, N)$$

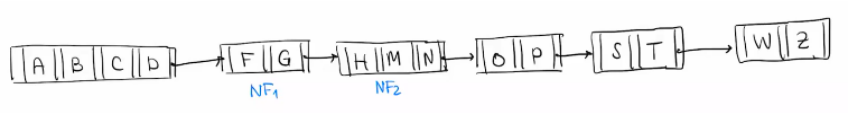



Figura 128: Risultato dello split: $NF_1 = (F, G)$, $NF_2 = (H, M, N)$

Ora abbiamo 6 nodi foglia, che richiedono 6 puntatori: la radice non può contenerne più di $n = 5$, quindi è necessario **propagare lo split** verso l'alto.

Si creano due nodi intermedi:

- il primo punta ai primi 3 nodi foglia, con chiavi $F$ e $H$;
- il secondo punta agli ultimi 3 nodi foglia, con chiavi $S$ e $W$.

La nuova radice contiene un unico valore di chiave $O$, che è il minimo valore del sottoalbero destro:

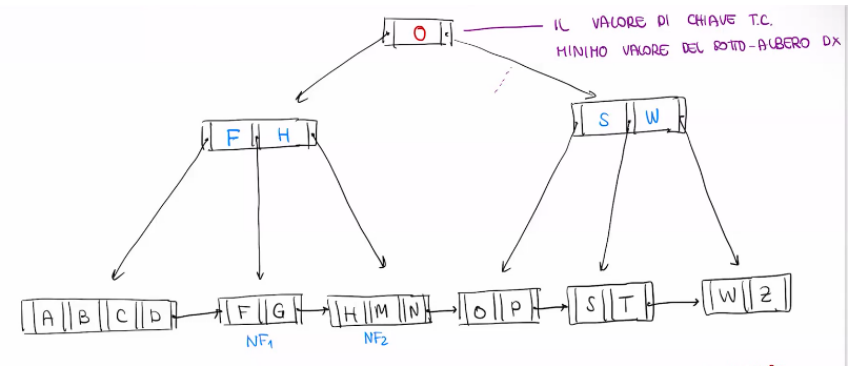



Figura 129: B$^+$-tree finale dopo l'inserimento di $H$ e la propagazione dello split

**Passo 3: eliminazione del valore $Z$**

Consideriamo il B$^+$-tree risultante dall'esercizio precedente e supponiamo di voler eliminare $K = Z$.

Partendo dalla radice, si scende verso destra fino a raggiungere il nodo foglia $(W, Z)$. Dopo la cancellazione di $Z$, il nodo foglia contiene un solo valore di chiave, violando il vincolo di riempimento minimo (#chiave $\geq 2$).

Si esegue un **merge** con il fratello $(S, T)$, ottenendo un unico nodo foglia $(S, T, W)$:

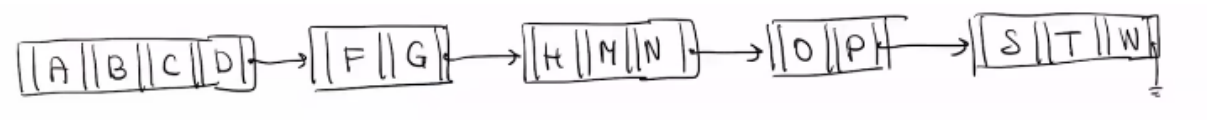



Figura 130: Risultato del merge: nodo foglia $(S, T, W)$

Ora abbiamo 5 nodi foglia, che richiedono 5 puntatori: è sufficiente un unico nodo intermedio (nuova radice) con chiavi $F, H, O, S$:

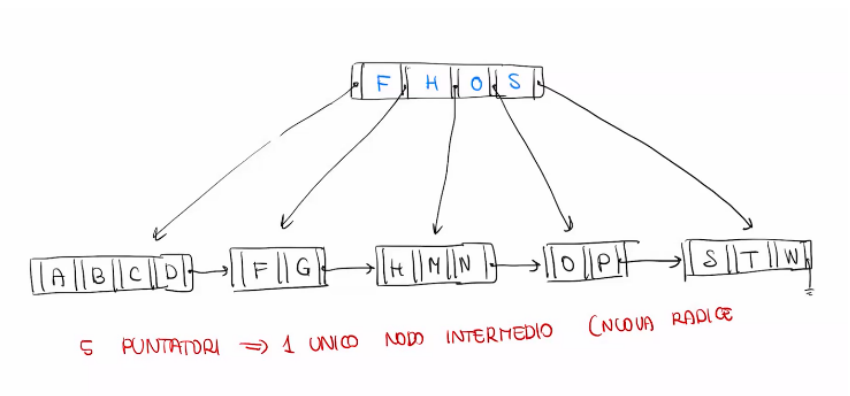



Figura 131: B$^+$-tree finale dopo la cancellazione di $Z$: unico nodo radice con chiavi $F, H, O, S$

## 12.7 Strutture ad accesso calcolato (Hash)

Le **strutture ad accesso calcolato** (o strutture hash) sono strutture dati che permettono un **accesso associativo** alle tuple, in cui la localizzazione fisica dei dati dipende dal valore della chiave.

Si utilizza una **funzione di hash** che mappa i valori della chiave in indirizzi di memoria:

$$h : K \to B$$

dove $K$ è il dominio delle chiavi e $B$ è il dominio degli indirizzi.

Il **costo della ricerca** è idealmente pari a 1 accesso alla memoria secondaria.

**Problema delle collisioni** Il principale problema delle funzioni di hash è quello delle **collisioni**: valori di chiave diversi possono produrre lo stesso indirizzo. Una funzione di hash è buona se **minimizza il numero di collisioni**.

**Dimensionamento della struttura** Viene definito un **fattore di riempimento** $f$, che indica la frazione di spazio fisico mediamente utilizzato in ciascun blocco.

Per memorizzare un file con $T$ tuple, il numero di blocchi necessari è:

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

dove:

- $T$ = numero di tuple;
- $F$ = fattore di blocco, ovvero il numero di tuple contenibili in un blocco;
- $f$ = fattore di riempimento.

Il DBMS alloca quindi un numero di bucket pari a $\#B$.

**Funzione di hash** La funzione di hash si compone di due passi:

1. Una **funzione di folding**:
$$g : K \to \mathbb{Z}^+$$
che trasforma i valori della chiave in interi positivi (necessaria poiché le chiavi non sono necessariamente interi).
2. Una **funzione di hashing**:
$$h : \mathbb{Z}^+ \to B$$
che mappa gli interi positivi nell'insieme dei bucket $B$.

La costruzione di una funzione hash performante non è banale e si basa su diverse euristiche. Un aspetto critico è che la funzione di hash è definita a priori in base alla dimensione $\#B$: modificarla in seguito richiederebbe una riorganizzazione completa del file.

**Esempio** Consideriamo un file con i seguenti parametri:

- $T = 7$ tuple;
- $F = 3$ tuple per blocco;
- $f = 0.4$ (fattore di riempimento).

Il numero di bucket è:

$$\#B = \left\lceil \frac{7}{3 \cdot 0.4} \right\rceil = 6$$

Il DBMS alloca quindi 6 bucket $B_1, \ldots, B_6$.

La funzione di hash produce i seguenti valori:

| **Chiave** | $h(g(k))$ |
|---|---|
| A | 2 |
| B | 3 |
| E | 5 |
| F | 3 |
| H | 2 |

Tabella 2: Valori della funzione di hash per ciascuna chiave. $F$ e $H$ collidono rispettivamente con $B$ e $A$.

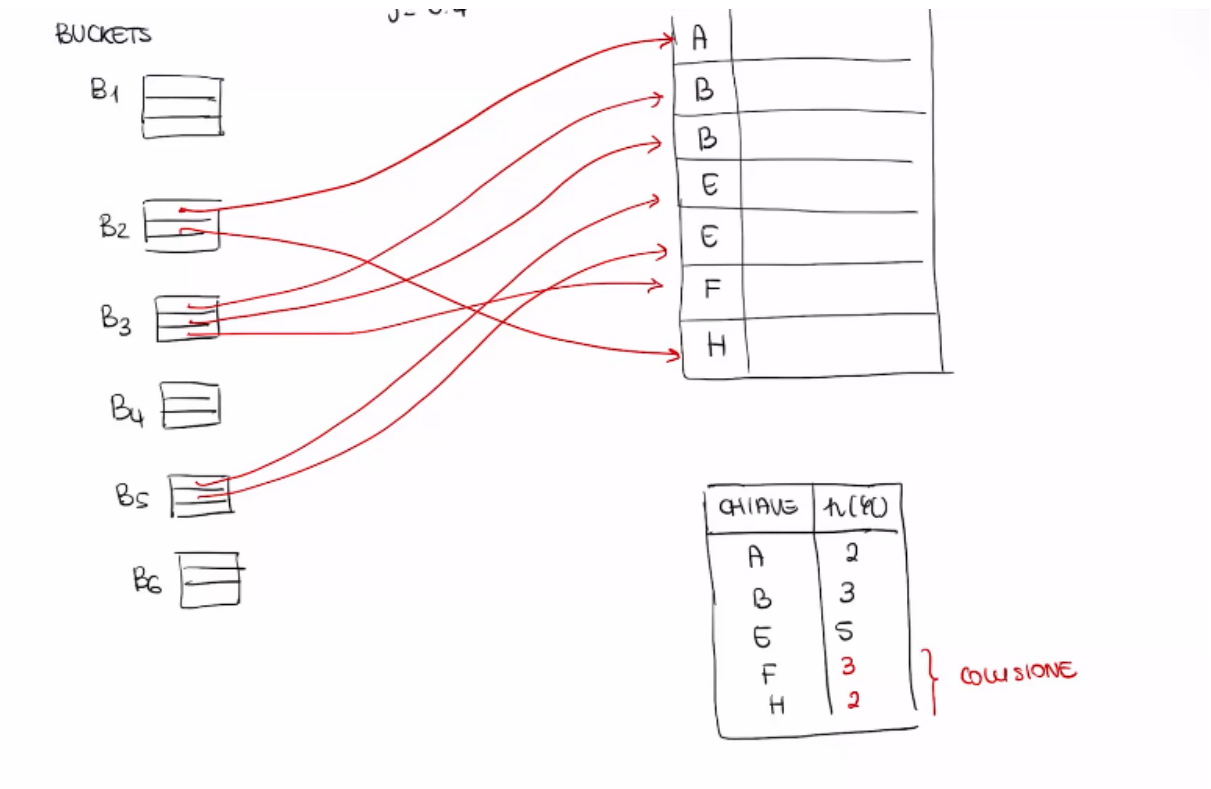



Figura 132: Mapping delle chiavi nei bucket tramite la funzione di hash

**Gestione delle collisioni: bucket di overflow** Quando un bucket è saturo a causa di troppe collisioni, si utilizza un **bucket di overflow**: si crea un nuovo bucket collegato al precedente tramite un puntatore, formando una catena. Il costo di accesso aumenta proporzionalmente alla lunghezza della catena.

**Ricerca con chiave $K$**

1. Calcolare $b = h(g(K))$: costo 0 (operazione in memoria centrale);
2. Accedere al bucket $b$: costo 1 accesso alla memoria secondaria;
3. Accedere alle tuple tramite i puntatori contenuti nel bucket: costo $m \leq n$ accessi, dove $m$ è il numero di pagine distinte da leggere.

### 12.7.1 Confronto tra B$^+$-tree e Hash

1. **Prestazioni**: il costo di ricerca del B$^+$-tree è $O(\log n)$, mentre quello della struttura hash è $O(1)$ (costante).
2. **Applicabilità**:
   - Per query di uguaglianza ($A = c$): entrambe le strutture funzionano, ma la hash è preferibile grazie al costo costante.
   - Per query di intervallo ($c_1 < A < c_2$): la struttura hash non è applicabile, poiché non mantiene un ordinamento tra le chiavi. In questo caso il B$^+$-tree è la scelta corretta.

# 13 Gestione della concorrenza

Una base di dati deve essere gestita da più programmi contemporaneamente, e si verifica quindi il problema della concorrenza. Per misurare il carico di lavoro si utilizza il **TPS** (*Transaction Per Second*), un'unità di misura che quantifica le transazioni elaborate al secondo; per gestire il sistema con prestazioni accettabili, un valore tipico di TPS è compreso tra 100 e 1000.

La concorrenza può tuttavia produrre delle **anomalie**, note come problemi di concorrenza. Per gestire tali anomalie si introducono dei meccanismi di controllo delle esecuzioni concorrenti. Il componente del DBMS preposto a questo compito è il **gestore della concorrenza**, detto anche **scheduler**, il quale riceve le richieste di accesso ai dati e decide se autorizzarle o meno; quando non le autorizza immediatamente, decide di ritardarle.

## 13.1 Anomalie da esecuzione concorrente

Le principali anomalie che possono verificarsi durante l'esecuzione concorrente di transazioni sono le seguenti:

1. **Perdita di aggiornamento** (*lost update*): gli effetti di una transazione vengono sovrascritti e quindi persi da un'altra transazione eseguita concorrentemente.
2. **Lettura inconsistente** (*unrepeatable read*): accessi successivi alla stessa risorsa da parte della stessa transazione restituiscono risultati diversi.
3. **Lettura sporca** (*dirty read*): viene letto un dato che rappresenta uno stato intermedio di un'altra transazione, non ancora confermato.
4. **Aggiornamento fantasma** (*phantom update*): viene osservato uno stato intermedio che viola i vincoli di integrità.
5. **Inserimento fantasma** (*phantom insert*): valutazioni diverse di operazioni di aggregazione producono risultati diversi a causa di inserimenti concorrenti.

L'obiettivo sarebbe evitarle tutte; tuttavia, vedremo che sarà necessario adottare un compromesso tra il grado di isolamento dalle anomalie e le prestazioni della base di dati.

Utilizzeremo la seguente notazione per gli esempi:

- $t_i$ per indicare una transazione.
- $r_i(x)$ per indicare un'operazione di lettura effettuata dalla transazione $t_i$ sulla risorsa $x$.
- $w_i(x)$ per indicare un'operazione di scrittura effettuata dalla transazione $t_i$ sulla risorsa $x$.

### 13.1.1 Perdita di aggiornamento

Gli effetti di una transazione concorrente vengono sovrascritti e quindi persi. Consideriamo le seguenti transazioni:

$$t_1 : \text{BOT},\ r_1(x),\ x = x + 1,\ w_1(x),\ \text{commit},\ \text{EOT}$$
$$t_2 : \text{BOT},\ r_2(x),\ x = x + 1,\ w_2(x),\ \text{commit},\ \text{EOT}$$

| **Op.** $t_1$ | **Seg. mem. centrale** $t_1$ | **Memoria secondaria** | **Seg. mem. centrale** $t_2$ | **Op.** $t_2$ |
|---|---|---|---|---|
| Stato iniziale | | $x = 2$ | Stato iniziale | |
| BOT | | 2 | | |
| $r_1(x)$ | $2 \leftarrow$ | [2] | | |
| $x = x + 1$ | 3 | [2] | | |
| | 3 | [2] | | BOT |
| | 3 | $[2] \rightarrow$ | 2 | $r_2(x)$ |
| | 3 | [2] | 3 | $x = x + 1$ |
| | 3 | $[3] \leftarrow$ | 3 | $w_2(x)$ |
| $w_1(x)$ | $3 \rightarrow$ | [3] | | EOT |
| commit | | [3] | | |
| EOT | | [3] | | |

Tabella 3: Esempio di perdita di aggiornamento: $t_2$ sovrascrive il valore scritto da $t_1$.

Un'esecuzione seriale delle due transazioni avrebbe prodotto il risultato corretto $x = 4$; questa esecuzione concorrente ha invece prodotto $x = 3$, a causa della sovrascrittura dell'aggiornamento di $t_2$ da parte di $t_1$. Per evitare questo tipo di anomalia è necessario ricorrere a opportune tecniche di controllo della concorrenza.

### 13.1.2 Lettura inconsistente

Accessi successivi allo stesso dato all'interno della stessa transazione producono valori diversi. Consideriamo le seguenti transazioni:

$$t_1 : \text{BOT},\ r_1(x),\ r_1'(x),\ \text{commit},\ \text{EOT}$$
$$t_2 : \text{BOT},\ r_2(x),\ x = x + 1,\ w_2(x),\ \text{commit},\ \text{EOT}$$

Si noti che in $t_1$ le operazioni $r_1(x)$ e $r_1'(x)$ rappresentano due letture successive dello stesso dato $x$ all'interno della stessa transazione.

| **Op.** $t_1$ | **Seg. mem. centrale** $t_1$ | **Memoria secondaria** | **Seg. mem. centrale** $t_2$ | **Op.** $t_2$ |
|---|---|---|---|---|
| Stato iniziale | | $x = 2$ | Stato iniziale | |
| BOT | | 2 | | |
| $r_1(x)$ | $2 \leftarrow$ | 2 | | |
| | 2 | 2 | | BOT |
| | 2 | $2 \rightarrow$ | 2 | $r_2(x)$ |
| | 2 | 2 | 3 | $x = x + 1$ |
| | 2 | $3 \leftarrow$ | 3 | $w_2(x)$ |
| | 2 | 3 | 3 | commit |
| | 2 | 3 | 3 | EOT |
| $r_1'(x)$ | $3 \leftarrow$ | 3 | | |
| commit | | 3 | | |
| EOT | | 3 | | |

Tabella 4: Esempio di lettura inconsistente: $t_1$ legge $x = 2$ la prima volta e $x = 3$ la seconda, pur non avendo mai modificato $x$.

Si può notare che il valore letto da $t_1$ per $x$ è cambiato tra la prima e la seconda lettura, nonostante $t_1$ non abbia mai modificato $x$.

### 13.1.3 Lettura sporca

Viene letto un dato che rappresenta uno stato intermedio nell'evoluzione di una transazione. Consideriamo le seguenti transazioni:

$$t_1 : \text{BOT},\ r_1(x),\ \text{commit},\ \text{EOT}$$
$$t_2 : \text{BOT},\ r_2(x),\ x = x + 1,\ w_2(x),\ \text{abort},\ \text{EOT}$$

Poiché $t_2$ esegue un **abort**, la transazione viene annullata e i suoi effetti sulla base di dati vengono ripristinati (*rollback*): è come se $t_2$ non fosse mai stata eseguita.

| **Op.** $t_1$ | **Mem. centr.** $t_1$ | **Memoria secondaria** | **Mem. centr.** $t_2$ | **Op.** $t_2$ |
|---|---|---|---|---|
| Stato iniziale | | $x = 2$ | Stato iniziale | |
| BOT | | 2 | | |
| | | 2 | | BOT |
| | | $2 \rightarrow$ | 2 | $r_2(x)$ |
| | | 2 | 3 | $x = x + 1$ |
| | | $3 \leftarrow$ | 3 | $w_2(x)$ |
| $r_1(x)$ | $3 \leftarrow$ | 3 | | |
| commit | 3 | 2 | | abort |
| EOT | | 2 | | EOT |

Tabella 5: Esempio di lettura sporca: $t_1$ legge il valore $x = 3$ scritto da $t_2$, che però esegue abort e ripristina $x = 2$.

Il valore 3 letto da $t_1$ non è conforme al contenuto reale della memoria secondaria: poiché $t_2$ ha eseguito l'abort, i suoi effetti sono stati annullati e il valore corretto di $x$ rimane 2. $t_1$ ha quindi letto uno stato intermedio che non avrebbe dovuto essere visibile.

### 13.1.4 Aggiornamento fantasma

Osservazione di uno stato intermedio che non soddisfa i vincoli di integrità. Supponiamo di avere tre risorse $x$, $y$, $z$ con il vincolo che la loro somma debba essere maggiore o uguale a 100:

$$x + y + z \geq 100$$

Consideriamo le seguenti transazioni:

$$t_1 : \text{BOT},\ r_1(y),\ r_1(x),\ r_1(z),\ S = x + y + z,\ \text{EOT}$$
$$t_2 : \text{BOT},\ r_2(y),\ y = y + 10,\ r_2(z),\ z = z - 10,\ w_2(y),\ w_2(z),\ \text{commit},\ \text{EOT}$$

Si noti che $t_2$ non viola il vincolo: sposta semplicemente 10 unità da $z$ a $y$, lasciando invariata la somma. Tuttavia, a causa dell'interleaving, $t_1$ può osservare uno stato intermedio in cui il vincolo risulta violato.

| **Op.** $t_1$ | **Mem. centr.** $t_1$ | **Memoria secondaria** | **Mem. centr.** $t_2$ | **Op.** $t_2$ |
|---|---|---|---|---|
| Stato iniziale | | $(x, y, z) = 20, 20, 60$ | Stato iniziale | |
| BOT | | 20, 20, 60 | | |
| $r_1(y)$ | $-, 20, - \leftarrow$ | 20, 20, 60 | | |
| | $-, 20, -$ | 20, 20, 60 | | BOT |
| | $-, 20, -$ | 20, 20, 60 $\rightarrow$ | $-, 20, -$ | $r_2(y)$ |
| | $-, 20, -$ | 20, 20, 60 | $-, 30, -$ | $y = y + 10$ |
| | $-, 20, -$ | 20, 20, 60 $\rightarrow$ | $-, 30, 60$ | $r_2(z)$ |
| | $-, 20, -$ | 20, 20, 60 | $-, 30, 50$ | $z = z - 10$ |
| | $-, 20, -$ | 20, 30, 60 $\leftarrow$ | $-, 30, 50$ | $w_2(y)$ |
| | $-, 20, -$ | 20, 30, 50 $\leftarrow$ | $-, 30, 50$ | $w_2(z)$ |
| | $-, 20, -$ | 20, 30, 50 | | commit (vincolo OK) |
| $r_1(x)$ | 20, 20, 50 | 20, 30, 50 | | |
| $r_1(z)$ | 20, 20, 50 | | | vincolo violato per $t_1$! |

Tabella 6: Aggiornamento fantasma: $t_2$ non viola il vincolo, ma $t_1$ osserva uno stato intermedio in cui $x + y + z = 20 + 20 + 50 = 90 < 100$.

### 13.1.5 Inserimento fantasma

Valutazioni diverse di operatori di aggregazione (SUM, COUNT, AVG, MAX, MIN) all'interno della stessa transazione producono risultati diversi. È un'anomalia simile alla lettura inconsistente, ma coinvolge operatori di aggregazione anziché singoli valori.

La differenza sostanziale rispetto alle anomalie precedenti risiede nella natura della risorsa coinvolta: non si tratta di un singolo dato, ma dell'intera tabella. L'anomalia si manifesta quando, tra due valutazioni successive dello stesso operatore di aggregazione, un'altra transazione inserisce nuove tuple, alterando il risultato.

## 13.2 Schedule

Uno **schedule** è una sequenza di operazioni di lettura e scrittura eseguite su risorse della base di dati da transazioni concorrenti. Date le seguenti transazioni:

$$t_1 : \text{BOT},\ r_1(x),\ w_1(x),\ \text{commit},\ \text{EOT}$$
$$t_2 : \text{BOT},\ r_2(x),\ w_2(x),\ \text{commit},\ \text{EOT}$$

Un possibile schedule è ad esempio:

$$S : r_1(x),\ r_2(x),\ w_2(x),\ w_1(x)$$

Date due o più transazioni, esistono più schedule possibili. L'obiettivo è accettare solo gli schedule che evitano le anomalie, ovvero gli schedule equivalenti a uno schedule seriale.

Uno **schedule seriale** è uno schedule in cui le operazioni di ogni transazione compaiono in sequenza, senza essere inframmezzate da operazioni di altre transazioni. Ad esempio:

$$S_1 : r_1(x),\ r_2(x),\ w_2(x),\ w_1(x) \quad \text{(non seriale)}$$
$$S_2 : r_1(x),\ w_1(x),\ r_2(x),\ w_2(x) \quad \text{(seriale)}$$
$$S_3 : r_2(x),\ w_2(x),\ r_1(x),\ w_1(x) \quad \text{(seriale)}$$

L'obiettivo del gestore della concorrenza è individuare gli schedule **serializzabili**, ovvero equivalenti a uno schedule seriale. Per equivalenza si intende che lo schedule produce lo stesso effetto sulla base di dati che si otterrebbe eseguendo le transazioni una di seguito all'altra. È quindi necessario definire formalmente quando uno schedule è equivalente a uno schedule seriale.

### 13.2.1 Ipotesi di commit-proiezione

Per semplificare l'analisi, supponiamo che l'esito di tutte le transazioni sia noto a priori. In base a questa ipotesi, escludiamo dagli schedule le operazioni delle transazioni che terminano con un abort, considerando quindi solo le transazioni che giungono a buon fine con un commit.

Introdurremo due nozioni di equivalenza tra schedule:

1. **View equivalenza**
2. **Conflict equivalenza**

### 13.2.2 View-Equivalenza

La view equivalenza si basa su due relazioni fondamentali: *legge-da* e *scritture finali*.

**Relazione legge-da.** Dato uno schedule $S$, si dice che un'operazione di lettura $r_i(x)$ **legge-da** un'operazione di scrittura $w_j(x)$ se:

1. $w_j(x)$ precede $r_i(x)$ in $S$;
2. $j \neq i$;
3. non esiste alcuna operazione $w_k(x)$ tra $w_j(x)$ e $r_i(x)$, ovvero $w_j(x)$ è la scrittura di $x$ immediatamente precedente a $r_i(x)$.

Consideriamo i seguenti esempi:

$$S_1 : r_1(x),\ r_2(x),\ w_2(x),\ w_1(x) \quad \Rightarrow \quad \text{LEGGE_DA}(S_1) = \emptyset$$

($r_1(x)$ non è preceduta da alcuna scrittura; $r_2(x)$ è preceduta solo da $r_1(x)$, non da una scrittura)

$$S_2 : r_1(x),\ w_1(x),\ r_2(x),\ w_2(x) \quad \Rightarrow \quad \text{LEGGE_DA}(S_2) = \{(r_2(x),\ w_1(x))\}$$

($r_2(x)$ è preceduta da $w_1(x)$, che è la scrittura di $x$ immediatamente precedente)

$$S_3 : r_1(x),\ w_1(x),\ w_2(x),\ r_1(x) \quad \Rightarrow \quad \text{LEGGE_DA}(S_3) = \emptyset$$

(la seconda $r_1(x)$ è preceduta da $w_2(x)$, ma $i = 1 = k$, quindi la condizione $j \neq i$ non è soddisfatta)

**Relazione scritture finali.** Dato uno schedule $S$, un'operazione di scrittura $w_i(x)$ è una **scrittura finale** se è l'ultima operazione di scrittura su $x$ in $S$. Per ogni risorsa si individua quindi l'ultima transazione che ha effettuato una scrittura su di essa:

$$S_1 : r_1(x),\ r_2(x),\ w_2(x),\ w_3(y),\ w_1(x) \quad \Rightarrow \quad \text{SCRITTURE_FINALI}(S_1) = \{w_1(x),\ w_3(y)\}$$

**Definizione di view equivalenza.** Due schedule $S_1$ e $S_2$ sono **view-equivalenti** ($S_1 \approx_V S_2$) se e solo se possiedono lo stesso insieme legge-da e lo stesso insieme di scritture finali.

Uno schedule $S$ è **view-serializzabile** se è view-equivalente a uno schedule seriale.

**Esempio: perdita di aggiornamento.** Consideriamo le transazioni:

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

Uno schedule che rappresenta la perdita di aggiornamento è:

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

Calcoliamo le relazioni:

- LEGGE_DA($S_{pa}$) = $\emptyset$: né $r_1(x)$ né $r_2(x)$ sono precedute da alcuna scrittura sullo stesso dato, quindi nessuna lettura legge-da una scrittura.
- SCRITTURE_FINALI($S_{pa}$) = $\{w_1(x)\}$: l'ultima operazione di scrittura su $x$ è $w_1(x)$, che compare per ultima tra le scritture.

Per determinare se $S_{pa}$ è view-serializzabile, dobbiamo verificare se esiste uno schedule seriale view-equivalente. Gli unici schedule seriali possibili sono:

$$S_{12} : r_1(x),\ w_1(x),\ r_2(x),\ w_2(x)$$

- LEGGE_DA($S_{12}$) = $\{(r_2(x),\ w_1(x))\}$: $r_2(x)$ è preceduta immediatamente da $w_1(x)$.
- SCRITTURE_FINALI($S_{12}$) = $\{w_2(x)\}$: l'ultima scrittura su $x$ è $w_2(x)$.

$S_{12}$ non è view-equivalente a $S_{pa}$: differiscono sia nell'insieme legge-da che nelle scritture finali.

$$S_{21} : r_2(x),\ w_2(x),\ r_1(x),\ w_1(x)$$

- LEGGE_DA($S_{21}$) = $\{(r_1(x),\ w_2(x))\}$: $r_1(x)$ è preceduta immediatamente da $w_2(x)$.
- SCRITTURE_FINALI($S_{21}$) = $\{w_1(x)\}$: l'ultima scrittura su $x$ è $w_1(x)$.

$S_{21}$ non è view-equivalente a $S_{pa}$: pur avendo le stesse scritture finali, l'insieme legge-da è diverso.

Poiché nessuno dei due schedule seriali è view-equivalente a $S_{pa}$, possiamo concludere che **$\mathbf{S_{pa}}$ non è view-serializzabile**.

**Esempio: lettura inconsistente.** Consideriamo le transazioni:

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

Lo schedule che rappresenta la lettura inconsistente è:

$$S_{li} : r_1(x),\ r_2(x),\ w_2(x),\ r'_1(x)$$

- LEGGE_DA($S_{li}$) = $\{(r'_1(x),\ w_2(x))\}$: $r'_1(x)$ è preceduta immediatamente da $w_2(x)$, mentre $r_1(x)$ non è preceduta da alcuna scrittura.
- SCRITTURE_FINALI($S_{li}$) = $\{w_2(x)\}$: l'unica scrittura su $x$ è $w_2(x)$, che è quindi anche l'ultima.

Gli schedule seriali possibili sono:

$$S_{12} : r_1(x),\ r'_1(x),\ r_2(x),\ w_2(x)$$

- LEGGE_DA($S_{12}$) = $\emptyset$: nessuna lettura è preceduta da una scrittura.
- SCRITTURE_FINALI($S_{12}$) = $\{w_2(x)\}$.

$S_{12}$ non è view-equivalente a $S_{li}$: l'insieme legge-da è diverso.

$$S_{21} : r_2(x),\ w_2(x),\ r_1(x),\ r'_1(x)$$

- LEGGE_DA($S_{21}$) = $\{(r_1(x),\ w_2(x)),\ (r'_1(x),\ w_2(x))\}$: sia $r_1(x)$ che $r'_1(x)$ sono precedute immediatamente da $w_2(x)$.
- SCRITTURE_FINALI($S_{21}$) = $\{w_2(x)\}$.

$S_{21}$ non è view-equivalente a $S_{li}$: pur avendo le stesse scritture finali, l'insieme legge-da differisce — $S_{li}$ contiene solo $(r'_1(x),\ w_2(x))$, mentre $S_{21}$ contiene anche $(r_1(x),\ w_2(x))$.

Poiché $S_{li}$ non è view-equivalente ad alcuno schedule seriale, concludiamo che **$\mathbf{S_{li}}$ non è view-serializzabile**.

**Esempio: verifica di view-serializzabilità.** Consideriamo lo schedule:

$$S_1 : r_1(x),\ w_1(x),\ r_2(z),\ r_1(y),\ w_1(y),\ r_2(x),\ w_2(x),\ w_2(z)$$

Le transazioni coinvolte sono:

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

Calcoliamo le relazioni per $S_1$:

- LEGGE_DA($S_1$) = $\{(r_2(x),\ w_1(x))\}$: $r_2(x)$ è preceduta immediatamente da $w_1(x)$; $r_1(y)$, $r_2(z)$ e $r_1(x)$ non sono precedute da alcuna scrittura.
- SCRITTURE_FINALI($S_1$) = $\{w_2(x),\ w_1(y),\ w_2(z)\}$: $w_2(x)$ è l'ultima scrittura su $x$, $w_1(y)$ l'unica su $y$, $w_2(z)$ l'unica su $z$.

Gli schedule seriali possibili sono $S_{12}$ (prima $t_1$, poi $t_2$) e $S_{21}$ (prima $t_2$, poi $t_1$). Possiamo escludere immediatamente $S_{21}$: in $S_1$ la relazione legge-da impone $t_1 < t_2$ (poiché $r_2(x)$ legge-da $w_1(x)$), e le scritture finali impongono anch'esse $t_1 < t_2$ su $w_2(x)$; in $S_{21}$ l'ordine sarebbe invertito, quindi non può essere view-equivalente.

Verifichiamo $S_{12}$:

$$S_{12} : r_1(x),\ w_1(x),\ r_1(y),\ w_1(y),\ r_2(z),\ r_2(x),\ w_2(x),\ w_2(z)$$

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

Poiché LEGGE_DA($S_1$) = LEGGE_DA($S_{12}$) e SCRITTURE_FINALI($S_1$) = SCRITTURE_FINALI($S_{12}$), concludiamo che $S_1 \approx_V S_{12}$, quindi **$\mathbf{S_1}$ è view-serializzabile**.

**Esempio: schedule non view-serializzabile.** Consideriamo lo schedule:

$$S_2 : r_1(x),\ w_1(x),\ w_3(x),\ r_2(y),\ r_3(y),\ w_3(y),\ w_1(y),\ r_2(x)$$

Le transazioni coinvolte sono:

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

Calcoliamo le relazioni per $S_2$:

- LEGGE_DA($S_2$) = $\{(r_2(x),\ w_3(x))\}$: $r_2(x)$ è preceduta immediatamente da $w_3(x)$.
- SCRITTURE_FINALI($S_2$) = $\{w_3(x),\ w_1(y)\}$: $w_3(x)$ è l'ultima scrittura su $x$, $w_1(y)$ è l'ultima scrittura su $y$.

In linea di principio occorrerebbe esaminare tutte le 6 permutazioni seriali possibili. Tuttavia, l'insieme delle scritture finali ci permette di escluderle tutte a priori imponendo due vincoli sull'ordine delle transazioni:

- $w_3(x) \in$ SCRITTURE_FINALI impone $t_1 < t_3$, affinché $w_1(x)$ non sovrascriva $w_3(x)$;
- $w_1(y) \in$ SCRITTURE_FINALI impone $t_3 < t_1$, affinché $w_3(y)$ non sovrascriva $w_1(y)$.

I due vincoli sono contraddittori: non può esistere uno schedule seriale che soddisfi contemporaneamente $t_1 < t_3$ e $t_3 < t_1$. Pertanto, nessuna delle sei permutazioni può essere view-equivalente a $S_2$, e concludiamo che **$S_2$ non è view-serializzabile**.

**Esempio: schedule con 5 transazioni.** Consideriamo lo schedule:

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

Le transazioni coinvolte sono:

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

Calcoliamo le relazioni per $S_3$:

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

Ricaviamo ora i vincoli sull'ordine delle transazioni in qualsiasi schedule seriale view-equivalente a $S_3$.

**Dalle scritture finali:**

- $w_5(x) \in$ SCRITTURE_FINALI impone $t_2 < t_5$, $t_1 < t_5$, $t_3 < t_5$;
- $w_5(y) \in$ SCRITTURE_FINALI impone $t_3 < t_5$, $t_1 < t_5$;

- $w_1(z) \in$ SCRITTURE_FINALI non aggiunge vincoli ulteriori.

**Dall'insieme legge-da:**

- $(r_3(x),\ w_2(x)) \in$ LEGGE_DA impone $t_2 < t_3$, affinché $r_3(x)$ legga da $w_2(x)$;
- $(r_5(z),\ w_1(z)) \in$ LEGGE_DA impone $t_1 < t_5$.

**Dalla non appartenenza a legge-da:** Poiché $r_2(x)$ precede fisicamente $w_1(x)$ in $S_3$ e $(r_2(x),\ w_1(x)) \notin$ LEGGE_DA$(S_3)$, in qualsiasi schedule seriale equivalente $t_2$ deve precedere $t_1$, affinché $r_2(x)$ non legga da $w_1(x)$:

$$t_2 < t_1$$

**Contraddizione:** i vincoli ricavati impongono simultaneamente $t_2 < t_3$, $t_2 < t_1$ e $t_1 < t_5$, ma dall'insieme legge-da avevamo già ricavato $t_2 < t_3$ con $t_1 < t_2$ implicato dalla struttura dello schedule — il che genera un ciclo $t_2 < t_1 < t_2$. Non esiste quindi alcuno schedule seriale che soddisfi tutti i vincoli contemporaneamente, e concludiamo che $\mathbf{S_3}$ **non è view-serializzabile**.

## 13.3 Conflict equivalenza

**Conflitto.** Dato uno schedule $S$, si dice che una coppia di operazioni $(a_i, a_j)$ rappresenta un **conflitto** se:

- $i \neq j$, ovvero le operazioni appartengono a transazioni diverse;
- entrambe operano sulla stessa risorsa;
- almeno una delle due è un'operazione di scrittura;
- $a_i$ compare prima di $a_j$ in $S$.

**Conflict equivalenza.** Due schedule $S_1$ e $S_2$ sono **conflict-equivalenti** ($S_1 \approx_C S_2$) se e solo se possiedono lo stesso insieme di conflitti.

**Conflict-serializzabilità.** Uno schedule $S$ è **conflict-serializzabile** (CSR) se esiste uno schedule seriale $S_r$ tale che $S \approx_C S_r$. Il vantaggio rispetto alla view-serializzabilità è che l'algoritmo per il test della conflict-serializzabilità ha complessità **lineare**.

**Algoritmo CSR.** Dato uno schedule $S$, l'algoritmo procede come segue:

1. Costruire l'insieme dei conflitti di $S$.
2. Costruire il **grafo dei conflitti** $G(N, A)$, dove:
   - $N$ è l'insieme dei nodi, uno per ciascuna transazione di $S$;
   - $A$ contiene un arco $(t_i, t_j)$ se esiste un conflitto $(a_i, a_j) \in S$.
3. Se il grafo dei conflitti è **aciclico**, allora $S$ è conflict-serializzabile; altrimenti non lo è.

**Dimostrazione di correttezza. ($\Rightarrow$) Se $S$ è CSR allora il suo grafo è aciclico.** Sia $S_r$ lo schedule seriale conflict-equivalente a $S$, con le transazioni ordinate $t_1, t_2, \ldots, t_n$. Poiché $S \approx_C S_r$, i due schedule hanno esattamente gli stessi conflitti nello stesso ordine. Nel grafo dei conflitti di $S_r$ possono quindi comparire solo archi $(t_i, t_j)$ con $i < j$: un ciclo richiederebbe un arco $(t_j, t_i)$ con $j > i$, che implicherebbe l'esistenza di un conflitto in ordine inverso — impossibile in uno schedule seriale. Poiché $S$ e $S_r$ hanno lo stesso grafo, anche il grafo di $S$ è aciclico.

**(⇐) Se il grafo di $S$ è aciclico allora $S$ è CSR.** Se il grafo è aciclico, esiste un **ordinamento topologico**, ovvero una numerazione dei nodi tale che il grafo contenga solo archi $(t_i, t_j)$ con $i < j$. Utilizzando questo ordinamento topologico possiamo definire un ordine delle transazioni che produce lo stesso grafo dei conflitti di $S$: otteniamo così uno schedule seriale $S_r$ conflict-equivalente a $S$, quindi $S$ è CSR.

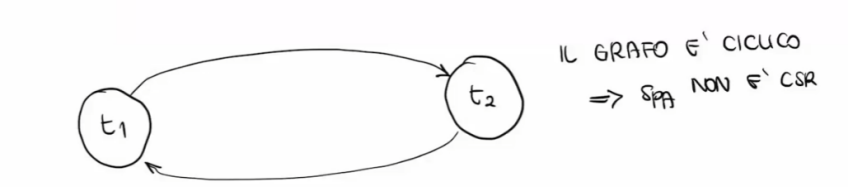
**Esempio CSR: perdita di aggiornamento.** Consideriamo lo schedule $S_{pa}$ già visto in precedenza:

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

Costruiamo l'insieme dei conflitti:

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

- $(r_1(x),\ w_2(x))$: $r_1$ e $w_2$ operano sulla stessa risorsa $x$, appartengono a transazioni diverse, e $w_2$ è una scrittura;
- $(r_2(x),\ w_1(x))$: analogo, con $w_1$ scrittura;
- $(w_2(x),\ w_1(x))$: entrambe scritture sulla stessa risorsa $x$, transazioni diverse.




Dal primo conflitto otteniamo un arco $t_1 \to t_2$; dal secondo e dal terzo un arco $t_2 \to t_1$. Il grafo dei conflitti contiene quindi un ciclo $t_1 \to t_2 \to t_1$, pertanto $\mathbf{S_{pa}}$ **non è CSR**.

**Esempio CSR: lettura inconsistente.** Consideriamo le transazioni:

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

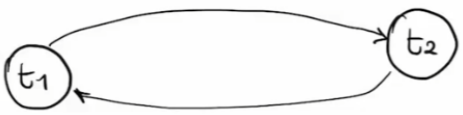
Lo schedule è:

$$S_{li} : r_1(x),\ r_2(x),\ w_2(x),\ r_1'(x)$$

Costruiamo l'insieme dei conflitti:

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

- $(r_1(x),\ w_2(x))$: $r_1$ precede $w_2$, stessa risorsa $x$, transazioni diverse, $w_2$ è scrittura $\Rightarrow$ arco $t_1 \to t_2$;
- $(w_2(x),\ r_1'(x))$: $w_2$ precede $r_1'$, stessa risorsa $x$, transazioni diverse, $w_2$ è scrittura $\Rightarrow$ arco $t_2 \to t_1$.




IL GRAFO HA UN CICLO
=> $S_{li}$ NON E' CSR

Il grafo dei conflitti contiene un ciclo $t_1 \to t_2 \to t_1$, pertanto $\mathbf{S_{li}}$ **non è CSR**.

## 13.4 Relazione tra CSR e VSR

La conflict-serializzabilità è una condizione **sufficiente ma non necessaria** per la view-serializzabilità. In altri termini:

- $S \in \text{CSR} \Rightarrow S \in \text{VSR}$: se uno schedule è conflict-serializzabile, allora è anche view-serializzabile.
- $S \notin \text{CSR} \not\Rightarrow S \notin \text{VSR}$: se uno schedule non è conflict-serializzabile, non possiamo concludere nulla sulla sua view-serializzabilità.
- $S \notin \text{VSR} \Rightarrow S \notin \text{CSR}$: se uno schedule non è view-serializzabile, allora non è nemmeno conflict-serializzabile.

Quindi l'insieme CSR è un sottoinsieme proprio di VSR: esistono schedule view-serializzabili che non sono conflict-serializzabili, ma non viceversa.

**Strategia per gli esercizi.** Data uno schedule $S$, la procedura da seguire è:

1. Applicare il **test CSR**: costruire il grafo dei conflitti e verificare se è aciclico.
   - Se $S$ è CSR $\Rightarrow$ $S$ è anche VSR. Non è necessario proseguire.
   - Se $S$ non è CSR $\Rightarrow$ applicare il **test VSR** per determinare se $S$ è comunque view-serializzabile.

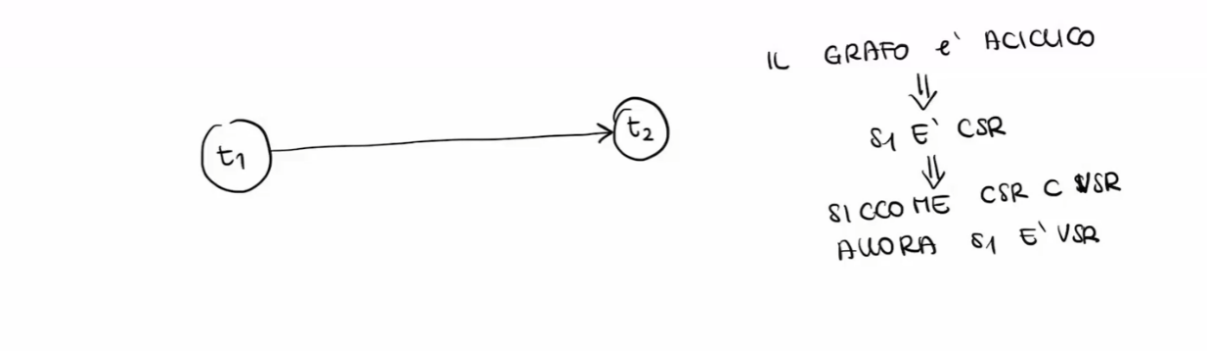
**Esempio CSR: schedule $S_1$.** Riprendiamo lo schedule già analizzato in precedenza:

$$S_1 : r_1(x),\ w_1(x),\ r_2(z),\ r_1(y),\ w_1(y),\ r_2(x),\ w_2(x),\ w_2(z)$$

Costruiamo l'insieme dei conflitti:

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

- $(r_1(x),\ w_2(x))$: $r_1$ precede $w_2$, stessa risorsa $x$, transazioni diverse, $w_2$ è scrittura $\Rightarrow$ arco $t_1 \to t_2$;
- $(w_1(x),\ r_2(x))$: $w_1$ precede $r_2$, stessa risorsa $x$, transazioni diverse, $w_1$ è scrittura $\Rightarrow$ arco $t_1 \to t_2$;
- $(w_1(x),\ w_2(x))$: entrambe scritture su $x$, transazioni diverse, $w_1$ precede $w_2$ $\Rightarrow$ arco $t_1 \to t_2$.




Tutti e tre i conflitti generano archi nella stessa direzione $t_1 \to t_2$. Il grafo dei conflitti è quindi aciclico, pertanto $S_1$ è CSR. Poiché CSR $\subseteq$ VSR, concludiamo che **$\mathbf{S_1}$ è sia CSR che VSR**.

**Esempio CSR: schedule $S_2$.** Riprendiamo lo schedule $S_2$ già analizzato in precedenza:

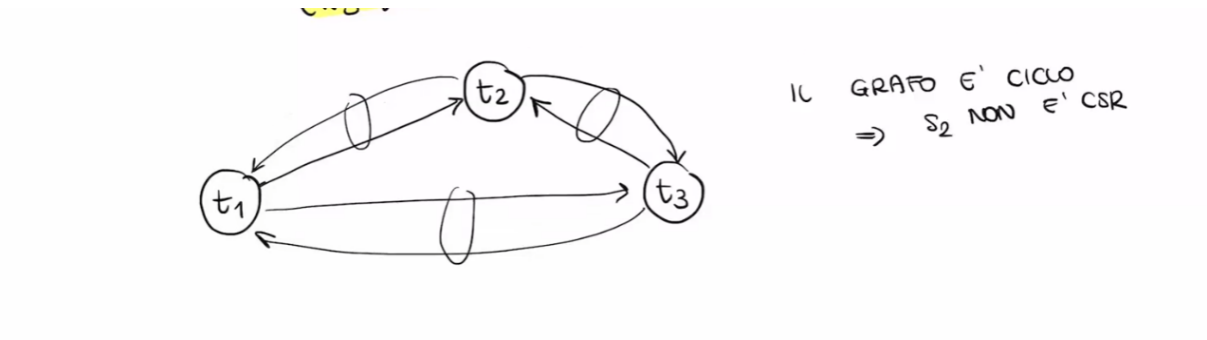
$$S_2 : r_1(x),\ w_1(x),\ w_3(x),\ r_2(y),\ r_3(y),\ w_3(y),\ w_1(y),\ r_2(x)$$

Costruiamo l'insieme dei conflitti:

CONFLITTI($S_2$) = $\{(r_1(x),\ w_3(x)),\ (w_1(x),\ w_3(x)),\ (w_1(x),\ r_2(x)),\ (w_3(x),\ r_2(x)),\ (r_2(y),\ w_3(y)),\ (r_2(y),\ w_1$

Ricaviamo gli archi del grafo dei conflitti:

- $(r_1(x),\ w_3(x))$, $(w_1(x),\ w_3(x)) \Rightarrow$ arco $t_1 \to t_3$;
- $(w_1(x),\ r_2(x)) \Rightarrow$ arco $t_1 \to t_2$;
- $(w_3(x),\ r_2(x)) \Rightarrow$ arco $t_3 \to t_2$;
- $(r_2(y),\ w_3(y)) \Rightarrow$ arco $t_2 \to t_3$;
- $(r_2(y),\ w_1(y)) \Rightarrow$ arco $t_2 \to t_1$;
- $(r_3(y),\ w_1(y))$, $(w_3(y),\ w_1(y)) \Rightarrow$ arco $t_3 \to t_1$.




Il grafo dei conflitti contiene cicli: $t_2 \to t_3 \to t_2$ e $t_1 \to t_3 \to t_1$. Il grafo è quindi ciclico, pertanto $S_2$ non è CSR. Poiché CSR $\subseteq$ VSR e avevamo già dimostrato che $S_2$ non è VSR, possiamo concludere che **$\mathbf{S_2}$ non è né CSR né VSR**.

**Esempio CSR: schedule $S_3$.** Riprendiamo lo schedule $S_3$ già analizzato in precedenza:

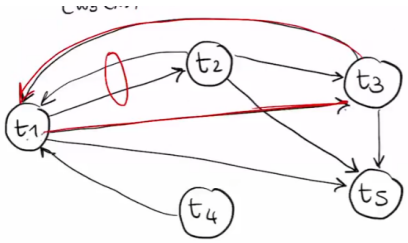
$$S_3 : r_1(x),\ r_2(x),\ w_2(x),\ r_3(x),\ r_4(z),\ w_1(x),\ w_3(y),\ w_3(x),\ w_1(y),\ w_5(x),\ w_1(z),\ w_5(y),\ r_5(z)$$

Costruiamo l'insieme dei conflitti:

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

Ricaviamo gli archi del grafo dei conflitti:

- $(r_1(x),\ w_2(x)) \Rightarrow$ arco $t_1 \to t_2$;
- $(r_1(x),\ w_3(x))$, $(r_1(x),\ w_5(x)) \Rightarrow$ archi $t_1 \to t_3$, $t_1 \to t_5$;
- $(r_2(x),\ w_1(x)) \Rightarrow$ arco $t_2 \to t_1$;
- $(r_2(x),\ w_3(x))$, $(r_2(x),\ w_5(x)) \Rightarrow$ archi $t_2 \to t_3$, $t_2 \to t_5$;
- $(w_2(x),\ r_3(x))$, $(w_2(x),\ w_1(x))$, $(w_2(x),\ w_3(x))$, $(w_2(x),\ w_5(x)) \Rightarrow$ archi $t_2 \to t_3$, $t_2 \to t_1$, $t_2 \to t_3$, $t_2 \to t_5$;
- $(r_3(x),\ w_1(x))$, $(r_3(x),\ w_5(x)) \Rightarrow$ archi $t_3 \to t_1$, $t_3 \to t_5$;
- $(r_4(z),\ w_1(z)) \Rightarrow$ arco $t_4 \to t_1$;
- $(w_1(x),\ w_3(x))$, $(w_1(x),\ w_5(x)) \Rightarrow$ archi $t_1 \to t_3$, $t_1 \to t_5$;
- $(w_3(y),\ w_1(y)) \Rightarrow$ arco $t_3 \to t_1$;
- $(w_3(y),\ w_5(y))$, $(w_3(x),\ w_5(x)) \Rightarrow$ archi $t_3 \to t_5$;
- $(w_1(y),\ w_5(y)) \Rightarrow$ arco $t_1 \to t_5$.




S3 NON E' CSR

Il grafo contiene cicli, ad esempio $t_1 \to t_2 \to t_1$ (dagli archi $t_1 \to t_2$ e $t_2 \to t_1$) e $t_2 \to t_3 \to t_1 \to t_2$. Il grafo è quindi ciclico, pertanto $S_3$ non è CSR. Poiché avevamo già dimostrato che $S_3$ non è VSR, concludiamo che **$S_3$ non è né CSR né VSR**.

**Esempio CSR: schedule $S_4$.** Consideriamo lo schedule:

$$S_4 : r_1(x),\ r_3(y),\ w_1(y),\ w_4(x),\ w_1(t),\ w_5(x),\ r_2(z),\ r_3(z),\ w_2(z),\ w_5(z),\ r_4(t),\ r_5(t)$$

Costruiamo l'insieme dei conflitti:

$$\begin{aligned}\text{CONFLITTI}(S_4) = \{&(r_1(x),\ w_4(x)),\ (r_1(x),\ w_5(x)),\\ &(r_3(y),\ w_1(y)),\\ &(w_4(x),\ w_5(x)),\\ &(w_1(t),\ r_4(t)),\ (w_1(t),\ r_5(t)),\\ &(r_2(z),\ w_5(z)),\\ &(r_3(z),\ w_2(z)),\ (r_3(z),\ w_5(z)),\\ &(w_2(z),\ w_5(z))\}\end{aligned}$$

Ricaviamo gli archi del grafo dei conflitti:

- $(r_1(x),\ w_4(x))$, $(r_1(x),\ w_5(x)) \Rightarrow$ archi $t_1 \to t_4$, $t_1 \to t_5$;
- $(r_3(y),\ w_1(y)) \Rightarrow$ arco $t_3 \to t_1$;
- $(w_4(x),\ w_5(x)) \Rightarrow$ arco $t_4 \to t_5$;
- $(w_1(t),\ r_4(t))$, $(w_1(t),\ r_5(t)) \Rightarrow$ archi $t_1 \to t_4$, $t_1 \to t_5$;
- $(r_2(z),\ w_5(z)) \Rightarrow$ arco $t_2 \to t_5$;
- $(r_3(z),\ w_2(z))$, $(r_3(z),\ w_5(z)) \Rightarrow$ archi $t_3 \to t_2$, $t_3 \to t_5$;
- $(w_2(z),\ w_5(z)) \Rightarrow$ arco $t_2 \to t_5$.

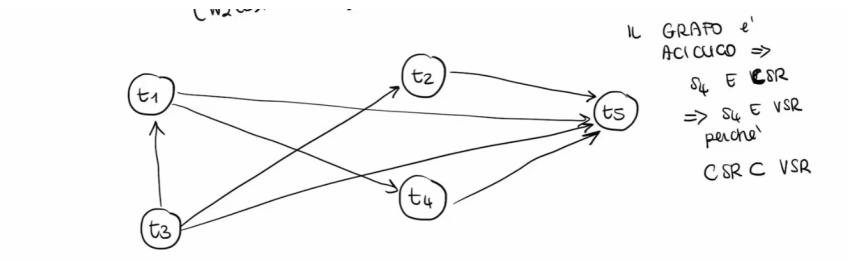



Figura 133:

Il grafo dei conflitti ha quindi archi: $t_3 \to t_1 \to t_4 \to t_5$, $t_3 \to t_2 \to t_5$, $t_1 \to t_5$, $t_3 \to t_5$. Il grafo è **aciclico**, pertanto $S_4$ è CSR. Poiché CSR $\subseteq$ VSR, concludiamo che **$S_4$ è sia CSR che VSR**.

**Ordinamenti topologici di $S_4$.** Poiché il grafo dei conflitti di $S_4$ è aciclico, possiamo ricavare gli schedule seriali conflict-equivalenti tramite l'ordinamento topologico. Il nodo di partenza è $t_3$, l'unico senza archi entranti. I possibili ordinamenti topologici sono:

$$\begin{aligned}S_{r1} &: t_3,\ t_1,\ t_5,\ t_2,\ t_4\\ S_{r2} &: t_3,\ t_1,\ t_5,\ t_4,\ t_2\\ S_{r3} &: t_3,\ t_2,\ t_1,\ t_4,\ t_5\end{aligned}$$

Tutti e tre gli schedule seriali sono conflict-equivalenti e view-equivalenti a $S_4$.

## 13.5 Locking a due fasi stretto (2PL stretto)

Il locking a due fasi stretto è il meccanismo principale per la gestione della concorrenza nei DBMS. I suoi vantaggi rispetto alla view-serializzabilità sono:

- non richiede di conoscere in anticipo l'esito delle transazioni;
- fornisce un meccanismo base per la gestione dei lock;
- definisce una politica di concessione dei lock;
- garantisce la serializzabilità tramite una regola precisa.

### 13.5.1 Meccanismo di base

Per bloccare le risorse su cui si vuole operare si introducono le seguenti primitive di lock:

- **R_LOCK**$_i(x)$: la transazione $t_i$ richiede un lock condiviso in lettura sulla risorsa $x$;
- **W_LOCK**$_i(x)$: la transazione $t_i$ richiede un lock esclusivo in scrittura sulla risorsa $x$;
- **UNLOCK**$_i(x)$: la transazione $t_i$ rilascia la risorsa $x$ da un precedente lock.

**Regole d'uso delle primitive.**

- Ogni operazione di **lettura** deve essere preceduta da una R_LOCK e seguita da una UNLOCK. Sono ammessi più R_LOCK contemporanei sulla stessa risorsa, poiché il lock in lettura è **condiviso**.
- Ogni operazione di **scrittura** deve essere preceduta da una W_LOCK e seguita da una UNLOCK. Non sono ammessi più W_LOCK contemporanei sulla stessa risorsa, poiché il lock in scrittura è **esclusivo**.

Una transazione che rispetta queste regole si dice **ben formata**.

### 13.5.2 Politica di concessione dei lock

Il gestore della concorrenza del DBMS mantiene per ogni risorsa $x$ le seguenti informazioni:

- stato$(x) \in \{$libero, r_lock, w_lock$\}$: lo stato corrente della risorsa;
- $C(x) = \{t_1, \ldots, t_n\}$: l'insieme delle transazioni che detengono un R_LOCK su $x$ (rilevante solo quando stato$(x) =$ r_lock).

| STATO x / RICHIESTA | LIBERO | | r_LOCK | | W_LOCK | |
|---|---|---|---|---|---|---|
| | ESITO | OPERAZIONI | ESITO | OPERAZIONI | ESITO | OPERAZIONI |
| $r_lock_i(x)$ | V | $s(x) = r_lock$<br>$c(x) = \{t_i\}$ | V | $c(x) = c(x) \cup \{t_k\}$ | ATTESA | – |
| $w_lock_i(x)$ | V | $s(x) = w_lock$ | IF $\lvert c(x) \rvert = 1 \wedge t_k \in c(x)$<br>V: $s(x) = w_lock$<br>ELSE<br>ATTESA: – | | ATTESA | – |
| $unlock_i(x)$ | ERRORE | – | V | $c(x) = c(x) \setminus \{t_k\}$<br>IF $c(x) = \emptyset$<br>THEN $s(x) = LIBERO$<br>ELSE VERIFICO la CODA | V | $s(x) = LIBERO$<br>VERIFICO CODA |

Figura 134: Tabella di concessione dei lock

## 13.6 Regola per la serializzabilità

La regola da cui si ottiene la serializzabilità, e da cui deriva il nome *2PL*, stabilisce che **una transazione, dopo aver rilasciato un lock, non può acquisirne altri**.

Il ciclo di vita di una transazione è suddiviso in due fasi distinte:

1. **Fase di acquisizione (Growing Phase):** il numero di lock acquisiti cresce monotonicamente.
2. **Fase di rilascio (Shrinking Phase):** il numero di lock decresce monotonicamente; non è più possibile acquisire nuovi lock.

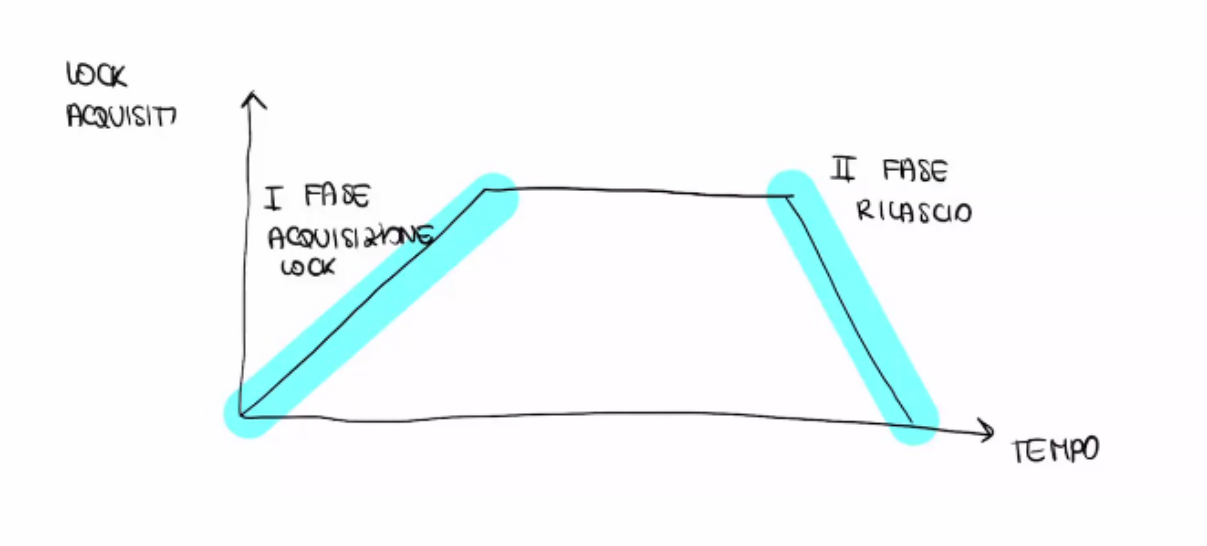



Figura 135: ciclo vita lock

**Esempio: perdita di aggiornamento con 2PL stretto.** Consideriamo le transazioni:

$$t_1 : \text{R_LOCK}_1(x),\ r_1(x),\ \text{W_LOCK}_1(x),\ w_1(x),\ \text{UNLOCK}_1(x)$$
$$t_2 : \text{R_LOCK}_2(x),\ r_2(x),\ \text{W_LOCK}_2(x),\ w_2(x),\ \text{UNLOCK}_2(x)$$

Lo schedule con perdita di aggiornamento era:

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

Con il 2PL stretto, introduciamo esplicitamente le operazioni di lock:

$$S_{pa}^{2PL} : \text{R_LOCK}_1(x),\ r_1(x),\ \text{R_LOCK}_2(x),\ r_2(x),\ \text{W_LOCK}_2(x)\ [?],\ \ldots$$

Quando $t_2$ tenta di acquisire $\text{W_LOCK}_2(x)$, lo stato della risorsa è:

$$\text{stato}(x) = \texttt{r_lock}, \quad C(x) = \{t_1,\ t_2\}$$

Dalla tabella di concessione, $\text{W_LOCK}_2(x)$ è concedibile solo se $C(x) = \{t_2\}$, ovvero se $t_1$ ha già rilasciato il proprio $\text{R_LOCK}_1(x)$. Tuttavia:

- Se $t_1$ rilascia $\text{R_LOCK}_1(x)$, entra nella **fase discendente** del 2PL.
- Nella fase discendente, $t_1$ non può più acquisire nuovi lock.
- Ma $t_1$ deve ancora eseguire $w_1(x)$, che richiede $\text{W_LOCK}_1(x)$: **impossibile**.

Il 2PL stretto impedisce quindi strutturalmente lo schedule $S_{pa}$: non esiste alcun modo di ordinare le operazioni di lock e unlock che rispetti contemporaneamente la regola del 2PL e consenta l'esecuzione di entrambe le scritture. La perdita di aggiornamento viene così **prevenuta**.

### 13.6.1 2PL stretto

Il **2PL stretto** aggiunge un vincolo ulteriore rispetto al 2PL base: una transazione può rilasciare i propri lock **esclusivamente dopo aver eseguito un commit o un rollback**. Il momento in cui inizia la fase discendente coincide quindi con il commit o l'abort della transazione.

Questo vincolo aggiuntivo permette di **eliminare l'ipotesi di commit-proiezione**: il 2PL stretto garantisce la serializzabilità anche in presenza di transazioni che terminano con un abort, senza dover assumere a priori l'esito delle transazioni.

### 13.6.2 Relazione tra 2PL stretto, CSR e VSR

I tre insiemi di schedule sono in relazione di inclusione stretta:

$$\text{2PL stretto} \subsetneq \text{CSR} \subsetneq \text{VSR}$$

- Ogni schedule prodotto dal 2PL stretto è conflict-serializzabile, ma non tutti gli schedule CSR sono producibili dal 2PL stretto.
- Ogni schedule CSR è view-serializzabile, ma non tutti gli schedule VSR sono conflict-serializzabili.

Il 2PL stretto è quindi la condizione più restrittiva: sacrifica parte della concorrenza ammessa da CSR e VSR, ma in cambio offre un meccanismo **pratico, efficiente e privo di ipotesi sull'esito delle transazioni**.

## 13.7 Blocco critico (Deadlock)

Il **blocco critico** (o *deadlock*) si verifica durante l'esecuzione concorrente di due o più transazioni quando si crea una situazione di attesa circolare:

- $t_1$ detiene una risorsa di cui $t_2$ ha bisogno, ed è in attesa di una risorsa detenuta da $t_2$;
- $t_2$ detiene una risorsa di cui $t_1$ ha bisogno, ed è in attesa di una risorsa detenuta da $t_1$.

Nessuna delle due transazioni può proseguire.

**Frequenza del deadlock.** Assumendo una granularità del lock a livello di tupla e indicando con $N$ il numero medio di tuple per tabella, la probabilità di deadlock di lunghezza due è:

$$P(\text{deadlock}) = \frac{1}{N^2}$$

All'aumentare del numero di tuple la probabilità decresce: con granularità fine il deadlock è un evento raro.

**Tecniche di gestione.**

1. **Timeout**: una transazione in attesa di una risorsa viene abortita allo scadere di un intervallo di tempo prefissato. È la tecnica più semplice da implementare, ma può abortire transazioni che non sono effettivamente in deadlock.
2. **Prevenzione**: ogni transazione acquisisce **tutte** le risorse di cui necessita in lettura e scrittura in un'unica soluzione, secondo una politica del tutto-o-niente. Elimina il deadlock alla radice, ma è molto costosa in termini di concorrenza, poiché le risorse vengono bloccate anche quando non sono ancora necessarie.
3. **Rilevamento**: è la tecnica adottata nella maggior parte dei sistemi reali. Periodicamente il sistema analizza la tabella dei lock alla ricerca di cicli di attesa; se viene rilevato un deadlock, una delle transazioni coinvolte viene abortita per spezzare il ciclo. Lo svantaggio è il rischio di **starvation**: la stessa transazione potrebbe essere ripetutamente scelta come vittima e non riuscire mai a completare.

## 13.8 Livelli di isolamento

Esiste un compromesso fondamentale tra il grado di isolamento dalle anomalie e le prestazioni del sistema (misurate in TPS): maggiore è il controllo della concorrenza, minore è il parallelismo ammesso e quindi le prestazioni. I DBMS consentono di rinunciare, in tutto o in parte, al controllo della concorrenza, specificando quali anomalie si è disposti a tollerare tramite i **livelli di isolamento**.

| ANOMALIA / LIV. ISOLAMENTO | PERDITA AGGIORNAMENTO | LETTURA SPORCA | LETTURA INCONSISTENTE | AGGIORN. FANTASMA LOCK RIGHE | INSERIM. FANTASMA LOCK TABELLA |
|---|---|---|---|---|---|
| SERIALIZABLE | V | V | V | V | V |
| REPEATABLE READ * | V | V | V | V | |
| READ COMMITTED | V | V | | | |
| READ UNCOMMITTED | V | | | | |

* POSTGRESQL ⇒ IMPLEMENTAZIONE PIU' STRINGENTE R.R. CHE PERMETTE DI EVITARE ANCHE INS. FANTASMA

Figura 136: Anomalie evitate (✓) per ciascun livello di isolamento

* *PostgreSQL* adotta un'implementazione più stringente del livello Repeatable Read, che permette di evitare anche l'inserimento fantasma.

Al livello più alto (**Serializable**) tutte le anomalie sono prevenute; scendendo di livello si tollerano anomalie via via più gravi, guadagnando in concorrenza e prestazioni.

### 13.8.1 Implementazione dei livelli di isolamento

Tutti i livelli di isolamento richiedono il **2PL stretto per le scritture**: i lock in scrittura vengono sempre rilasciati solo dopo il commit o il rollback. I livelli si differenziano invece nella gestione dei lock in lettura.

- **Serializable**: applica il 2PL stretto anche per le letture e acquisisce lock a livello di **tabella**, prevenendo così l'anomalia di inserimento fantasma.
- **Repeatable Read**: applica il 2PL stretto anche per le letture, ma acquisisce lock a livello di **tupla** anziché di tabella. Previene la lettura inconsistente e l'aggiornamento fantasma, ma ammette l'anomalia di inserimento fantasma.

- **Read Committed**: acquisisce lock condivisi in lettura, ma **non** segue la regola del 2PL stretto per essi: i lock in lettura vengono rilasciati non appena l'operazione di lettura è completata, senza attendere il commit. Previene la lettura sporca, ma ammette la lettura inconsistente.
- **Read Uncommitted**: non acquisisce alcun lock in lettura. È il livello con la massima concorrenza, ma ammette tutte le anomalie di lettura, compresa la lettura sporca.

# 14 Gestore delle ottimizzazioni

Il componente **gestore delle ottimizzazioni** riceve le richieste tramite **SQL**, che è un linguaggio *dichiarativo*. Questo componente trasforma la richiesta in un **piano di esecuzione**, rappresentato tramite un linguaggio *procedurale*.

Come linguaggio dichiarativo abbiamo visto **SQL** e il **calcolo relazionale**; come linguaggio procedurale abbiamo visto l'**algebra relazionale**.

In questo caso il DBMS determina il miglior piano di esecuzione, ottimizzando rispetto alle *caratteristiche fisiche* del database.

$$\text{Query} \Rightarrow \text{Analisi lessicale e sintattica} \Rightarrow \text{Ottimizzazione algebrica}$$

Le ottimizzazioni algebriche comprendono:

- **Anticipo delle selezioni**
- **Anticipo delle proiezioni**

Questa fase è *indipendente* dal contenuto della base di dati.

Il gestore delle ottimizzazioni esegue anche un'ottimizzazione che dipende dai **metodi di accesso**, ovvero dalle strutture fisiche e dalle **statistiche** (ad esempio, quante righe sono presenti in ogni tabella). Questa fase deve interagire con la parte del DBMS che contiene i **profili delle tabelle**. Solo dopo aver completato questa ottimizzazione è possibile generare il **piano di esecuzione**.

Le operazioni tipiche di accesso supportate dai DBMS sono:

- Scansione sequenziale delle tuple di una tabella
- Ordinamento delle tuple
- Accesso diretto tramite indice

## 14.1 Scansione sequenziale delle tuple di una tabella

È la prima operazione che viene eseguita e spesso viene effettuata contestualmente ad altre operazioni:

- **Scan + proiezione** senza eliminazione dei duplicati
- **Scan + selezione** rispetto a un predicato semplice

Il costo della scansione sequenziale è definito come:

$$\text{Costo} = NP(R)$$

dove:

- $NP$ = numero di pagine/blocchi
- $NP(R)$ = numero di pagine della relazione/tabella $R$

## 14.2 Operazione di ordinamento

L'ordinamento viene utilizzato nei seguenti contesti SQL:

- `ORDER BY` — ordinare il risultato di un'interrogazione

- `DISTINCT` — eliminare i duplicati
- `GROUP BY` — raggruppare le tuple

L'ordinamento viene eseguito in memoria secondaria tramite l'algoritmo **Sort-Merge**, composto da 2 fasi:

1. **Sort interno:** si legge *una pagina alla volta*, caricando un blocco di memoria centrale alla volta. Tutte le tuple della pagina vengono ordinate tramite **QuickSort** o un algoritmo equivalente. La pagina ordinata risultante viene chiamata **RUN** e viene salvata in un file temporaneo in memoria secondaria.
2. **Merge:** si applicano uno o più passi di **fusione** con cui le RUN vengono unite progressivamente fino a produrre un'*unica RUN ordinata*.

### 14.2.1 Esempio di Sort-Merge a 2 vie ($z = 2$)

**Parametri:**

- $NP = 4$ — numero di pagine della tabella da ordinare
- $NB = 3$ — numero di buffer di memoria centrale disponibili
- Sort-Merge a **2 vie** ($z = 2$): ad ogni passo di fusione si uniscono **2 RUN** alla volta

La tabella da ordinare rispetto alla *prima colonna* è la seguente:

| | |
|---|---|
| $g$ | 24 |
| $a$ | 19 |
| $d$ | 21 |
| $c$ | 23 |
| $b$ | 14 |
| $e$ | 16 |
| $r$ | 16 |
| $d$ | 31 |
| $m$ | 3 |
| $p$ | 2 |
| $d$ | 17 |
| $a$ | 14 |

**Fase 1 — Sort interno:** si carica e ordina una pagina alla volta. Ogni pagina ordinata costituisce una **RUN**, salvata in memoria secondaria.

- **RUN 1:** $a : 19,\ d : 21,\ g : 24$
- **RUN 2:** $b : 14,\ c : 23,\ e : 16$
- **RUN 3:** $d : 31,\ m : 3,\ r : 16$
- **RUN 4:** $a : 14,\ d : 17,\ p : 2$

**Fase 2 — Merge:** si applicano passi di fusione a 2 vie fino a ottenere un'unica RUN ordinata. Il buffer di output ha capacità di una pagina (3 tuple): quando si riempie viene **scaricato in memoria secondaria** e svuotato.

**Passo 1 — fusione a coppie:**

- **RUN 1 + RUN 2** $\longrightarrow \underbrace{a:19,\ b:14,\ c:23}_{P_2},\ \underbrace{d:21,\ e:16,\ g:24}_{P_3}$
- **RUN 3 + RUN 4** $\longrightarrow \underbrace{a:14,\ d:17,\ d:31}_{P'_2},\ \underbrace{m:3,\ p:2,\ r:16}_{P'_3}$

**Passo 2 — fusione finale:**

- **(RUN 1+2) + (RUN 3+4)** $\longrightarrow a:14,\ a:19,\ b:14,\ c:23,\ d:17,\ d:21,\ d:31,\ e:16,\ g:24,\ m:3,\ p:2,\ r:16$

### 14.2.2 Costo del $z$-way Sort-Merge

Il costo è misurato in termini di **accessi alla memoria secondaria** (letture e scritture). Consideriamo il caso $z = 2$, $NB = 3$.

Gli accessi possono essere in **lettura** o **scrittura**.

- **Fase di sort interno:** si leggono $NP(R)$ pagine e si riscrivono $NP(R)$ pagine ordinate in memoria secondaria.
- **Fase di merge:** ad ogni passo si leggono le $NP(R)$ pagine e si riscrivono ordinate. Il numero di passi di merge necessari è:

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

  poiché ad ogni passo il numero di RUN da unire si **dimezza** (nel caso $z = 2$).

Il **costo complessivo** del 2-way Sort-Merge è quindi:

$$\text{Costo} = 2 \cdot NP(R) \cdot (\lceil \log_2 NP(R) \rceil + 1)$$

dove:

- il fattore 2 tiene conto di lettura **e** scrittura
- $\lceil \log_2 NP(R) \rceil$ è il numero di passi di merge
- $+1$ rappresenta la fase di sort interno

## 14.3 Accesso diretto tramite indice

- **Condizione atomica di uguaglianza** $A = v$: richiede la presenza di un **B+Tree** o **Hash**. Il costo dipende dal tipo di indice utilizzato.
- **Selezione con condizione di range** $A \geq v_1$ `AND` $A \leq v_2$: si utilizza un **B+Tree**.
- **Selezione con congiunzione (`AND`) di condizioni di uguaglianza** $A = v_1$ `AND` $B = v_2$: viene scelto l'indice in base alla **condizione più selettiva**. Il DBMS determina autonomamente se utilizzare l'indice su $A$ o su $B$ a seconda di quale condizione riduce maggiormente il numero di tuple restituite.
- **Selezione con disgiunzione (`OR`) di condizioni di uguaglianza** $A = v_1$ `OR` $B = v_2$: si possono utilizzare **più indici in parallelo**, eseguendo successivamente il **merge dei risultati**.

## 14.4 Calcolo del costo di un Join

Il Join è l'operazione più gravosa da eseguire per un DBMS. Gli algoritmi disponibili sono:

- **Nested Loop Join**
- **Merge Scan Join**
- **Hash Based Join**

**Osservazione:** dal punto di vista *logico* il Join è **commutativo**, ma dal punto di vista *fisico* il Join **non è commutativo**: invertire l'ordine degli operandi non cambia il risultato, ma può cambiare significativamente i **costi di esecuzione**.

### 14.4.1 Nested Loop Join

Date due relazioni $R$ e $S$ su cui è definito il predicato di Join $PJ$, si distinguono:

- $R$ — **relazione esterna**
- $S$ — **relazione interna**

L'algoritmo è il seguente:

```
per ogni tupla t_R di R {
 per ogni tupla t_S di S {
  if (t_R, t_S) soddisfa PJ {
   aggiungi (t_R, t_S) al risultato
  }
 }
}
```

**Esempio:** siano date le seguenti relazioni con predicato di Join $PJ : R.A = S.A$.

$R$ :

| $A$ | $B$ |
|---|---|
| 22 | $a$ |
| 87 | $s$ |
| 45 | $h$ |
| 32 | $b$ |

$S$ :

| $A$ | $C$ | $D$ |
|---|---|---|
| 22 | $z$ | 8 |
| 45 | $k$ | 4 |
| 22 | $s$ | 7 |
| 87 | $s$ | 9 |
| 32 | $c$ | 3 |
| 45 | $h$ | 5 |
| 32 | $g$ | 6 |

dove $R$ è composta da 2 pagine $(P_1, P_2)$ e $S$ da 4 pagine $(P_1, P_2, P_3, P_4)$.

L'algoritmo carica la prima tupla di $R$ e scansiona *per intero* $S$ verificando il predicato; poi passa alla tupla successiva di $R$ e ripete la scansione completa di $S$, e così via. Il risultato è:

| $A$ | $B$ | $C$ | $D$ |
|---|---|---|---|
| 22 | $a$ | $z$ | 8 |
| 22 | $a$ | $s$ | 7 |

E poi cosi via per ogni elemento

**Costo del Nested Loop Join** Per calcolare il costo è necessario conoscere la **dimensione del buffer** disponibile.

**Ipotesi semplificativa:** 1 pagina per caricare $R$ e 1 pagina per caricare $S$.

Il costo si compone di due termini:

- Si legge $R$ completamente **una volta**: costo $NP(R)$
- Per ogni riga di $R$ si legge $S$ completamente: costo $NR(R) \cdot NP(S)$

dove $NR(R)$ indica il numero di righe di $R$.

Il **costo complessivo** è quindi:

$$\text{Costo} = NP(R) + NR(R) \cdot NP(S)$$

**Nested Loop Join con Indice** Se è disponibile un **indice** costruito sugli attributi di Join di $S$, la scansione completa di $S$ può essere sostituita da un **accesso tramite indice**.

Per ogni tupla di $R$, invece di scansionare interamente $S$, si accede all'indice su $S.A$ per recuperare *solo i blocchi di $S$ che contengono tuple con $S.A = R.A$.*

**Costo del Nested Loop Join con indice B+Tree su $S.A$:**

$$\text{Costo} = NP(R) + NR(R) \cdot \left(\text{Prof_Indice} + \frac{NR(S)}{VAL(A,S)}\right)$$

dove:

- $NP(R)$ — costo invariato: $R$ deve essere caricata completamente
- $NR(R)$ — invariato: numero di volte in cui si esegue il ciclo esterno
- Prof_Indice — profondità del B+Tree: costo di navigazione dell'indice per ogni tupla di $R$
- $\dfrac{NR(S)}{VAL(A,S)}$ — **selettività** dell'attributo $A$ in $S$: stima del numero di pagine di $S$ da caricare per ogni tupla di $R$

La **selettività** dell'attributo $A$ nella relazione $S$ è definita come:

$$\text{Selettività}(A,S) = \frac{NR(S)}{VAL(A,S)}$$

dove $VAL(A,S)$ è il numero di **valori distinti** di $A$ che compaiono in $S$. **Esempio:** sia $A = \text{NomeRegione}$, con:

- $VAL(A,S) = 21$ — numero di regioni italiane (valori distinti di $A$)
- $NR(S) = 21.000$ — numero di righe della relazione $S$

La selettività risulta:

$$\frac{NR(S)}{VAL(A,S)} = \frac{21.000}{21} = 1.000 \text{ tuple per ogni valore di } A$$

Assumendo una **distribuzione uniforme** dei dati e nel **caso pessimo** (1 tupla per pagina), si devono caricare **1.000 pagine** di $S$ per ogni tupla di $R$. **Esempio:** siano date le relazioni:

- `RICETTA` — $NP = 12$, $NR = 260$
- `COMPOSIZIONE` — $NP = 200$, $NR = 13.000$

con predicato di Join $PJ$ : `RICETTA.CODICE` = `COMPOSIZIONE.RICETTA`.

**Metodo 1 — Nested Loop Join senza indice**
(`RICETTA` = relazione esterna, `COMPOSIZIONE` = relazione interna):

$$NP(\text{Ricetta}) + NR(\text{Ricetta}) \times NP(\text{Composizione}) = 12 + 260 \times 200 = \mathbf{52.012}$$

**Metodo 2 — Nested Loop Join con indice B+Tree su `Composizione.Ricetta`**
(profondità indice = 2, $VAL(\text{Ricetta}, \text{Composizione}) = 260$):

$$NP(\text{Ricetta}) + NR(\text{Ricetta}) \times \left(\text{Prof_Indice} + \frac{NR(\text{Composizione})}{VAL(\text{Ricetta}, \text{Composizione})}\right)$$

$$= 12 + 260 \times \left(2 + \frac{13.000}{260}\right) = 12 + 260 \times 52 = \mathbf{13.532}$$

**Conclusione:** in questo caso conviene il **Metodo 2** (Nested Loop Join con indice B+Tree), poiché $13.532 < 52.012$: l'utilizzo dell'indice riduce significativamente il numero di accessi in memoria secondaria.

# 15 Laboratorio

## 15.1 PostgreSQL

PostgreSQL è un **Relational Database Management System (RDBMS)**, cioè un sistema di gestione di basi di dati relazionali, che integra anche alcune **funzionalità orientate agli oggetti**.

È un software **open source** e **multipiattaforma**, disponibile su diversi sistemi operativi come **Linux, macOS e Windows**.

Il progetto è mantenuto da una comunità internazionale di sviluppatori. Il sito ufficiale è:

`https://www.postgresql.org/`

PostgreSQL rappresenta una **valida alternativa open source** ai principali DBMS proprietari, come quelli sviluppati da **Oracle** e **Microsoft**. L'interazione tra un utente (programmatore o utente finale) e le basi di dati gestite da **PostgreSQL** avviene secondo il modello **client–server**.

- **Client**: è qualsiasi programma in grado di comunicare con il server PostgreSQL. Il client permette di:
  - inserire comandi SQL;
  - inviare le richieste al server;
  - visualizzare i risultati delle interrogazioni.

  Esempi di client sono:
  - **psql**: applicazione a riga di comando per interagire con PostgreSQL;
  - **pgAdmin**: client grafico che offre un'interfaccia più intuitiva per la gestione del database.
- **Server**: è il sistema che gestisce effettivamente il database. Il server è controllato dal processo principale chiamato **postmaster**, che:
  - gestisce le connessioni dei client;
  - coordina i processi del database;
  - esegue le operazioni richieste.

Nel modello client–server il client invia una **request** (richiesta) al server, ad esempio un comando SQL. Il server elabora la richiesta e restituisce al client una **response** (risposta) contenente il risultato dell'operazione. **SQL** è il linguaggio di riferimento per la gestione delle **basi di dati relazionali**.

Originariamente SQL era l'acronimo di *Structured Query Language*; oggi è considerato semplicemente un **nome proprio**.

SQL è un linguaggio che comprende diverse categorie di operazioni:

- **Data Definition Language (DDL)** Serve per definire la struttura della base di dati e i vincoli di integrità. Esempi di operazioni: creazione, modifica ed eliminazione di tabelle e schemi.
- **Data Manipulation Language (DML)** Serve per manipolare i dati presenti nella base di dati. Include operazioni di:

  - inserimento dei dati
  - aggiornamento dei dati
  - cancellazione dei dati

- **Data Query Language (DQL o QL)** Serve per interrogare la base di dati e recuperare le informazioni desiderate tramite query.

Nel tempo sono state definite **diverse versioni dello standard SQL**. SQL permette di svolgere diverse operazioni sulle basi di dati relazionali:

- **Definizione dello schema della base di dati** Permette di creare le strutture che compongono il database, come tabelle, attributi e vincoli.
- **Modifica dello schema della base di dati** Consente di modificare la struttura esistente del database, ad esempio aggiungendo o rimuovendo attributi o tabelle.
- **Inserimento, modifica e cancellazione dei dati** Permette di manipolare i dati contenuti nel database tramite operazioni di inserimento, aggiornamento e cancellazione.
- **Interrogazioni semplici sulla base di dati** Consente di recuperare informazioni dal database attraverso query di selezione.
- **Interrogazioni complesse** Permette di eseguire query avanzate che includono, ad esempio, **query annidate** e **raggruppamenti di dati**.

## 15.2 SQL comandi base

### 15.2.1 Creazione Tabella

L'istruzione per la definizione di una relazione consente di:

- definire lo **schema della relazione**;
- creare un'**istanza iniziale vuota** della relazione;
- specificare **attributi**, **domini** e **vincoli**.

Nella sintassi utilizzata per descrivere i comandi SQL vengono adottate alcune convenzioni:

- [ ] indicano **parti opzionali** della sintassi. L'elemento racchiuso tra parentesi quadre può comparire **zero oppure una sola volta**. Sta al programmatore decidere se inserirlo o meno in base alle necessità.
- [**valoreDefault**] indica che, durante la definizione di un attributo, è possibile specificare un **valore predefinito** che verrà assegnato automaticamente se non viene fornito un valore.
- **{ }** indicano elementi che possono essere **ripetuti zero, una o più volte**.
- **{vincolo}** indica che a una singola colonna possono essere associati **più vincoli consecutivi** oppure nessuno. Ad esempio: `NOT NULL`, `UNIQUE`, ecc.

La sintassi generale per la creazione di una tabella è:

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

```
CREATE TABLE Impiegato (
     Matricola CHARCTER(6) PRIMARY KEY,          -- attributo
     Nome CHARCTER VARYING(20) NOT NULL,         -- attributo
     Cognome CHARCTER VARYING(20) NOT NULL,      -- attributo
     Dipart CHARCTER VARYING(15),                -- attributo
     Stipendio NUMERIC(9) DEFAULT 0,             -- attributo
     FOREIGN KEY(Dipart)
       REFERENCES Dipartimento(NomeDip),      -- vincolo di tabella
     UNIQUE (Cognome,Nome)                -- vincolo di tabella
 )
```

Figura 137: Esempio creazione tabella

### 15.2.2 Concetti fondamentali del DDL

Già con l'istruzione `CREATE TABLE` è necessario conoscere alcuni concetti fondamentali del **Data Definition Language (DDL)**:

- **Identificatori**
- **Domini** (tipi di dato)
- **Vincoli**

#### Gli Identificatori

Gli **identificatori SQL** sono stringhe utilizzate per assegnare un nome agli elementi del database, come tabelle, attributi, vincoli, ecc.

- Un identificatore deve iniziare con una **lettera (UTF-8)** oppure con un **underscore (_)**.
- Può contenere anche **cifre** (`0-9`) e altri caratteri validi.
- Gli identificatori SQL **non sono sensibili alle maiuscole e minuscole**.

Ad esempio:

```
idPersona
idpersona
```

rappresentano lo **stesso identificatore**.

Tutti i nomi utilizzati nel database, come **nomi di tabelle, attributi, indici o vincoli**, sono identificatori SQL.

#### Domini

Un **dominio** definisce il **tipo di valori** che un attributo può assumere all'interno di una relazione.

SQL mette a disposizione diversi **domini elementari predefiniti**, ma consente anche di definire **domini personalizzati**.

### Caratteri

Permettono di rappresentare **singoli caratteri oppure stringhe**.

Un carattere o una stringa di caratteri viene rappresentato tra apici:

```
'a'
'Questa è una stringa'
```

La lunghezza delle stringhe può essere **fissa** o **variabile**:

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

- **CHARACTER**: rappresenta un carattere singolo.
- **CHARACTER(20)**: stringa di lunghezza fissa. Se si assegna una stringa di lunghezza inferiore a 20, la stringa viene riempita con spazi fino a raggiungere la lunghezza specificata.
- **CHARACTER VARYING(20)**: stringa di lunghezza variabile con lunghezza massima pari a 20.
- **TEXT**: stringa di lunghezza variabile senza limite prefissato. Questo tipo è una **estensione specifica di PostgreSQL**.

### Boolean

Permettono di rappresentare **valori logici**:

```
BOOLEAN
```

I valori possibili sono:

- **TRUE**
- **FALSE**
- **NULL** (valore sconosciuto)

Possono essere rappresentati anche con forme equivalenti come:

```
TRUE / FALSE
t / f
1 / 0
yes / no
```

### Numerici esatti

Permettono di rappresentare **valori numerici esatti**, interi oppure con parte decimale di lunghezza prefissata.

#### Valori interi

- **SMALLINT** Valori interi su 2 byte: da $-32768$ a $+32767$.
- **INTEGER** Valori interi su 4 byte: da $-2147483648$ a $+2147483647$.

#### Valori decimali

- **NUMERIC [(precisione [, scala])]**
  - precisione: numero totale di cifre significative;

  - scala: numero di cifre dopo la virgola.

- **DECIMAL [(precisione [, scala])]** Equivalente a `NUMERIC`.

Esempio:

```
NUMERIC(5,2)
```

Permette valori come:

```
100.01
100.999 -> arrotondato a 101.00
```

ma non valori come:

```
1000.01
```

### Numerici approssimati

Permettono di rappresentare numeri in **virgola mobile**.

- **REAL**: precisione di circa 6 cifre decimali.
- **DOUBLE PRECISION**: precisione di circa 15 cifre decimali.

Ogni numero è rappresentato tramite:

- **mantissa** (numero frazionario)
- **esponente** (numero intero)

Il valore si ottiene moltiplicando la mantissa per $10^{esponente}$.

Esempi:

```
0.17E16 = 1.7 x 10^15
-0.4E-6 = -4 x 10^-7
```

**Nota importante**

- `REAL` e `DOUBLE PRECISION` sono valori **approssimati**.
- Se si devono rappresentare **importi di denaro**, non bisogna usare questi tipi, ma utilizzare:

  ```
  NUMERIC(precisione)
  ```

### Istanti temporali

Permettono di rappresentare **momenti specifici nel tempo**.

- **DATE**

  Rappresenta istanti nel formato:

  ```
  YYYY-MM-DD
  ```

  Esempio:

  ```
  '2025-03-10'
  ```

- **TIME [(precisione)] [WITH TIME ZONE]**

Rappresenta istanti del tipo:

```
hour : minute : second
```

  - **precisione**: numero di cifre decimali per le frazioni di secondo
  - **WITH TIME ZONE**: permette di specificare il fuso orario

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

  Rappresenta una combinazione di:

  - data
  - ora

### Intervalli temporali

Permettono di rappresentare **durate o intervalli di tempo**.

Sintassi generale:

```
INTERVAL PrimaUnitàDiTempo [TO UltimaUnitàDiTempo]
```

Le unità temporali sono divise in due gruppi:

- **YEAR - MONTH**
- **DAY - HOUR - MINUTE - SECOND**

Esempi:

```
INTERVAL YEAR
INTERVAL MONTH
INTERVAL YEAR TO MONTH
INTERVAL DAY
INTERVAL DAY TO HOUR
INTERVAL DAY TO MINUTE
INTERVAL DAY TO SECOND
```

### Domini definiti dall'utente

SQL consente di definire **nuovi domini personalizzati** basati su domini esistenti.

Sintassi:

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

Il dominio definito può essere utilizzato nelle definizioni delle relazioni e può includere:

- valori di default
- vincoli

Esempio:

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

In questo esempio viene definito un dominio che rappresenta i giorni della settimana tramite una stringa di tre caratteri.

## 15.3 Vincoli intrarelazionali

I **vincoli intrarelazionali** sono vincoli che si applicano agli attributi di una singola relazione e permettono di garantire la correttezza dei dati.

I principali vincoli sono:

- **NOT NULL**
- **UNIQUE**
- **PRIMARY KEY**
- **CHECK**

### NOT NULL e DEFAULT

Il vincolo **NOT NULL** impone che il valore dell'attributo non possa essere nullo.

Questo significa che il valore deve sempre essere specificato, oppure deve essere presente un valore di default.

Il vincolo **DEFAULT** permette di specificare un valore predefinito quando il comando di inserimento non fornisce alcun valore per quell'attributo.

Esempio:

```
nome VARCHAR(20) NOT NULL
cognome VARCHAR(20) NOT NULL DEFAULT 'VUOTO'
```

### UNIQUE

Il vincolo **UNIQUE** impone che i valori di un attributo (o di un insieme di attributi) costituiscano una **superchiave**. Di conseguenza, due tuple della tabella non possono avere gli stessi valori per quegli attributi.

Può essere definito su:

- un singolo attributo;
- un insieme di attributi.

Esempio con vincolo su più attributi:

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

In questo caso non possono esistere due righe con lo stesso `Nome` e `Cognome` contemporaneamente.

Esempio con vincoli separati:

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

In questo caso non possono esistere due righe con lo stesso `Nome` oppure con lo stesso `Cognome`.

### PRIMARY KEY

Il vincolo **PRIMARY KEY** identifica l'attributo (o l'insieme di attributi) che rappresenta la **chiave primaria** della relazione.

Caratteristiche:

- può essere definito una sola volta per tabella;
- implica automaticamente il vincolo **NOT NULL**.

Esistono due modalità di definizione.

1. **Nella definizione dell'attributo** (chiave primaria singola):

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

2. **Come vincolo di tabella** (chiave primaria composta):

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

### CHECK

Il vincolo **CHECK** permette di specificare una condizione che deve essere soddisfatta dai valori della tabella.

Un vincolo CHECK è soddisfatto se la sua espressione è:

- **vera**
- oppure **NULL**

Esempio:

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

In questo esempio:

- `CHECK(stipendio >= 0.0)` è un **check di attributo**;
- `CHECK(nome <> cognome)` è un **check di tabella**.

## 15.4 Vincoli interrelazionali

Un **vincolo di integrità referenziale** (o **vincolo interrelazionale**) crea un legame tra i valori di un attributo (o di un insieme di attributi) di una tabella e i valori di un attributo (o di un insieme di attributi) di un'altra tabella.

In particolare:

- Il vincolo collega un attributo (o insieme di attributi) $A$ della **tabella corrente** (detta **interna** o **slave**)
- con un attributo (o insieme di attributi) $B$ di un'altra tabella (detta **esterna** o **master**)

Il vincolo impone che:

- per ogni tupla della tabella interna, il valore di $A$, se diverso da `NULL`, deve essere presente tra i valori di $B$ nella tabella esterna;
- l'attributo $B$ della tabella esterna deve essere soggetto a un vincolo **UNIQUE** oppure **PRIMARY KEY**;
- l'attributo (o insieme di attributi) $B$ non deve necessariamente essere la chiave primaria, ma deve comunque identificare in modo univoco le tuple della tabella esterna.

### FOREIGN KEY e REFERENCES

I costrutti **REFERENCES** e **FOREIGN KEY** permettono di definire vincoli di integrità referenziale.

Esistono due modalità di definizione.

#### Vincolo su un singolo attributo

Se il vincolo coinvolge un solo attributo, è possibile utilizzare direttamente `REFERENCES` nella dichiarazione dell'attributo, specificando la tabella esterna e l'attributo corrispondente.

Esempio:

```
CREATE TABLE Impiegato (
matricola CHAR(6) PRIMARY KEY,
reparto INTEGER REFERENCES Reparto(idReparto)
);
```

In questo caso l'attributo `reparto` della tabella `Impiegato` fa riferimento all'attributo `idReparto` della tabella `Reparto`.

#### Vincolo su più attributi

Quando il vincolo coinvolge più attributi, oppure si preferisce dichiararlo separatamente, si utilizza il costrutto **FOREIGN KEY** alla fine della definizione degli attributi.

Si elencano gli attributi della tabella interna e si specificano gli attributi corrispondenti della tabella esterna tramite `REFERENCES`.

Esempio:

```
CREATE TABLE Iscrizione (
matricola CHAR(6),
codiceCorso CHAR(5),
FOREIGN KEY (matricola)
REFERENCES Studente(matricola)
);
```

In questo modo si definisce un vincolo di integrità referenziale tra le due tabelle.

Tabella Interna

```
CREATE TABLE Impiegato (
   Matricola   CHARACTER(6) primary key,
   Nome        CHARACTER VARYING(20)  not null,
   Cognome     CHARACTER VARYING(20)  not null,
   Dipart      CHARACTER VARYING(15)
   REFERENCES Dipartimento(NomeDip),
   Ufficio     NUMERIC(3),
   Stipendio   NUMERIC(9) default 0,
   UNIQUE (Cognome, Nome)
)
```

Tabella Esterna

Attributo chiave

Figura 138: References e Foreign key

Tabella Interna

```
CREATE TABLE Impiegato (
   Matricola   CHARACTER(6) primary key,
   Nome        CHARACTER VARYING(20)  not null,
   Cognome     CHARACTER VARYING(20)  not null,
   Dipart      CHARACTER VARYING(15)
   REFERENCES Dipartimento(NomeDip),
   Ufficio     NUMERIC(3),
   Stipendio   NUMERIC(9) default 0,
   FOREIGN KEY (Nome, Cognome)
   REFERENCES Anagrafica (Nome, Cognome)
)
```

Tabella Esterna

Attributi chiave

Figura 139:

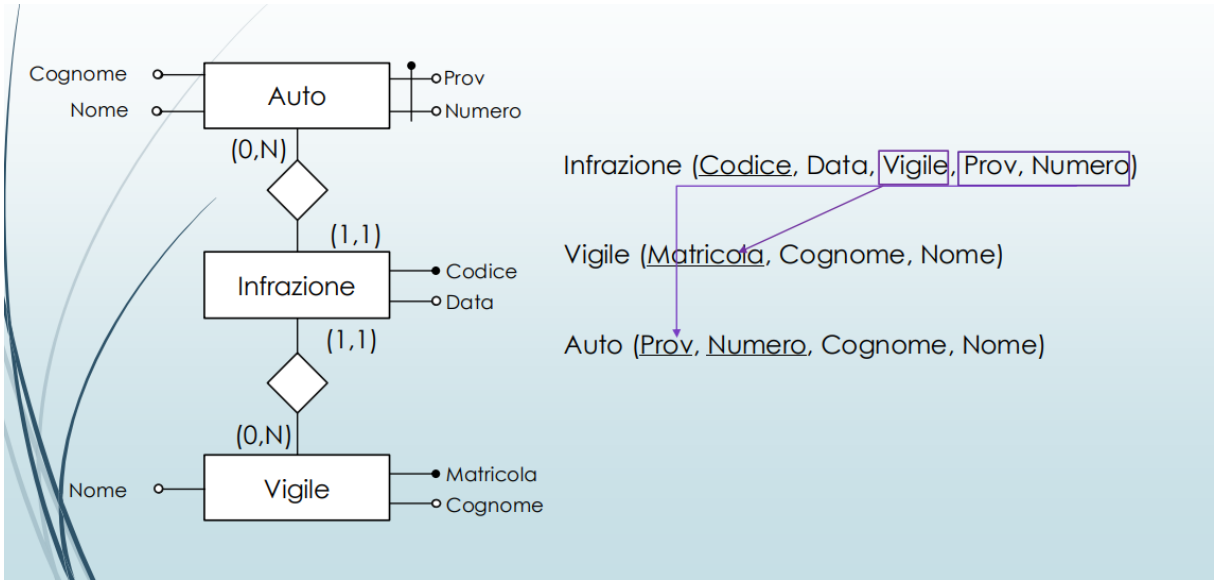



Figura 140: Esempio

```
CREATE TABLE Infrazione(
   Codice CHAR(6) PRIMARY KEY,
   Data DATE NOT NULL,
   Vigile INTEGER NOT NULL
      REFERENCES Vigile(Matricola),
   Provincia CHAR(2),
   Numero CHAR(6) ,
   FOREIGN KEY(Provincia, Numero)
      REFERENCES Auto(Provincia, Numero)
 )
```

Figura 141: Risoluzione Esempio

## 15.5 Modifica ed eliminazione delle strutture

SQL mette a disposizione diversi comandi per modificare o eliminare gli elementi di una base di dati, come domini, tabelle e attributi.

- **ALTER**: permette di effettuare modifiche alla struttura di domini e tabelle (ad esempio aggiungere o modificare colonne).

- **DROP DOMAIN**: permette di rimuovere un dominio.
- **DROP TABLE**: permette di eliminare una tabella dal database.

### Aggiunta di un attributo

Un nuovo attributo può essere aggiunto a una tabella tramite il comando `ADD COLUMN`.

Sintassi generale:

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

Esempio:

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

### Rimozione di un attributo

Un attributo può essere rimosso da una tabella tramite il comando `DROP COLUMN`.

Sintassi generale:

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

Esempio:

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

### Modifica del valore di default

È possibile modificare o rimuovere il valore di default di un attributo tramite il comando `ALTER COLUMN`.

Sintassi:

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

Esempio:

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

## 15.6 Cancellazione dei dati

Le tuple di una tabella possono essere eliminate tramite il comando `DELETE`.

Sintassi generale:

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

- `condizione` è un'espressione booleana che seleziona le tuple da cancellare.
- Se la clausola `WHERE` non è presente, **tutte le tuple della tabella vengono cancellate**.

Esempio:

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

### 15.6.1 Eliminazione di una tabella

Una tabella può essere eliminata completamente dal database tramite il comando `DROP TABLE`.

Sintassi:

```
DROP TABLE tabella;
```

## 15.7 Linguaggio di Interrogazione SQL

**SQL** (Structured Query Language) è il linguaggio di riferimento per la gestione delle **basi di dati relazionali**. In origine l'acronimo indicava *Structured Query Language*, mentre oggi viene considerato semplicemente un **nome proprio**.

SQL è un linguaggio molto completo che permette di svolgere diverse operazioni su una base di dati.

SQL consente di:

- Definire la struttura di una base di dati
- Modificare lo schema della base di dati
- Inserire, modificare e cancellare i dati
- Eseguire interrogazioni semplici
- Eseguire interrogazioni complesse (interrogazioni annidate, raggruppamenti, ecc.)

Per questo motivo SQL è diviso in diversi sottolinguaggi.

- **DDL (Data Definition Language)**: permette di definire la struttura della base di dati e i vincoli di integrità.
- **DML (Data Manipulation Language)**: permette di inserire, modificare o cancellare i dati.
- **DQL (Data Query Language)**: permette di interrogare la base di dati tramite query.

SQL è un **linguaggio dichiarativo**. Questo significa che l'utente specifica **cosa vuole ottenere**, ma non **come** il sistema deve eseguire l'operazione.

Ad esempio, quando si scrive una query SQL, si indica semplicemente il risultato desiderato.

```
SELECT nome FROM Studenti WHERE voto > 25
```

Il modo in cui la query viene eseguita è deciso automaticamente dal **DBMS**.

Una componente del DBMS chiamata **query optimizer** traduce la query SQL in operazioni interne più efficienti. Questo processo è nascosto all'utente e permette di esprimere interrogazioni in modo **astratto e ad alto livello**.

Nel tempo sono state definite diverse versioni dello standard SQL.

- **SQL-86**: prima standardizzazione del linguaggio
- **SQL-89**: introduzione dell'integrità referenziale
- **SQL-92 (SQL-2)**: estensione significativa del linguaggio
- **SQL:1999 (SQL-3)**: introduzione di concetti orientati agli oggetti, trigger e funzioni
- Versioni successive (2003, 2006, 2008, 2011, 2016): introduzione di nuove funzionalità come supporto XML, JSON e dati temporali

Le varie versioni dello standard hanno progressivamente ampliato le capacità del linguaggio, mantenendo comunque compatibilità con le versioni precedenti.

## 15.8 Clausola SELECT

**Sintassi SELECT semplificata**

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

Figura 142: Sintassi SELECT

- **expression** è un'espressione che determina un attributo.
- **from_item** è un'espressione che determina una sorgente per gli attributi.
- **condition** è un'espressione booleana utilizzata per selezionare i valori degli attributi.
- **grouping_element** è un'espressione che permette di eseguire operazioni su più valori di un attributo e considerare il risultato complessivo.

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

Figura 143: Forma semplificata SELECT

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

**Sintassi SELECT semplificata**

```
SELECT [ DISTINCT ]
[{* | expression [[ AS] output_name ]} [, ...]]
```

Figura 144: DISTINCT

- ***** è un'abbreviazione utilizzata per indicare tutti gli attributi delle tabelle.

- **expression** è un'espressione che coinvolge gli attributi della tabella (può anche essere semplicemente il nome di un attributo).
- **output_name** è il nome assegnato all'attributo che conterrà il risultato della valutazione dell'espressione **expression** nella relazione risultato.
- **DISTINCT**: se presente richiede l'eliminazione delle tuple duplicate.

- Q1: Estrarre lo stipendio degli impiegati di cognome 'Rossi'

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

Impiegato

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

| Stipendio |
|---|
| 45 |
| 80 |

Figura 145: Esempio : Estrarre lo stipendio degli impiegati di cognome "Rossi"

SELECT Stipendio as Salario ← alias
FROM Impiegato
WHERE Cognome='Rossi'

| Salario |
|---|
| 45 |
| 80 |

- Restituisce lo stesso risultato della query precedente, ma ridenomina (temporaneamente) la colonna output Stipendio

Figura 146: Uso di AS (ridenominazione)

```
SELECT *
FROM Impiegato
WHERE Cognome='Rossi'
```

Seleziona tutti gli attributi delle tabella che compaiono nella clausola FROM

- L'uso dell'asterisco mi permette di non elencare il nome di tutte le colonne (è come se prioettassi su tutti gli attributi)

| Nome | Cognome | Dipart | Ufficio | Stipendio | Città |
|---|---|---|---|---|---|
| Mario | Rossi | Amministrazione | 10 | 45 | Milano |
| Carlo | Rossi | Direzione | 14 | 80 | Milano |

Figura 147: Uso dell'asterisco

## 15.9 Clausola FROM

**Sintassi**

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

Figura 148: Sintassi FROM

dove **from_item** può essere una delle seguenti clausole (versione semplificata):

- **table_name [[AS] alias [(column_alias [, ...])]]**

  Se sono presenti due o più tabelle, si esegue il **prodotto cartesiano** tra tutte le tabelle. Se ci sono attributi con lo stesso nome su tabelle diverse, nella clausola **SELECT** questi attributi sono identificati anteponendo il nome della tabella e un punto:

  `nomeTabella.nomeAttributo`

- **(other_select) [AS] nomeRisultato [(column_alias [, ...])]**

  È una **SELECT innestata**. Il risultato della SELECT interna viene trattato come una tabella temporanea con nome **nomeRisultato**, sulla quale viene eseguita la SELECT esterna.

- **from_item [NATURAL] join_type from_item [ON condition]**

  È la clausola **JOIN**.

  Rappresenta un metodo più sofisticato rispetto al caso 1), che permette di selezionare un sottoinsieme del prodotto cartesiano di due o più tabelle. Il parametro **join_type** specifica il tipo di join da utilizzare. I principali tipi di join sono descritti nelle slide successive.

- Q2: Estrarre i nomi ed il cognome degli impiegati e le città in cui lavorano

- SELECT Impiegato.Nome, Impiegato.Cognome,
  Dipartimento.Città
  FROM Impiegato, Dipartimento
  WHERE Impiegato.Dipart = Dipartimento.Nome

Dipartimento

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

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

Impiegato

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

Figura 149: Estrarre i nomi ed il cognome degli impiegati e le città in cui lavorano

- Q2: Estrarre i nomi ed il cognome degli impiegati e le città in cui lavorano

uso dell'alias per abbreviare il nome delle tabelle

- SELECT i.Nome, i.Cognome,
  d.Città
  FROM Impiegato AS i, Dipartimento AS d
  WHERE i.Dipart = d.Nome

Dipartimento

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

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

Impiegato

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

Figura 150: Stessa cosa di prima ma con AS

## 15.10 Clausola WHERE

**Sintassi**

```
WHERE condition
```

Figura 151: Sintassi WHERE

- La clausola **WHERE** ammette come argomento un'espressione booleana costruita combinando predicati semplici con gli operatori **AND**, **OR** e **NOT**.
- **AND**: seleziona le righe in cui tutti i predicati sono veri.
- **OR**: seleziona le righe in cui almeno un predicato è vero.
- **NOT**: seleziona le righe in cui il predicato è falso.

- Ciascun predicato semplice utilizza gli operatori $=$, $<>$, $<$, $>$, $\leq$ e $\geq$ per confrontare un'espressione costruita a partire dal valore di un attributo con un valore costante oppure con un'altra espressione.
- Per esprimere la precedenza tra gli operatori **AND** e **OR** si utilizzano le **parentesi**.

- Q3: Estrarre i nomi ed il cognome degli impiegati che lavorano nell'ufficio 20 del dipartimento Amministrazione.

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

Dipartimento

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

Impiegato

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

| Nome | Cognome |
|---|---|
| Giovanni | Verdi |

Figura 152: Estrarre i nomi ed il cognome degli impiegati che lavorano nell'ufficio 20 del dipartimento Amministrazione

- Q4: Estrarre i nomi ed il cognome degli impiegati che lavorano dipartimento Amministrazione o nel dipartimento Produzione.

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

Dipartimento

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

Impiegato

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

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

Figura 153: Estrarre i nomi ed il cognome degli impiegati che lavorano dipartimento Amministrazione o nel dipartimento Produzione.

- Q5: Estrarre i nomi degli impiegati di cognome Rossi che lavorano dipartimento Amministrazione o nel dipartimento Produzione.

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

| Nome |
|---|
| Mario |

Dipartimento

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

Impiegato

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

Figura 154: Estrarre i nomi degli impiegati di cognome Rossi che lavorano dipartimento Amministrazione o nel dipartimento Produzione.

### 15.10.1 Operatore LIKE

- Per il confronto tra stringhe, SQL mette a disposizione anche l'operatore **LIKE**, che consente di verificare se una stringa rispetta un certo **pattern** (oppure **ILIKE** per confronti *case insensitive*).
- Due caratteri speciali utilizzati nei pattern sono _ e %:
  - _: rappresenta un singolo carattere arbitrario.
  - %: rappresenta una sequenza di lunghezza arbitraria (anche nulla) di caratteri arbitrari.
- Ad esempio:

  `_o%i`

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

- Q6: Estrarre gli impiegati che hanno un cognomen cha ha una “o” in seconda posizione e finisce con una “I”.
- SELECT *
  FROM Impiegato
  WHERE Cognome LIKE ‘_o%i’;

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

Dipartimento

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

Impiegato

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

Figura 155: Estrarre gli impiegati che hanno un cognomen cha ha una “o” in seconda posizione e finisce con una “I”.

### 15.10.2 Operatore SIMILAR TO

L’operatore **SIMILAR TO** è una versione più espressiva dell’operatore **LIKE**. Permette di confrontare stringhe utilizzando un **pattern** basato su un sottoinsieme delle **espressioni regolari POSIX** nella versione prevista dallo standard SQL.

Questo operatore consente quindi di effettuare confronti più avanzati rispetto a `LIKE`, utilizzando operatori tipici delle espressioni regolari.

- `_` : rappresenta **un singolo carattere qualsiasi**.
- `%` : rappresenta **una sequenza di zero o più caratteri qualsiasi**.
- `*` : indica la **ripetizione del match precedente zero o più volte**.
- `+` : indica la **ripetizione del match precedente una o più volte**.
- `{n,m}` : indica la **ripetizione del match precedente almeno** $n$ **volte e al massimo** $m$ **volte**.
- `[...]` : rappresenta un **insieme di caratteri ammessi**; il match è valido se il carattere appartiene all’insieme specificato.

Esempio

Studenti con cognome che inizia con 'A' o 'B', o 'D', o 'N' e finisce con 'a':

```
SELECT cognome, nome, città FROM Studente
WHERE cognome SIMILAR TO '[ABDN]{1}%a';
```

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

Figura 156: Esempio SIMILAR TO

### 15.10.3 Operatore BETWEEN

L'operatore **BETWEEN** permette di verificare se il valore di un attributo è compreso tra due valori estremi.

Esempio

Tutti gli studenti che hanno matricola tra 'IN0002' e 'IN0004'.

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

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

Figura 157: Esempio Between

**Esempio**

Tutti gli studenti che hanno matricola tra 'IN0002' e 'IN0004'.

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

| matricola | cognome | nome | indirizzo | città | media |
|---|---|---|---|---|---|
| IN0001 | Ros | | | na | 27.50 |
| IN0002 | Pa | | | va | 28.6666 |
| IN0003 | Bian | | | NA | 18.345 |
| IN0004 | Ros | | | na | 24.567 |
| IN0005 | Ver | | | | 28.6666 |
| IN0006 | Nocasa | Caio | | | |

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

Figura 158: Esempio 2 di Between

### 15.10.4 Operatore IN

Nella clausola **WHERE** può essere utilizzato l'operatore **[NOT] IN** per verificare se il valore di un attributo appartiene ad un insieme di valori.

**Sintassi**

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

Figura 159: Sintassi IN

**Esempio**

Tutti gli studenti che hanno matricola nell'elenco 'IN0001', 'IN0003' e 'IN0005'.

```
SELECT cognome, nome, matricola FROM Studente
WHERE matricola IN('IN0001 ','IN0003 ','IN0005 ');
```

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

Figura 160: Esempio Operatore IN

### 15.10.5 Operatore IS NULL

Per verificare se un attributo ha un valore nullo oppure no si utilizza il predicato **IS NULL**.

- `attributo IS NULL`: restituisce **true** se l'attributo ha valore nullo.
- `attributo IS NOT NULL`: è la negazione del predicato precedente e restituisce **true** se l'attributo ha un valore diverso da `NULL`.

Sintassi

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

Figura 161: Sintassi IS NULL

Esempio

Tutti gli studenti che NON hanno una città.

```
SELECT cognome, nome, città FROM Studente
WHERE città IS NULL;
```

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

Figura 162: Esempio IS NULL

## 15.11 SQL e Algebra Relazionale

Vediamo ora alcuni esempi di interrogazioni in **SQL** e analizziamo la loro corrispondenza con le equivalenti interrogazioni procedurali espresse tramite l'**algebra relazionale**.

L'obiettivo è comprendere come le interrogazioni dichiarative scritte in SQL possano essere interpretate come una sequenza di operazioni dell'algebra relazionale, che descrivono in modo procedurale i passi necessari per ottenere il risultato desiderato.

- SELECT $T_1$.attributo$_{1l}$, ..., $T_h$.attributo$_{hm}$
- FROM Tabella$_1$ as $T_1$, ..., Tabella$_h$ as $T_h$
- WHERE Condizione

$$\Pi_{T1.attributo1l, ..., Th.attributohm} (\sigma_{Condizione} (T1 \bowtie ... \bowtie Th))$$

Figura 163: Sintassi algebra relazione e sql

Nel caso di condizioni più complesse, la formula di conversione tra una interrogazione **SQL** e la corrispondente espressione in **algebra relazionale** non è sempre direttamente rappresentabile.

Una condizione fondamentale per poter effettuare questa conversione è che l'interrogazione SQL utilizzi solo operatori che esistono anche nell'algebra relazionale.

Se l'interrogazione SQL utilizza operatori non presenti nell'algebra relazionale (ad esempio **operatori aggregati** come `COUNT`, `SUM`, `AVG`, ecc.), la conversione diretta non è possibile.

Maternità

| Madre | Figlio |
|---|---|
| Luisa | Maria |
| Luisa | Luigi |
| Anna | Olga |
| Anna | Filippo |
| Maria | Andrea |
| Maria | Aldo |

Paternità

| Padre | Figlio |
|---|---|
| Sergio | Franco |
| Luigi | Olga |
| Luigi | Filippo |
| Franco | Andrea |
| Franco | Aldo |

Persone

| Nome | Età | Reddito |
|---|---|---|
| Andrea | 27 | 21 |
| Aldo | 25 | 15 |
| Maria | 55 | 42 |
| Anna | 50 | 35 |
| Filippo | 26 | 30 |
| Luigi | 50 | 40 |
| Franco | 60 | 20 |
| Olga | 30 | 41 |
| Sergio | 85 | 35 |
| Luisa | 75 | 87 |

Figura 164: Tabella usata per gli esempi

## 15.12 SELECT

- Nome e reddito delle persone con meno di trenta anni
  - $\Pi_{\text{Nome, Reddito}}(\sigma_{\text{Età}<30}(\text{Persone}))$
  - SELECT Nome, Reddito
    FROM Persone
    WHERE età < '30'

Figura 165:

### 15.12.1 SELECT : alias di tabella o di attributo

**ALIAS di Tabella**

- SELECT nome, reddito
  FROM persone
  WHRE Età < '30'

- SELECT p.nome, p.reddito
  FROM persone as p
  WHERE p.Età < '30'

**ALIAS di attributo**

- SELECT nome, reddito
  FROM persone
  WHRE Età < '30'

- SELECT p.nome as nomePersona,
  p.reddito as redditoPersona
  FROM persone as p
  WHERE p.Età < '30'

Figura 166: SELECT : alias di tabella o di attributo

### 15.12.2 SELECT: senza proiezione

- Nome, reddito ed età delle persone con meno di trenta anni
  - $\sigma_{\text{Età}<30}(\text{Persone})$
  - SELECT *
    FROM Persone
    WHERE età < '30'

Figura 167: SELECT senza proiezione

### 15.12.3 Proiezione:gestione dei duplicati in SQL

Una differenza significativa tra **SQL** e l'**algebra relazionale** riguarda la gestione dei duplicati.

Nell'algebra relazionale i risultati delle operazioni sono considerati **insiemi** e quindi non possono contenere duplicati. In SQL, invece, i risultati sono trattati come **multinsiemi** (*bag*), per cui le righe duplicate possono essere presenti.

L'eliminazione delle righe duplicate è un'operazione computazionalmente costosa; per questo motivo, **SQL mantiene i duplicati per default**.

Il costrutto `DISTINCT`, utilizzato dopo la parola chiave `SELECT`, permette di eliminare le righe duplicate dal risultato.

Il costrutto `ALL` permette invece di mantenere esplicitamente i duplicati, anche se questa è già la scelta predefinita del linguaggio SQL.

Impiegati

| Cognome | Nome | Filiale |
|---|---|---|
| Neri | Mario | Napoli |
| Neri | Giorgio | Napoli |
| Rossi | Claudio | Roma |

- Trovare cognome e filiale di tutti gli impiegati

{(Neri, Napoli), (Rossi, Roma)}

- $\Pi_{\text{Cognome, Filiale}}$ (Impiegati)

Figura 168: Proiezione: gestione dei duplicati in SQL

Impiegati

| Cognome | Filiale |
|---|---|
| Neri | Napoli |
| Neri | Napoli |
| Rossi | Roma |

- SELECT Cognome, Filiale
  FROM Impiegati

Impiegati

| Cognome | Filiale |
|---|---|
| Neri | Napoli |
| Rossi | Roma |

- SELECT DISTINCT Cognome, Filiale
  FROM Impiegati

Figura 169:

- Q7: Estrarre le città delle persone il cui cognome è Rossi

- SELECT Città
  FROM Persona
  WHERE Cognome = 'Rossi'

| Città |
|---|
| Verona |
| Verona |
| Milano |

- SELECT DISTINCT Città
  FROM Persona
  WHERE Cognome = 'Rossi'

| Città |
|---|
| Verona |
| Milano |

Persona

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

Figura 170: Esempio : gestione dei duplicati in SQL

### 15.12.4 SELECT: Selezione,Proiezione,Join

L'istruzione **SELECT** permette di esprimere diverse operazioni dell'**algebra relazionale**, a seconda della struttura dell'interrogazione.

- Con **una sola relazione** nella clausola `FROM` è possibile realizzare:
  - **selezioni** ($\sigma$)
  - **proiezioni** ($\pi$)
  - **ridenominazioni** ($\rho$), utilizzando il costrutto `AS`
- Con **più relazioni** nella clausola `FROM` si possono realizzare:
  - **join** ($\bowtie$)
  - **prodotti cartesiani**

Le principali corrispondenze tra le clausole SQL e le operazioni dell'algebra relazionale sono:

- `SELECT` $\rightarrow$ $\pi$ **Proiezione** + $\rho$ **Ridenominazione**
- `FROM` $\rightarrow$ $\bowtie$ **Join** (prodotto cartesiano)
- `WHERE` $\rightarrow$ $\sigma$ **Selezione** + $\bowtie$ **Join** (condizione)

## 15.13 Interrogazioni con JOIN

In **SQL** il **join** può essere definito in due modi diversi.

- Il primo metodo non richiede l'uso di una nuova sintassi: il legame tra le tabelle coinvolte viene specificato nella clausola `WHERE`.
- Il secondo metodo utilizza invece una sintassi dedicata all'interno della clausola `FROM`. In questo caso il join viene espresso esplicitamente tramite gli operatori di join.

Nel seguito verrà utilizzato il secondo approccio. La clausola **FROM** ammette come argomento:

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

Se nella clausola `FROM` sono presenti due o più nomi di tabelle, viene eseguito il **prodotto cartesiano** tra tutte le tabelle. Lo schema del risultato può quindi contenere tutti gli attributi derivanti dal prodotto cartesiano delle relazioni coinvolte.

Il prodotto cartesiano tra due o più tabelle è chiamato **CROSS JOIN**.

A partire dallo standard **SQL-2**, sono stati introdotti altri tipi di join (*join_type*), tra cui:

- **INNER JOIN**
- **LEFT [OUTER] JOIN**
- **RIGHT [OUTER] JOIN**
- **FULL [OUTER] JOIN**

**Sintassi generale**

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

Figura 171: Sintassi JOIN

dove *join_condition* è un'espressione booleana che seleziona le tuple del join da aggiungere al risultato.

Le tuple selezionate possono essere successivamente filtrate mediante la condizione specificata nella clausola `WHERE`.

**Prodotto Cartesiano**

- Formulazione con JOIN:

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

- Formulazione con ';':

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

| cognome | nome |
|---|---|
| Rossi | Acquisti |
| Verdi | Acquisti |
| Rossi | Vendite |
| Verdi | Vendite |

Figura 172: Prodotto cartesiano

### 15.13.1 INNER JOIN

**Sintassi**

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

Figura 173: Sintassi INNER JOIN

L'**INNER JOIN** rappresenta il tradizionale $\theta$-join dell'algebra relazionale. Combina ciascuna riga $r_1$ della `table1` con ciascuna riga della `table2` che soddisfa la condizione specificata nella clausola `ON`.

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

| cognome | nomerep | sede |
|---|---|---|
| Rossi | Acquisti | Verona |
| Verdi | Vendite | Verona |

Figura 174: Esempio INNER JOIN

### 15.13.2 LEFT OUTER JOIN

**Sintassi generale**

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

Figura 175: Sintassi LEFT OUTER JOIN

Il **LEFT OUTER JOIN** esegue prima un `INNER JOIN`. Successivamente, per ogni riga della `table1` che non trova corrispondenza nella `table2`, viene comunque inserita nel risultato una riga contenente i valori della `table1` e `NULL` negli attributi della `table2`.

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

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

| nomerep | cognome |
|---|---|
| Acquisti | Rossi |
| Vendite | Verdi |
| Finanza | |

Figura 176: Esempio

**Nota**

Il LEFT [OUTER] JOIN non è simmetrico!

Con le medisime tabelle si possono ottenere risultati diversi invertendo l'ordine delle tabelle nel join!

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

| cognome | nomerep |
|---|---|
| Rossi | Acquisti |
| Verdi | Vendite |

Figura 177: Nota

### 15.13.3 RIGHT OUTER JOIN

**Sintassi**

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

Figura 178: Sintassi RIGHT OUTER JOIN

Il **RIGHT OUTER JOIN** esegue prima un `INNER JOIN`. Successivamente, per ogni riga della `table2` che non trova corrispondenza nella `table1`, viene comunque inserita nel risultato una riga contenente i valori della `table2` e `NULL` negli attributi della `table1`.

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

| nomerep | cognome |
|---|---|
| Acquisti | Rossi |
| Vendite | Verdi |
| Finanza | |

Figura 179: Esempio RIGHT OUTER JOIN

### 15.13.4 FULL OUTER JOIN

**Sintassi**

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

Figura 180: FULL OUTER JOIN

Il **FULL OUTER JOIN** restituisce tutte le tuple prodotte da un `INNER JOIN`, più le tuple della prima tabella che non trovano corrispondenza nella seconda (come nel `LEFT OUTER JOIN`) e le tuple della seconda tabella che non trovano corrispondenza nella prima (come nel `RIGHT OUTER JOIN`).

Non è equivalente a un `CROSS JOIN`.

- Query: Nome, Cognome di tutti gli impiegati e città del Dipartimento in cui lavora
- SELECT i.Nome, i.Cognome, d.Città
  FROM Impiegato as i, Dipartimento
  WHERE i.Dipart = d.Nome

Prodotto cartesiano

Nota: le tabelle Impiegato e Dipartimento, hanno entrambe un attributo "nome", quindi devo specificare a quale mi riferisco

Impiegato

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

Dipartimento

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

Figura 181: Esempio

Il **join** può essere specificato direttamente nella clausola `FROM`, a livello delle tabelle coinvolte. Nello svolgimento degli esercizi utilizzeremo questa sintassi dedicata.

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

Esempio:

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

**Q8:** Selezionare i padri di persone che guadagnano più di 20.

**Algebra relazionale**

$$\pi_{Padre}(\text{Paternità} \bowtie_{Figlio=Nome} (\sigma_{Reddito>20}(\text{Persone})))$$

**SQL**

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

## 15.14 Uso di variabili (alias)

SQL permette di associare un **nome alternativo** (alias) alle tabelle che compaiono nella clausola `FROM`.

- Il nome alternativo viene utilizzato per fare riferimento alla tabella nel contesto dell'interrogazione.
- L'alias viene definito nella clausola `FROM` tramite la parola chiave `AS` (opzionale).

**Motivi per usare gli alias:**

- Fare riferimento alla tabella in modo più compatto.
- Accedere più volte alla stessa tabella all'interno della stessa interrogazione (self join).

- Q9: Estrarre tutti gli impiegati che hanno lo stesso cognome ma nome diverso, del dipartimento Produzione.

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

Impiegato

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

- Questa interrogazione confronta ciascuna riga di Impiegato con tutte le righe di Impiegato associate al dipartimento Produzione.

Figura 182: Esempio Uso di variabili (alias)

- Si può immaginare che, al momento della definizione degli alias, vengano create due copie della tabella `Impiegato`, ciascuna associata a una delle due variabili `i1` e `i2`.
- Le due copie contengono inizialmente tutte le righe della tabella `Impiegato`.
- Successivamente, sulla copia `i2` vengono selezionate solo le righe relative al dipartimento `'Produzione'`.

## 15.15 Ordinamento del risultato

Come abbiamo visto, una relazione è un insieme di tuple **non ordinato**. Nella pratica, però, spesso è necessario visualizzare le righe secondo un determinato ordine.

SQL permette di specificare un ordinamento (ascendente o discendente) delle righe dell'output tramite la clausola `ORDER BY`, che viene posta alla fine dell'interrogazione.

**Sintassi**

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

Ordinamento di default = ASC

Figura 183: Sintassi Ordinamento del risultato

Se nella clausola `ORDER BY` sono presenti più attributi, le righe vengono ordinate rispetto al primo attributo dell'elenco; a parità di valore del primo attributo si utilizza il secondo, e così via in sequenza.

- Q10: Estrarre il nome, l'eta ed il reddito delle persone con meno di trenta anni in ordine alfabetico

- SELECT Nome, Età, Reddito
  FROM Persone
  WHERE Età <'30'
  ORDER BY Età

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

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

Figura 184: Esempio: Ordinamento del risultato

- Q11: Estrarre il contenuto della tabella Automobile ordinato in base alla marca (in modo discendente) e al modello.

Automobile

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

- SELECT *
  FROM Automobile
  ORDER BY Marca DESC, Modello

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

Figura 185: Esempio Ordinamento del risultato

### 15.15.1 Operazioni Insiemistiche

SQL mette a disposizione alcuni **operatori insiemistici** che permettono di combinare i risultati di più interrogazioni.

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

- Gli operatori insiemistici, a differenza del resto del linguaggio SQL, **eliminano i duplicati per default**.
- Se si vogliono mantenere i duplicati è necessario utilizzare la parola chiave `ALL`.
- Le interrogazioni combinate devono restituire lo stesso **numero di attributi** e avere **domini compatibili**.
- La corrispondenza tra gli attributi avviene per **posizione** (ordine posizionale), non per nome.
- Se i nomi degli attributi sono diversi, di solito il sistema utilizza il nome degli attributi del **primo operando**.

### 15.15.2 Unione

- Q12: Estrarre i nomi ed i cognomi degli impiegati.

- SELECT Nome
  FROM Impiegato
  UNION
  SELECT Cognome
  FROM Impiegato

| Nome |
|---|
| Mario |
| Carlo |
| Giovanni |
| Franco |
| Lorenzo |
| Paola |
| Marco |
| Rossi |
| Bianchi |
| Verdi |
| Neri |
| Gialli |
| Rosati |

Impiegato

| Nome | Cognome | Dipart | Ufficio | Stipendio |
|---|---|---|---|---|
| Mario | Rossi | Amministrazione | 10 | 45 |
| Carlo | Bianchi | Produzione | 20 | 36 |
| Giovanni | Verdi | Amministrazione | 20 | 40 |
| Franco | Neri | Distribuzione | 16 | 45 |
| Carlo | Rossi | Direzione | 14 | 80 |
| Lorenzo | Gialli | Direzione | 7 | 73 |
| Paola | Rosati | Amministrazione | 75 | 40 |
| Marco | Franco | Produzione | 20 | 46 |

Figura 186: Esempio Unione

- Q13: Estrarre i nomi ed i cognomi degli impiegati eccetto quelli del dipartimento Amministrazione mantenendo i duplicati.

- SELECT Nome
  FROM Impiegato
  WHERE Dipart <> 'Amministrazione'
  UNION ALL
  SELECT Cognome
  FROM Impiegato
  WHERE Dipart <> 'Amministrazione'

| Nome |
|---|
| Carlo |
| Franco |
| Carlo |
| Lorenzo |
| Marco |
| Bianchi |
| Neri |
| Rossi |
| Gialli |
| Franco |

Impiegato

| Nome | Cognome | Dipart | Ufficio | Stipendio |
|---|---|---|---|---|
| Mario | Rossi | Amministrazione | 10 | 45 |
| Carlo | Bianchi | Produzione | 20 | 36 |
| Giovanni | Verdi | Amministrazione | 20 | 40 |
| Franco | Neri | Distribuzione | 16 | 45 |
| Carlo | Rossi | Direzione | 14 | 80 |
| Lorenzo | Gialli | Direzione | 7 | 73 |
| Paola | Rosati | Amministrazione | 75 | 40 |
| Marco | Franco | Produzione | 20 | 46 |

Figura 187: Esempio Unione

### 15.15.3 Intersezione

- Q14: Estrarre i cognomi degli impiegati che sono anche nomi

- SELECT Nome
  FROM Impiegato
  INTERSECT
  SELECT Cognome
  FROM Impiegato

| Nome |
|---|
| Franco |

Impiegato

| Nome | Cognome | Dipart | Ufficio | Stipendio |
|---|---|---|---|---|
| Mario | Rossi | Amministrazione | 10 | 45 |
| Carlo | Bianchi | Produzione | 20 | 36 |
| Giovanni | Verdi | Amministrazione | 20 | 40 |
| Franco | Neri | Distribuzione | 16 | 45 |
| Carlo | Rossi | Direzione | 14 | 80 |
| Lorenzo | Gialli | Direzione | 7 | 73 |
| Paola | Rosati | Amministrazione | 75 | 40 |
| Marco | Franco | Produzione | 20 | 46 |

Figura 188: Intersezione

### 15.15.4 Differenza

- Q15: Estrarre i nomi degli impiegati che **non** sono cognomi di qualche impiegato

- SELECT Nome
  FROM Impiegato
  EXCEPT
  SELECT Cognome
  FROM Impiegato

| Nome |
|---|
| Mario |
| Carlo |
| Giovanni |
| Lorenzo |
| Paola |
| Marco |

Impiegato

| Nome | Cognome | Dipart | Ufficio | Stipendio |
|---|---|---|---|---|
| Mario | Rossi | Amministrazione | 10 | 45 |
| Carlo | Bianchi | Produzione | 20 | 36 |
| Giovanni | Verdi | Amministrazione | 20 | 40 |
| Franco | Neri | Distribuzione | 16 | 45 |
| Carlo | Rossi | Direzione | 14 | 80 |
| Lorenzo | Gialli | Direzione | 7 | 73 |
| Paola | Rosati | Amministrazione | 75 | 40 |
| Marco | Franco | Produzione | 20 | 46 |

Figura 189: Differenza

## 15.16 Operatori aggregati

Gli operatori di aggregazione rappresentano una delle più importanti estensioni di SQL rispetto all'algebra relazionale.

- I principali operatori di aggregazione sono: `COUNT`, `SUM`, `MAX`, `MIN` e `AVG`.
- Nell'algebra relazionale tutte le condizioni specificate vengono valutate su **una tupla alla volta**.
- Gli operatori di aggregazione, invece, permettono di valutare proprietà che dipendono da **insiemi di tuple**.
- Prima viene eseguita l'interrogazione considerando solo le clausole `FROM` e `WHERE`.
- Successivamente l'operatore di aggregazione viene applicato alla tabella contenente il risultato dell'interrogazione.

### 15.16.1 COUNT

L'operatore `COUNT` conta il numero di righe di una tabella oppure il numero di valori di uno o più attributi.

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

- `*`: conta il numero totale di righe della tabella.
- `DISTINCT ListaAttributi`: conta il numero di valori **distinti** degli attributi specificati.
- `ALL ListaAttributi`: conta il numero di righe che hanno valori **diversi da NULL** negli attributi indicati.
- `ALL` rappresenta il **comportamento di default**.

- Q16: Estrarre il numero di impiegati del dipartimento Produzione

Impiegato

| Nome | Cognome | Dipart | Ufficio | Stipendio |
|---|---|---|---|---|
| Mario | Rossi | Amministrazione | 10 | 45 |
| Carlo | Bianchi | Produzione | 20 | 36 |
| Giovanni | Verdi | Amministrazione | 20 | 40 |
| Franco | Neri | Distribuzione | 16 | 45 |
| Carlo | Rossi | Direzione | 14 | 80 |
| Lorenzo | Gialli | Direzione | 7 | 73 |
| Paola | Rosati | Amministrazione | 75 | 40 |
| Marco | Franco | Produzione | 20 | 46 |

- SELECT COUNT(*)
  FROM Impiegato
  WHERE Dipart = 'Produzione' → 2

Figura 190: Esempio COUNT

- Q17: Estrarre il numero di valori diversi dell'attributo Stipendio fra tutte le righe di Impiegato

Impiegato

| Nome | Cognome | Dipart | Ufficio | Stipendio |
|---|---|---|---|---|
| Mario | Rossi | Amministrazione | 10 | 45 |
| Carlo | Bianchi | Produzione | 20 | 36 |
| Giovanni | Verdi | Amministrazione | 20 | 40 |
| Franco | Neri | Distribuzione | 16 | 45 |
| Carlo | Rossi | Direzione | 14 | 80 |
| Lorenzo | Gialli | Direzione | 7 | 73 |
| Paola | Rosati | Amministrazione | 75 | 40 |
| Marco | Franco | Produzione | 20 | 46 |

- SELECT COUNT(DISTINCT Stipendio)
  FROM Impiegato → 6

Figura 191: Esempio Count

- Q18: Estrarre il numero di righe che possiedono un valore non nullo per l'attributo Nome

Impiegato

| Nome | Cognome | Dipart | Ufficio | Stipendio |
|---|---|---|---|---|
| Mario | Rossi | Amministrazione | 10 | 45 |
| Carlo | Bianchi | Produzione | 20 | 36 |
| Giovanni | Verdi | Amministrazione | 20 | 40 |
| Franco | Neri | Distribuzione | 16 | 45 |
| Carlo | Rossi | Direzione | 14 | 80 |
| Lorenzo | Gialli | Direzione | 7 | 73 |
| Paola | Rosati | Amministrazione | 75 | 40 |
| Marco | Franco | Produzione | 20 | 46 |

- SELECT COUNT(ALL Nome)
  FROM Impiegato
- SELECT COUNT(Nome)
  FROM Impiegato

→ 8

Figura 192: Esempio Count

### 15.16.2 Operatori aggregati: SUM.MAX,MIN,AVG

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

- `SUM`: restituisce la somma dei valori posseduti dall'espressione.
- `MAX` / `MIN`: restituiscono rispettivamente il valore massimo o minimo tra quelli posseduti dall'espressione.
- `AVG`: restituisce la media dei valori.
- `SUM` e `AVG` ammettono come argomento solo espressioni che rappresentano valori **numerici** o **intervalli di tempo**.
- `MAX` e `MIN` ammettono come argomento espressioni su cui è definito un **ordinamento** (ad esempio anche stringhe).
- `DISTINCT` elimina i valori duplicati prima dell'applicazione dell'operatore.
- `ALL` considera tutti i valori, **trascurando solo i valori NULL**. Questo è il comportamento di default.
- `DISTINCT` e `ALL` non hanno effetto sugli operatori `MIN` e `MAX`.

- Q19: Estrarre la somma degli stipendi del dipartimento Amministrazione

Impiegato

| Nome | Cognome | Dipart | Ufficio | Stipendio |
|---|---|---|---|---|
| Mario | Rossi | Amministrazione | 10 | 45 |
| Carlo | Bianchi | Produzione | 20 | 36 |
| Giovanni | Verdi | Amministrazione | 20 | 40 |
| Franco | Neri | Distribuzione | 16 | 45 |
| Carlo | Rossi | Direzione | 14 | 80 |
| Lorenzo | Gialli | Direzione | 7 | 73 |
| Paola | Rosati | Amministrazione | 75 | 40 |
| Marco | Franco | Produzione | 20 | 46 |

- SELECT SUM(Stipendio)
  FROM Impiegato
  WHERE Dipart = 'Amministrazione' → 125

Figura 193: Esempio SUM

- Q20: Estrarre gli stipendi minimo, massimo e medio fra quelli di tutti gli impiegati

Impiegato

| Nome | Cognome | Dipart | Ufficio | Stipendio |
|---|---|---|---|---|
| Mario | Rossi | Amministrazione | 10 | 45 |
| Carlo | Bianchi | Produzione | 20 | 36 |
| Giovanni | Verdi | Amministrazione | 20 | 40 |
| Franco | Neri | Distribuzione | 16 | 45 |
| Carlo | Rossi | Direzione | 14 | 80 |
| Lorenzo | Gialli | Direzione | 7 | 73 |
| Paola | Rosati | Amministrazione | 75 | 40 |
| Marco | Franco | Produzione | 20 | 46 |

- SELECT MIN(Stipendio), MAX(Stipendio), AVG(Stipendio)
  FROM Impiegato → 36, 80, 50.625

Figura 194: Esempio MAX,MIN,AVG

- Q21: Estrarre lo stipendio massimo tra quelli degli impiegati che lavorano in un dipartimento con sede a Milano

- SELECT MAX( I.Stipendio )
  FROM Impiegato as I JOIN Diparmento as D
  ON I.Dipart = D.Nome
  WHERE D.Città = 'Milano'

80

Impiegato

| Nome | Cognome | Dipart | Ufficio | Stipendio |
|---|---|---|---|---|
| Mario | Rossi | Amministrazione | 10 | 45 |
| Carlo | Bianchi | Produzione | 20 | 36 |
| Giovanni | Verdi | Amministrazione | 20 | 40 |
| Franco | Neri | Distribuzione | 16 | 45 |
| Carlo | Rossi | Direzione | 14 | 80 |
| Lorenzo | Gialli | Direzione | 7 | 73 |
| Paola | Rosati | Amministrazione | 75 | 40 |
| Marco | Franco | Produzione | 20 | 46 |

Dipartimento

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

Figura 195: Esempio MAX

```
SELECT Cognome, Nome, MAX(I.Stipendio)
FROM Impiegato AS I JOIN Dipartimento AS D
ON I.Dipart = D.Nome
WHERE D.Città = 'Milano'
```

- La seguente interrogazione non è corretta.
- Gli operatori di aggregazione non sono un meccanismo di selezione, ma funzioni che restituiscono un valore quando vengono applicate a un insieme di tuple.
- Gli attributi `Cognome` e `Nome` restituiscono un valore per **ogni tupla selezionata**, mentre `MAX()` restituisce un **unico valore** per l'intero insieme.
- Si crea quindi un'incoerenza tra attributi normali e funzione di aggregazione.
- Per risolvere questa situazione è necessario utilizzare il meccanismo di **raggruppamento** tramite la clausola `GROUP BY`.

## 15.17 Interrogazioni con raggruppamento

Gli operatori di aggregazione possono essere applicati separatamente a **sottoinsiemi di righe**.

- La clausola `GROUP BY` permette di specificare come dividere le righe della tabella in sottoinsiemi.
- La clausola consente di indicare un insieme di attributi rispetto ai quali effettuare il raggruppamento.
- L'interrogazione raggruppa quindi tutte le righe che possiedono gli **stessi valori** per tali attributi.

- Q22: Estrarre la somma degli stipendi di tutti gli impiegati dello stesso dipartimento.

- SELECT Dipart, SUM(Stipendio)
  FROM Impiegato
  GROUP BY Dipart

Impiegato

| Nome | Cognome | Dipart | Ufficio | Stipendio |
|---|---|---|---|---|
| Mario | Rossi | Amministrazione | 10 | 45 |
| Carlo | Bianchi | Produzione | 20 | 36 |
| Giovanni | Verdi | Amministrazione | 20 | 40 |
| Franco | Neri | Distribuzione | 16 | 45 |
| Carlo | Rossi | Direzione | 14 | 80 |
| Lorenzo | Gialli | Direzione | 7 | 73 |
| Paola | Rosati | Amministrazione | 75 | 40 |
| Marco | Franco | Produzione | 20 | 46 |

Figura 196: Esempio GROUP BY

- SELECT Dipart, SUM(Stipendio)
  FROM Impiegato
  GROUP BY Dipart

Impiegato

| Nome | Cognome | Dipart | Ufficio | Stipendio |
|---|---|---|---|---|
| Mario | Rossi | Amministrazione | 10 | 45 |
| Carlo | Bianchi | Produzione | 20 | 36 |
| Giovanni | Verdi | Amministrazione | 20 | 40 |
| Franco | Neri | Distribuzione | 16 | 45 |
| Carlo | Rossi | Direzione | 14 | 80 |
| Lorenzo | Gialli | Direzione | 7 | 73 |
| Paola | Rosati | Amministrazione | 75 | 40 |
| Marco | Franco | Produzione | 20 | 46 |

Esecuzione:

1. Viene eseguita l'interrogazione come se la clausola GROUP BY non esistesse:
   SELECT Dipart, SUM(Stipendio)
   FROM Impiegato

| Dipart | Stipendio |
|---|---|
| Amministrazione | 45 |
| Produzione | 36 |
| Amministrazione | 40 |
| Distribuzione | 45 |
| Direzione | 80 |
| Direzione | 73 |
| Amministrazione | 40 |
| Produzione | 46 |

Figura 197: Esempio GROUP BY

- SELECT Dipart, SUM(Stipendio)
  FROM Impiegato
  GROUP BY Dipart

Impiegato

| Nome | Cognome | Dipart | Ufficio | Stipendio |
|---|---|---|---|---|
| Mario | Rossi | Amministrazione | 10 | 45 |
| Carlo | Bianchi | Produzione | 20 | 36 |
| Giovanni | Verdi | Amministrazione | 20 | 40 |
| Franco | Neri | Distribuzione | 16 | 45 |
| Carlo | Rossi | Direzione | 14 | 80 |
| Lorenzo | Gialli | Direzione | 7 | 73 |
| Paola | Rosati | Amministrazione | 75 | 40 |
| Marco | Franco | Produzione | 20 | 46 |

Esecuzione:

2. La tabella ottenuta viene analizzata, dividendo le righe in insiemi caratterizzati dallo stesso valore degli attributi che compaiono nella clausola GROUP BY.

| Dipart | Stipendio |
|---|---|
| Amministrazione | 45 |
| Amministrazione | 40 |
| Amministrazione | 40 |
| Produzione | 36 |
| Produzione | 46 |
| Distribuzione | 45 |
| Direzione | 80 |
| Direzione | 73 |

Figura 198: Esempio GROUP BY

- SELECT Dipart, SUM(Stipendio)
  FROM Impiegato
  GROUP BY Dipart

Impiegato

| Nome | Cognome | Dipart | Ufficio | Stipendio |
|---|---|---|---|---|
| Mario | Rossi | Amministrazione | 10 | 45 |
| Carlo | Bianchi | Produzione | 20 | 36 |
| Giovanni | Verdi | Amministrazione | 20 | 40 |
| Franco | Neri | Distribuzione | 16 | 45 |
| Carlo | Rossi | Direzione | 14 | 80 |
| Lorenzo | Gialli | Direzione | 7 | 73 |
| Paola | Rosati | Amministrazione | 75 | 40 |
| Marco | Franco | Produzione | 20 | 46 |

Esecuzione:

3. Dopo che le righe sono state raggruppate in sottoinsiemi, l'operatore aggregato viene applicato separatamente su ogni sottoinsieme.

| Dipart | sum(Stipendio) |
|---|---|
| Amministrazione | 125 |
| Produzione | 82 |
| Distribuzione | 45 |
| Direzione | 153 |

Figura 199: Esempio GROUP BY

In una interrogazione che utilizza la clausola `GROUP BY`, nella clausola `SELECT` possono comparire solo:

- attributi che appartengono all'insieme specificato nella clausola `GROUP BY`;
- oppure operatori di aggregazione.

**Esempio di interrogazione non corretta**

```
SELECT Ufficio
FROM Impiegato
GROUP BY Dipart
```

**Errore:**

- Ad ogni valore dell'attributo `Dipart` possono corrispondere diversi valori dell'attributo

`Ufficio`.

- Ogni sottoinsieme di righe generato dal `GROUP BY` deve corrispondere ad **una sola riga** nella tabella risultato.

### 15.17.1 HAVING

È possibile applicare delle **selezioni ai gruppi** generati tramite la clausola `GROUP BY`.

- La clausola `HAVING` permette di specificare condizioni che devono essere soddisfatte dai **gruppi di tuple** generati dopo l'esecuzione del raggruppamento tramite `GROUP BY`.
- A differenza della clausola `WHERE`, che filtra le **singole tuple** prima del raggruppamento, la clausola `HAVING` filtra i **gruppi** dopo il raggruppamento.

**Esempio**

```
SELECT Dipart, SUM(Stipendio) AS SommaStipendi
FROM Impiegato
GROUP BY Dipart
HAVING SUM(Stipendio) > 100;
```

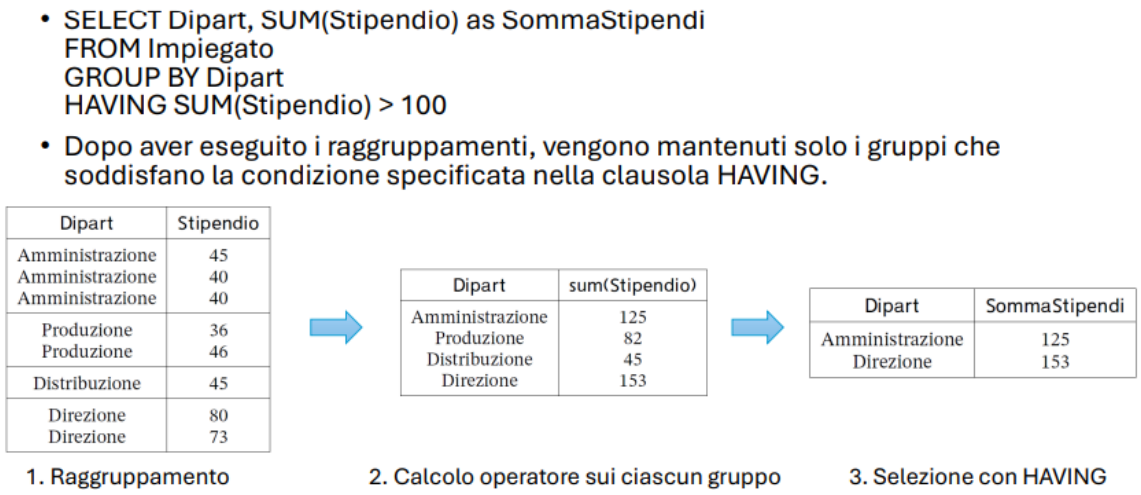



Figura 200: HAVING + GROUP BY

- La clausola `WHERE` applica delle selezioni su **singole righe**, prima dell'eventuale raggruppamento tramite `GROUP BY`.
- La clausola `HAVING` applica delle selezioni su **gruppi di righe**, generati dopo l'esecuzione del raggruppamento tramite `GROUP BY`.

**Esempio**

```
SELECT Dipart
FROM Impiegato
WHERE Ufficio = 20
GROUP BY Dipart
HAVING AVG(Stipendio) > 25;
```

**Ordine logico di esecuzione**

1. `WHERE` → seleziona le righe che verranno raggruppate.

2. `GROUP BY` → forma i gruppi con le righe selezionate.
3. `HAVING` → seleziona i gruppi che faranno parte del risultato.

## 15.18 Interrogazioni nidificate

SQL permette la costruzione di interrogazioni che presentano al loro interno altre interrogazioni. La nidificazione può avvenire nelle clausole **SELECT**, **FROM**, **WHERE**. L'uso più comune è la nidificazione nella clausola WHERE

**Clausola FROM :** Con l'utilizzo di questa opzione, è possibile comporre catene di interrogazioni, in cui ciascuna interrogazione utilizza il risultato della precedente come se fosse una tabella di base.

```
SELECT titolo, prezzoIntero
FROM Mostra, ( SELECT MAX ( prezzoIntero )
FROM Mostra ) AS T( prezzoMax )
WHERE prezzoIntero = prezzoMax;
```

Figura 201: nidificazione con clausola FROM

**Clausola WHERE :** l predicato (predicato complesso) nella clausola WHERE confronta un valore (ottenuto come espressione valutata sulla singola riga), con il risultato dell'esecuzione di un'interrogazione SQL (in generale un insieme di valori). Occorre estendere gli operatori di confronto per poter realizzare la comparazione tra un valore e un insieme di valori.
Soluzione offerta da SQL: estensione degli operatori di confronto (>,<, =, <=, >=, <>) con ANY/SOME e ALL (IN, NOT IN).

- **ANY/SOME**: specifica che la riga soddisfa la condizione se risulta vero il confronto (con l'operatore specificato) tra il valore dell'attributo per la riga e almeno uno degli elementi restituiti dall'interrogazione.
- **ALL**: specifica che la riga soddisfa la condizione solo se tutti gli elementi restituiti dall'interrogazione nidificata rendono vero il confronto.
- **IN e NOT IN**: usati per rappresentare l'appartenenza o l'esclusione rispetto ad un insieme (identici a = ANY o <>ALL)

```
expression operator ANY( subquery )
expression operator SOME( subquery )
```

Figura 202: Sintassi Operatori ANY e SOME

**Operatore ANY/SOME**

- **expression** è un'espressione che coinvolge attributi della SELECT principale.

- (**subquery**) è una SELECT che deve restituire UNA sola colonna;
- **perator** è un operatore di confronto, come '=', '>=',...
- **ANY** è vero se la condizione è vera per almeno un valore della subquery.
- **SOME** è uno sinonimo di ANY.

**Visualizzare il nome degli insegnamenti che hanno un numero di crediti inferiore alla media dell'ateneo di un qualsiasi anno accademico.**

```
SELECT DISTINCT I.nomeins , IE.crediti
FROM Insegn I JOIN InsErogato IE ON I.id=IE.id_insegn
WHERE IE.crediti < ANY (
SELECT AVG ( crediti ) FROM InsErogato
WHERE modulo =0
GROUP BY annoaccademico
);
```

Figura 203: Esempio ANY e SOME

```
expression operator ALL( subquery )
```

Figura 204: Sintassi Operatore ALL

**Operatore ALL**

- **expression** è un'espressione che coinvolge attributi della SELECT principale.
- (**subquery**) è una SELECT che deve restituire UNA sola colonna;
- **operator** è un operatore di confronto, come '=', '>=', ...
- **ALL**: La condizione è vera solo se è vera per TUTTI i valori della subquery

**Trovare il nome degli insegnamenti con almeno un docente e crediti maggiori rispetto ai crediti di ciascun insegnamento del corso di laurea con id=6. (si considerano solo occorrenze insegnamento genitore).**

```
SELECT DISTINCT I. nomeins , IE. crediti
FROM Insegn I JOIN InsErogato IE ON I.id = IE. id_insegn
JOIN Docenza D ON IE.id = D. id_inserogato
WHERE IE. modulo = 0 AND IE. crediti > ALL (
SELECT crediti FROM InsErogato
WHERE id_corsostudi = 6 AND modulo =0
);
```

(1485 righe )

Figura 205: Esempio Operatore ALL

**Operatore IN:** E' un operatore utilizzabile nei predicati complessi

```
[ROW] ( expr [ ,...]) IN ( subquery )
```

Figura 206: Operatore IN

- **expr** è un'espressione costruita con un attributo della query principale. Ci possono essere una o più espressioni.
- La (**subquery**) deve restituire un numero di colonne pari al numero di espressioni in (expr [,...]).
- I valori dell'espressioni vengono confrontati con i valori di ciascuna riga del risultato di (subquery).
- Il confronto ritorna vero se i valori sono uguali ai valori di almeno una riga della subquery.

**Trovare gli impiegati che hanno stesso nome e cognome di qualcuno in un'altra azienda.**

```
SELECT I.nome , I.cognome
FROM Impiegato I
WHERE ROW( I.nome , I.cognome ) IN (
SELECT I1.nome, I1. cognome FROM ImpiegatoAltraAzienda I1
);
```

Figura 207: Operatore IN

**Operatore EXISTS:** E' una clausola utilizzabile nei predicati complessi.

```
EXISTS (subquery)
```

Figura 208: Sintassi EXISTS

- Una **subquery** è una `SELECT`.
- L'operatore `EXISTS` restituisce:
  - **falso** se la subquery non produce alcuna riga;
  - **vero** se la subquery produce almeno una riga.
- `EXISTS` è particolarmente significativo quando la subquery è **correlata** alla query esterna, ovvero quando utilizza valori della riga corrente della query principale (*data binding*).
- In questo caso, la subquery viene valutata per ogni riga della query esterna.

**Determinare i nomi degli impiegati che sono diversi tra loro ma di pari lunghezza.**

```
SELECT I. nome
FROM Impiegato I
WHERE EXISTS (
SELECT 1 FROM Impiegato I1 WHERE I.nome <> I1. nome
AND CHAR_LENGTH( I.nome ) = CHAR_LENGTH( I1. nome )
);
```

I.nome nella subquery è il valore di nome nella riga corrente della SELECT principale.

Figura 209: Operatore EXISTS

**Visualizzare il nome dei corsi di studio che nel 2006/2007 non hanno erogato insegnamenti il cui nome contiene la sottostringa 'Info'.**

```
SELECT CS. nome FROM CorsoStudi CS
WHERE NOT EXISTS (
SELECT 1 FROM InsErogato IE JOIN Insegn I ON IE. id_insegn =
I.id
WHERE I.nomeins LIKE '%Info%'
AND IE.annoaccademico = '2006/2007'
AND IE.id_corsostudi = CS.id);
```

Figura 211: Esempio NOT EXISTS

Se si usano attributi esterni nella subquery (data binding), la subquery deve essere valutata per ogni riga della SELECT principale.

**Visualizzare il nome e il cognome dei docenti, escludendo i coordinatori, che hanno tenuto almeno due insegnamenti (o moduli) con più di 24 crediti nel 2010/2011.**

```
SELECT P.nome , P.cognome
FROM Persona P
WHERE EXISTS (
SELECT 1 FROM Docenza D JOIN InsErogato IE ON D. id_inserogato = IE.id
WHERE IE. crediti > 24 AND D. coordinatore = '0'
AND IE. annoaccademico = '2010/2011' AND D. id_persona = P.id
GROUP BY D. id_persona
HAVING COUNT (*) >=2
);
```

Figura 210: Esempio Operatore EXISTS

### 15.18.1 Interrogazioni di tipo insiemistico

- Gli operatori insiemistici si possono applicare solo quando `query1` e `query2` producono risultati con:
  - lo **stesso numero di colonne**;
  - tipi di dato **compatibili** tra loro.
- Tutti gli operatori eliminano i **duplicati** dal risultato, a meno che non venga specificata la clausola `ALL`.
- `UNION`: unisce il risultato di `query2` a quello di `query1`.
- `INTERSECT`: restituisce le righe presenti sia nel risultato di `query1` sia in quello di `query2`.
- `EXCEPT`: restituisce le righe di `query1` che non sono presenti nel risultato di `query2`, realizzando la **differenza insiemistica**.

```
Visualizzare i nomi degli insegnamenti e i nomi dei corsi di laurea
che non iniziano per 'A' mantenendo i duplicati.

SELECT nomeins
FROM Insegn
WHERE NOT nomeins LIKE 'A%'

UNION ALL

SELECT nome
FROM CorsoStudi
WHERE NOT nome LIKE 'A%';
```

Figura 212: Esempio UNION

```
Visualizzare i nomi degli insegnamenti che sono anche nomi
di corsi di laurea.

SELECT nomeins
FROM Insegn

INTERSECT ALL

SELECT nome
FROM CorsoStudi;
```

Figura 213: Esempio INRERSECT

```
Visualizzare i nomi degli insegnamenti che NON sono anche
nomi di corsi di laurea.

SELECT nomeins
FROM Insegn

EXCEPT

SELECT nome
FROM CorsoStudi ;
```

Figura 214: Esempio EXCEPT

### 15.18.2 Le VISTE

- Le **viste** sono **tabelle virtuali** il cui contenuto dipende dai dati presenti in altre tabelle della base di dati.
- In SQL, una vista viene definita associando:

  - un **nome**;
  - una **lista di attributi**;
  - il risultato di una `SELECT`.
- Ogni volta che una vista viene utilizzata, viene eseguita la query che la definisce.
- Nell'interrogazione che definisce una vista possono comparire anche altre viste.
- Tuttavia, SQL **non ammette**:
  - **dipendenze immediate**: una vista definita in termini di se stessa;
  - **dipendenze ricorsive**: definizioni che prevedono una parte base e una parte ricorsiva;
  - **dipendenze transitive circolari**: ad esempio $V_1$ definita tramite $V_2$, $V_2$ tramite $V_3$, $\ldots$, $V_n$ tramite $V_1$.

```
CREATE [ TEMP ] VIEW nome [( col_name [, ...]) ] AS query
```

Figura 215: SINTASSI VISTA

- `TEMP`: indica che la vista è **temporanea**. Quando la sessione termina (disconnessione), la vista viene automaticamente eliminata. Si tratta di un'estensione di **PostgreSQL**. Nella base di dati `did2014` è possibile creare solo viste temporanee.
- `column_name`: rappresenta i nomi delle colonne della vista.
  - Se non vengono specificati, i nomi sono derivati dalle colonne restituite dalla query;
  - possono essere definiti esplicitamente tramite **alias**.
- La **query** deve restituire un insieme di attributi:
  - con lo stesso **numero di colonne**;
  - nello stesso **ordine**;

  rispetto a quelli specificati in `(column_name [, ...])`, se presenti.
- Una vista è **aggiornabile** solo se ogni tupla della vista corrisponde a **una sola tupla** di ciascuna tabella di base.

**Definire la vista che contiene gli insegnamenti erogati completi di nomeins e codiceins presi dalla tabella Insegn.**

```
CREATE TEMP VIEW InsErogatiCompleti AS
SELECT I. nomeins , I. codiceins , IE .*
FROM InsErogato IE
JOIN Insegn I ON IE. id_insegn = I.id;
```

Figura 216: Esempio1

**Si vuole determinare quali sono i corsi di studi con il massimo numero di nome di insegnamenti diversi.**

Prima si crea una vista:

```
CREATE TEMP VIEW InsCorsoStudi (Nome , NumIns) AS
SELECT CS.nome , COUNT ( DISTINCT I. nomeins )
FROM CorsoStudi CS
JOIN InsErogato IE ON CS.id=IE. id_corsostudi
JOIN Insegn I ON IE. id_insegn = I.id
GROUP BY CS. nome;
```

Figura 217: Esempio Vista2

Poi la si usa:

```
SELECT Nome , NumIns
FROM InsCorsoStudi
WHERE NumIns = ANY (
SELECT MAX ( NumIns ) FROM InsCorsoStudi
);
```

Figura 218: Esempio Vista 3