Appunti Completi di Basi di Dati

Basati sulle dispense del Prof. Danilo Montesi

Alessandro Amella, Gemini e Claude

19 maggio 2025

Indice

Informazioni sul Documento

Questo documento è la versione compilata automaticamente di tutti i miei appunti di Basi di Dati per il corso di Laurea in Informatica all'Università di Bologna.

- **Fonte**: Tutti gli appunti sono disponibili come file separati nella repository GitHub: https://github.com/alessandroamella/appunti-db
- **Generazione**: Questo PDF è stato generato automaticamente utilizzando uno script che unisce tutti i singoli file .tex degli appunti.
- **Immagini**: Le immagini sono generate con PlantUML. Maggiori informazioni sono disponibili nel README del progetto.
- **Aggiornamenti**: Per la versione più recente degli appunti, visita la pagina delle release: https://github.com/alessandroamella/appunti-db/releases/latest
- **Uso di AI**: Ho usato Gemini e Claude a manetta.

Sentiti libero di utilizzare, condividere o contribuire a questi appunti attraverso la repository GitHub.

Capitolo 1

Introduzione ai Database

1.1 Informazioni Chiave per il Corso

Per avere successo in questo corso, il professore sottolinea l'importanza di:

- **Studio Autonomo:** Concentrarsi sui concetti fondamentali.
- **Esperienza Personale:** Cercare di collegare i concetti a proprie esperienze pregresse.
- **Esercizi:** Svolgere regolarmente gli esercizi proposti.
- **Progetto/Esercitazioni Pratiche:**
  - Sviluppare un progetto.
  - OPPURE partecipare attivamente agli esercizi durante le lezioni.
  - Utilizzare strumenti concreti come: `DB2`, `SQLServer`, `Oracle`, `PostgreSQL`, `MySQL`, `MS Access`, ecc. (`PostgreSQL` e `MySQL` sono ottimi punti di partenza se già conosci SQL).

1.2 Concetti Fondamentali

1.2.1 Cos'è un Database

- **Definizione Generale:** Un insieme organizzato di dati che supporta lo svolgimento di attività specifiche (per un'istituzione, un'azienda, un ufficio, una persona).
  - *Esempio pratico:* La tua rubrica telefonica è un piccolo database personale. L'anagrafe comunale è un database istituzionale.
- **Definizione Specifica (nel contesto del corso):** Un insieme di dati gestito da un **DBMS** (DataBase Management System).

1.2.2 Sistema Informativo vs. Sistema IT

- **Sistema Informativo:** La componente di un'organizzazione che gestisce le informazioni rilevanti per raggiungere gli obiettivi aziendali. Può esistere anche senza computer (pensa agli archivi cartacei di secoli fa). Include:
  - Raccolta e acquisizione informazioni.
  - Memorizzazione e conservazione.
  - Elaborazione, trasformazione, produzione.
  - Distribuzione, comunicazione, scambio.
- **Sistema IT (Information Technology):** La parte *automatizzata* del sistema informativo che utilizza tecnologie informatiche.
  - *Gerarchia:* Impresa $>$ Organizzazione $>$ Sistema Informativo $>$ Sistema IT. Un database è una componente chiave del Sistema IT.

1.2.3 Informazione vs. Dati

- **Informazione:** Fatti forniti o appresi su qualcosa o qualcuno. Ha un significato intrinseco.
  - *Esempio:* 'La lezione di Basi di Dati è alle 9:45 in aula N2 con il Prof. Rossi.'
- **Dati:** Rappresentazioni grezze dell'informazione, spesso codificate. Un punto di partenza fisso per un'operazione. I dati, da soli, possono non avere significato senza un'interpretazione.
  - *Esempio (dal cartello stradale):* Il dato `"8-17"` sul cartello. Da solo non significa nulla. Diventa informazione ('Divieto di sosta dalle 8 alle 17') quando interpretato nel contesto del cartello, del giorno ('Mon-Fri'), e delle regole stradali.
  - *Esempio pratico (MongoDB/JSON):*

    ```
    { "nome": "Mario", "cognome": "Rossi", "eta": 30 }
    ```

    L'informazione è che c'è una persona di nome Mario Rossi di 30 anni.
- **Processo:** L'informazione viene organizzata, codificata e memorizzata sotto forma di dati.

1.2.4 Perché i Dati sono Importanti?

- È difficile rappresentare precisamente informazioni e conoscenze complesse.
- I **dati sono una risorsa strategica** perché sono più stabili rispetto ad altre rappresentazioni (processi di business, tecnologie, ruoli umani).
  - *Esempio:* I dati anagrafici di una persona o i dati di un conto bancario rimangono fondamentalmente gli stessi anche se cambiano le procedure dell'ufficio anagrafe, il software della banca o le persone che ci lavorano.

1.2.5 DBMS: DataBase Management System

Un software progettato per creare, gestire e interrogare database. Deve gestire collezioni di dati che sono:

- **Grandi (Big):** Tipicamente più grandi della memoria principale (RAM) del sistema. Si parla di Terabyte, miliardi di record.
- **Persistenti (Persistent):** La loro durata di vita è indipendente dai processi che li utilizzano. I dati sopravvivono allo spegnimento del computer o alla chiusura dell'applicazione.
  - *Esempio pratico:* Quando la tua app Node.js scrive su un database PostgreSQL usando Prisma, quei dati rimangono nel DB anche se riavvii il server Node.js.
- **Condivisi (Shared):** Accessibili e utilizzabili da multiple applicazioni e utenti, spesso contemporaneamente.
  - Questo porta a problemi di:
    - **Ridondanza:** Stessi dati duplicati in più posti.
    - **Inconsistenza:** Se i dati duplicati non vengono aggiornati ovunque, si creano discrepanze.
  - Un DB centralizzato riduce questi problemi.

Un DBMS deve inoltre garantire:

- **Privacy/Sicurezza (Privacy):** Controllo degli accessi. Chi può vedere/modificare cosa?
  - *Esempio SQL:*

    ```
    GRANT SELECT ON tabella_clienti TO 'utente_marketing';
    ```

- **Affidabilità (Reliability):** Tolleranza ai guasti (hardware o software). I dati devono essere preservati. La tecnica cruciale è la gestione delle **Transazioni**.
- **Efficienza (Efficiency):** Uso ottimale delle risorse (memoria, tempo di CPU, I/O disco) per rispondere rapidamente alle richieste.
  - *Esempio pratico:* L'uso di indici (`CREATE INDEX`) su colonne frequentemente interrogate accelera enormemente le ricerche.
- **Efficacia (Effectiveness):** Fornire funzionalità potenti e flessibili che migliorino le attività degli utenti.

Esempi di DBMS: Oracle, SQLServer, DB2, **MySQL, PostgreSQL** (Relazionali/SQL), MS Access, SQLite (embedded), BigQuery (cloud data warehouse), **MongoDB** (NoSQL documentale).

1.2.6 Transazioni

Una transazione è una sequenza di operazioni sul database che viene trattata come una singola unità logica di lavoro. Deve avere le proprietà **ACID** (anche se non esplicitamente nominate come acronimo, i concetti ci sono):

- **Atomicità (Atomic):** Le operazioni all'interno di una transazione vengono eseguite *tutte o nessuna*. Se una qualsiasi operazione fallisce, l'intera transazione viene annullata (rollback) e il database torna allo stato precedente.
  - *Esempio classico:* Trasferimento di denaro da un conto A a un conto B. Deve avvenire sia il prelievo da A SIA il deposito su B. Se uno dei due fallisce, nessuno dei due deve avere effetto.
- **Coerenza (Consistency):** Una transazione porta il database da uno stato coerente a un altro stato coerente. Rispetta tutti i vincoli definiti.
- **Isolamento (Isolation, riferito a "Concurrent"):** Le transazioni concorrenti (eseguite contemporaneamente da più utenti/processi) non devono interferire tra loro. Ogni transazione deve apparire come se fosse l'unica in esecuzione.
  - *Esempio:* Se due utenti tentano di prenotare l'ultimo posto disponibile su un volo, il sistema deve garantire che solo uno ci riesca, evitando doppie prenotazioni.
- **Durabilità (Durability, riferito a "Permanent"):** Una volta che una transazione è stata confermata (**commit**), i suoi effetti sono permanenti e sopravvivono a guasti del sistema (es. crash del server, mancanza di corrente).
  - *Esempio SQL:* Dopo un `COMMIT;`, i dati sono scritti in modo sicuro.

Punti di vista sulla transazione:

- **Utente:** Un'operazione di business completa (es. 'registra nuovo cliente', 'paga fattura').
- **Sistema:** Una sequenza indivisibile di operazioni che garantisce affidabilità.

1.2.7 DBMS vs. File System

- **File System:** Gestisce file e cartelle. L'accesso ai dati è 'grezzo' (tutto il file o niente).
  - *Esempio:* Leggere un file CSV riga per riga.
- **DBMS:** Estende le funzionalità del file system offrendo:
  - Accesso granulare ai dati (righe specifiche, colonne, filtri complessi).
  - Linguaggi di interrogazione potenti (SQL).
  - Controllo della concorrenza, gestione delle transazioni, sicurezza, ecc.
  - Indipendenza dei dati (vedi sotto).

1.3 Architettura e Modelli dei Dati

1.3.1 Evoluzione della Gestione dei Dati

- **Anni '70:** Applicazioni gestivano i propri file. Logica dei dati e logica applicativa mescolate. Alta ridondanza.
- **Anni '80:** Primi DBMS. Separazione tra dati e applicazioni. Nascono le 'tabelle dati'.
- **Anni '90 (Comportamento Procedurale):** Introduzione di logica condivisa (procedure) all'interno del DBMS.
  - **Stored Procedures:** Blocchi di codice (spesso SQL esteso) memorizzati nel DB ed eseguibili.
  - **Trigger:** Procedure speciali eseguite automaticamente dal DBMS in risposta a determinati eventi (INSERT, UPDATE, DELETE su una tabella).
    - *Esempio SQL (concettuale):*

```
CREATE TRIGGER aggiorna_quantita_magazzino
AFTER INSERT ON ordini
FOR EACH ROW
BEGIN
UPDATE prodotti
SET quantita = quantita - NEW.quantita_ordinata
WHERE id = NEW.prodotto_id;
END;
```

- **Anni 2000 (Web):** Architetture a più livelli (Client con Javascript, Server Applicativo con Java/Node.js, DBMS).
- **Anni 2010 (Mobile):** Simile all'architettura web, ma con client mobile.

1.3.2 Descrizione dei Dati e Data Model

- **Senza DBMS:** Ogni software descrive internamente la struttura dei file che processa. Molteplici descrizioni possono portare a inconsistenza.
- **Con DBMS:** Esiste un **catalogo** o **dizionario dei dati** (una porzione del database stesso) che contiene una descrizione centralizzata dei dati (lo schema). Questa descrizione è condivisa tra tutte le applicazioni.
- **Data Model:** Un insieme di costrutti usati per organizzare e descrivere il comportamento dei dati.
  - Componente cruciale: fornire **struttura** ai dati (tramite 'costruttori di tipo').
  - Il **modello relazionale** (usato da SQL) fornisce il costruttore 'relazione' (tabella), che permette di definire insiemi di record omogenei.

1.3.3 Schema e Istanza

- **Schema (Intensionale):** La descrizione della struttura del database. È (relativamente) invariante nel tempo. Definisce 'come sono fatti' i dati.
  - *Esempio SQL:*

```
CREATE TABLE Utenti (
ID INT PRIMARY KEY,
Nome VARCHAR(100),
Email VARCHAR(100) UNIQUE
);
```

definisce lo schema della tabella `Utenti`. Include nomi delle colonne, tipi di dato, chiavi, vincoli. Corrisponde agli 'headers' delle tabelle.

- **Istanza (Estensionale):** I valori effettivi contenuti nel database in un dato momento. Cambia rapidamente con le operazioni di inserimento, modifica, cancellazione. Corrisponde al 'corpo' delle tabelle (le righe).

1.3.4 Livelli di Modellazione

- **Modellazione Concettuale:**
  - Rappresenta i dati in modo indipendente da sistemi specifici.
  - Descrive concetti del mondo reale.
  - Usata nella fase preparatoria di un progetto (analisi dei requisiti).
  - Il linguaggio più diffuso è **Entity-Relationship (E-R)**.
    - *Esempio:* In un sistema universitario, identifichiamo entità come 'Studente', 'Corso', 'Docente' e relazioni come 'Studente SI ISCRIVE A Corso', 'Docente TIENE Corso'.
- **Modelli Logici:**
  - Usati dai DBMS per memorizzare e organizzare i dati.
  - Sono indipendenti dalla rappresentazione fisica.
  - Esempi: **Relazionale (SQL)**, gerarchico (antenato di JSON/XML), a oggetti, XML.
    - *Esempio:* Il diagramma E-R viene tradotto in uno schema di tabelle relazionali (`CREATE TABLE...`).

1.3.5 Architettura di un DBMS

Architettura standard ANSI/SPARC a tre livelli per garantire l'**indipendenza dei dati**:

1. **Schema Interno (o Fisico):**
   - Descrive come i dati sono fisicamente memorizzati (file, indici, puntatori).
   - È il livello più basso, nascosto agli utenti e alla maggior parte degli sviluppatori.
   - *Esempio:* Il DBMS decide di memorizzare una tabella in un certo file su disco e di creare un B-Tree index su una colonna.
2. **Schema Logico (o Concettuale, ma qui 'logico' è il termine usato per il modello del DBMS):**
   - Descrive la struttura dell'intero database usando un modello logico (es. il modello relazionale).
   - È la visione completa di tutte le tabelle, le relazioni tra esse, i vincoli, ecc.
   - È il livello a cui lavorano i DBA e gli sviluppatori backend (es. quando definisci le tabelle con `CREATE TABLE` o usi Prisma per definire i tuoi model).
3. **Schema Esterno (o Viste):**
   - Descrive una porzione del database per specifici utenti o applicazioni.
   - Può essere una 'vista' parziale dei dati (alcune colonne, alcune righe filtrate) o una combinazione di dati da più tabelle.
   - *Esempio SQL:*

```sql
CREATE VIEW StudentiMarketing AS
SELECT Matricola, Nome, Cognome
FROM Studenti
WHERE CorsoLaurea = 'Marketing';
```

Questa vista mostra solo alcuni dati degli studenti di marketing, nascondendo altre informazioni o altri studenti.

1.3.6 Indipendenza dei Dati

È la capacità di modificare la definizione dello schema a un livello senza influenzare la definizione dello schema al livello superiore. È una conseguenza diretta dell'architettura a livelli.

- **Indipendenza Fisica dei Dati:**
  - La rappresentazione logica ed esterna è indipendente da quella fisica.
  - Si possono fare modifiche allo schema interno (es. cambiare algoritmi di accesso, aggiungere indici, spostare file su dischi diversi) senza dover modificare lo schema logico o le applicazioni che interrogano il DB.
  - *Esempio:* Il DBA aggiunge un indice a una tabella per migliorare le prestazioni. Le query SQL delle applicazioni continuano a funzionare come prima, senza modifiche.
- **Indipendenza Logica dei Dati:**
  - Lo schema esterno (viste) è indipendente dallo schema logico.
  - Si possono fare modifiche allo schema logico (es. aggiungere una colonna a una tabella, dividere una tabella in due mantenendo la possibilità di ricostruire l'originale) senza dover modificare le applicazioni che usano le viste, purché le viste possano ancora essere derivate dal nuovo schema logico.
  - È più difficile da ottenere pienamente rispetto a quella fisica.
  - *Esempio:* Se una tabella `Dipendenti` viene divisa in `DipendentiInfoPersonali` e `DipendentiInfoLavorative`, una vista `DipendentiCompleto` che fa il join delle due nuove tabelle può permettere alle vecchie applicazioni di funzionare senza modifiche.

1.3.7 Linguaggi per Database

I DBMS offrono diverse modalità di interazione:

- **Linguaggi testuali 'interattivi' (es. SQL):** L'utente scrive query direttamente in un client.
- **Statement SQL 'iniettati' in un linguaggio ospite (Host Language):** Comandi SQL incorporati in linguaggi di programmazione come Java, C, Python, Node.js.
  - *Esempio Node.js (con `pg` driver per PostgreSQL):*

```
const { rows } = await client.query('SELECT * FROM users WHERE id = $1', [userId]);
```

  - ORM come Prisma astraggono ulteriormente questo, permettendo di scrivere:

```
prisma.user.findUnique({ where: { id: userId } });
```

- **Linguaggi Ad Hoc (es. PL/SQL di Oracle, T-SQL di SQL Server):** Estensioni procedurali di SQL specifiche del DBMS.
- **Interfacce Utente Grafiche (GUI):** Strumenti visuali per interagire con il DB (es. pgAdmin, MySQL Workbench, MongoDB Compass).

1.3.8 DDL e DML (Separazione dati da software)

Due categorie principali di comandi SQL:

- **DDL (Data Definition Language):**
  - Usato per definire e modificare gli **schemi** (logico, esterno, fisico) e altre operazioni sulla struttura.
  - Comandi: `CREATE TABLE`, `ALTER TABLE`, `DROP TABLE`, `CREATE INDEX`, `CREATE VIEW`, `DROP VIEW`.
  - *Esempio:*

```
CREATE TABLE hours (
course CHAR(20),
teacher CHAR(20),
room CHAR(4),
hour CHAR(5)
);
```

- **DML (Data Manipulation Language):**
  - Usato per interrogare e aggiornare le **istanze** (i dati effettivi) del database.
  - Comandi: `SELECT`, `INSERT`, `UPDATE`, `DELETE`.

1.4 Attori e Ruoli

Diverse figure interagiscono con i database:

- **Progettisti e sviluppatori di DBMS:** Creano il software DBMS stesso.
- **Progettisti e sviluppatori di Database:** Disegnano lo schema del database per una specifica applicazione, scrivono query, stored procedure.
- **Amministratori di Database (DBA):**
  - Responsabili del controllo e della gestione centralizzata del database.
  - Si occupano di: efficienza (tuning prestazioni), affidabilità (backup, recovery), sicurezza (gestione permessi), installazione, aggiornamenti.
  - Spesso progettano anche il database, tranne in progetti molto complessi.
- **Progettisti e sviluppatori di applicazioni end-user:** Creano le applicazioni (web, mobile, desktop) che usano il database.
- **Utenti:**
  - **Utenti finali (operatori terminali):** Eseguono operazioni predefinite (transazioni di business, es. un cassiere in un supermercato).
  - **Utenti occasionali/casual:** Eseguono operazioni non definite a priori, usando linguaggi interattivi (es. un analista che esplora i dati con SQL).

1.5 Vantaggi e Svantaggi dei DBMS

1.5.1 Pro (Vantaggi)

- Dati come risorsa condivisa, modellano l'ambiente reale.
- Gestione centralizzata dei dati, standardizzabile e scalabile.
- Fornisce servizi integrati (query, sicurezza, backup, recovery).
- Riduce ridondanze e inconsistente.
- **Indipendenza dei dati:** Supporta lo sviluppo e la gestione delle applicazioni software (le app non devono preoccuparsi di come i dati sono memorizzati fisicamente).

1.5.2 Contro (Svantaggi)

- I prodotti DBMS (specialmente quelli commerciali enterprise) possono essere costosi, così come l'adozione di tali soluzioni (richiede personale specializzato).
- Le funzionalità integrate possono a volte ridurre l'efficienza specifica per compiti molto particolari, rispetto a soluzioni custom altamente ottimizzate (ma questo è un caso limite).

Capitolo 2

Modello Relazionale dei Dati

2.1 Introduzione ai Modelli Logici

I database utilizzano diversi approcci per organizzare logicamente i dati.

- **Modelli Tradizionali:**
  - **Gerarchico:** Struttura ad albero (es. file system). Ogni "figlio" ha un solo "genitore". Navigazione rigida.
  - **Di Rete (Network):** Evoluzione del gerarchico, permette a un "figlio" di avere più "genitori". Più flessibile ma complesso.
  - **Relazionale:** Il modello dominante. Dati organizzati in tabelle.
- **Modelli Più Recenti:**
  - **Object Oriented:** Dati visti come oggetti con proprietà e metodi. Poco comune per DBMS generici.
  - **XML:** Per dati semi-strutturati, spesso complementare al relazionale (es. salvare configurazioni complesse in una cella).

Caratteristica distintiva:

- I modelli Gerarchico e Network usano **riferimenti espliciti (puntatori)** tra record.
- Il modello **Relazionale è "value-based"**: i collegamenti avvengono tramite valori condivisi (es. `userID` in `Posts` che corrisponde a `id` in `Users`).

2.2 Il Modello Relazionale: Fondamenti

- **Definito da E. F. Codd nel 1970:** Obiettivo principale era l'**indipendenza dei dati**, separando la rappresentazione logica dalla memorizzazione fisica.
- *Prisma/ORM Insight:* Un ORM astrae ulteriormente, ma il DBMS relazionale sottostante già opera questa separazione.
- **Implementato nei DBMS reali dal 1981.**
- **Basato sulla definizione logica di "relazione"** (dalla teoria degli insiemi), con differenze pratiche.
- **Le relazioni sono rappresentate tramite tabelle.**

Terminologia Importante:

- **Relazione Logica (Teoria degli Insiemi):** Un sottoinsieme del prodotto cartesiano di due o più insiemi (domini).
- **Relazione (Modello Relazionale):** Una tabella con righe e colonne.
- **Relationship (Modello Entità-Relazione - ER):** Descrive un legame specifico tra entità (es. "uno studente *si iscrive a* un corso").

2.3 Relazioni Logiche vs. Tabelle

2.3.1 Relazione Logica (Matematica)

- Dati due insiemi (domini) $D_1 = \{a, b\}$ e $D_2 = \{x, y, z\}$.
- Il **prodotto cartesiano** $D_1 \times D_2$ è l'insieme di tutte le coppie ordinate possibili: $\{(a, x), (a, y), (a, z), (b, x), (b, y), (b, z)\}$.
- Una **relazione** r è un *sottoinsieme* di questo prodotto cartesiano, es: $r = \{(a, x), (a, z), (b, y)\}$.
- Questo si estende a n domini $D_1, \ldots, D_n$. Una tupla è $(d_1, \ldots, d_n)$.
- **Proprietà di una relazione logica (come insieme):**
  1. **Nessun ordine tra le tuple:** L'ordine delle righe non ha significato.
  2. **Le tuple sono tutte distinte:** Non ci possono essere righe duplicate.
  3. **Ogni n-upla è ordinata:** L'ordine dei valori *all'interno* di una tupla (cioè, l'ordine delle colonne) è significativo.

Esempio Posizionale: `Matches` $\subseteq$ `string` $\times$ `string` $\times$ `int` $\times$ `int`
Una tupla: `(Barca, Bayern, 3, 1)` Qui il significato dipende dalla *posizione*: (SquadraCasa, SquadraOspite, GolCasa, GolOspite).

2.4 Strutture Dati Non Posizionali

Nelle tabelle reali, non ci affidiamo solo alla posizione delle colonne.

- **Ogni colonna ha un nome univoco (attributo)** associato a un dominio (tipo di dato). L'attributo definisce il "ruolo" del dominio.
- Esempio: `Home (string)`, `Away (string)`, `GoalsH (int)`, `GoalsA (int)`.
- **La struttura dati diventa non posizionale:** L'ordine specifico delle colonne nella definizione della tabella è irrilevante per la logica.
- *SQL Insight:* `SELECT Home, Away FROM Matches` e `SELECT Away, Home FROM Matches` accedono agli stessi dati; cambia solo la presentazione.

Una tabella rappresenta una relazione se:

1. Ogni riga può assumere qualsiasi posizione.
2. Ogni colonna può assumere qualsiasi posizione (identificate dal nome).
3. Tutte le righe sono differenti.
4. Tutti i nomi delle colonne (intestazioni) sono differenti.
5. I valori all'interno di una colonna sono omogenei (stesso tipo di dato).

2.5 Il Modello "Value-Based" (Basato su Valori)

Questo è un concetto chiave.

- I riferimenti (collegamenti) tra dati in relazioni (tabelle) diverse sono rappresentati tramite **valori** nelle tuple (righe).
- **Esempio Pratico:**
  - Tabella `STUDENT (Number, Surname, Name, ...)`
  - Tabella `EXAM (Student_ID, Lecture_ID, Grade, ...)`

  - Per collegare un esame a uno studente, `EXAM.Student_ID` conterrà un valore che corrisponde a un valore in `STUDENT.Number`.
- *SQL Insight:* Questo è come funzionano le `FOREIGN KEY` e le clausole `JOIN ... ON table1.column = table2.column`.

Vantaggi della struttura "value-based":

1. Indipendenza dalla struttura fisica dei dati.
2. Memorizzazione solo dei dati rilevanti.
3. Utente e programmatore vedono gli stessi dati.
4. Dati facilmente condivisibili tra ambienti diversi.
5. I collegamenti basati su valori possono essere "navigati" in entrambe le direzioni.

2.6 Definizioni Chiave: Schema, Tupla, Istanza

- **Schema di una Relazione (Tabella):**
  - Nome della relazione seguito dall'elenco dei suoi attributi (colonne).
  - Notazione: $R(A_1, A_2, \ldots, A_n)$
  - Esempio: `STUDENTS (Number, Surname, Name, YearOfBirth)`
  - *SQL Insight:* Corrisponde a `CREATE TABLE STUDENTS (...)`.
- **Schema di un Database:**
  - Insieme degli schemi di tutte le relazioni nel database.
  - Esempio: $R = \{\text{STUDENTS}(\ldots), \text{EXAMS}(\ldots), \text{LECTURES}(\ldots)\}$
  - *Prisma/ORM Insight:* Il tuo file `schema.prisma` definisce lo schema.
- **Tupla (Riga):**
  - Una tupla t su un insieme di attributi X è una mappatura che associa a ogni attributo $A \in X$ un valore dal dominio di A.
  - $t[A]$ esprime il valore della tupla t per l'attributo A.
  - Esempio: Se $t = (6554, \text{Rossi}, \text{Mario}, 1978/12/05)$, allora $t[\text{Name}] = \text{Mario}$.
- **Istanza di una Relazione (Contenuto di una Tabella):**
  - Insieme *finito* di tuple che soddisfano lo schema. Dati attuali in un dato momento.
- **Istanza di un Database Relazionale:**
  - Insieme di istanze di relazione, una per ogni schema. Tutti i dati in tutte le tabelle.

Schema vs. Istanza: Lo schema è la "definizione" (statico), l'istanza sono i "dati reali" (dinamica).

2.7 Gestione di Strutture Dati Annidate

Il modello relazionale classico (Prima Forma Normale - 1NF) richiede valori **atomici**.

- **Esempio:** Una ricevuta con una *lista* di prodotti.

```
Ricevuta 1235, Data 2002/10/12, Totale 39.20
Items:
- 3 x Coperto @ 3.00
- 2 x Antipasto @ 6.20
- ...
```

- **Rappresentazione relazionale (unnesting):** Tabelle separate collegate da chiavi.
  1. Tabella `RECEIPT (Number, Date, Total)`
  2. Tabella `COURSE_ITEM (ReceiptNumber, Qty, Description, Price)`

  `ReceiptNumber` in `COURSE_ITEM` è una chiave esterna.
- **Considerazioni sull' "unnesting":**
  - **Ordine delle righe:** Aggiungere colonna `LineNumber` o `ItemOrder`.
  - **Righe ripetute:** `LineNumber` diventa essenziale per distinguerle.

2.8 Informazioni Parziali e Valori NULL

Spesso i dati sono incompleti.

- **Soluzioni errate per dati mancanti:** Usare valori specifici (0, "", "99").
  - Problemi: Valore "non usato" potrebbe non esistere, o diventare utile; complessità applicativa.
- **Soluzione del Modello Relazionale: Valore** `NULL`
  - `NULL` indica l'**assenza di un valore**. Non è 0, non è stringa vuota.
  - `NULL` **non appartiene al dominio** dell'attributo.
  - Per ogni attributo A, $t[A]$ può mappare a un valore in dom(A) o a `NULL`.
  - Si possono definire vincoli per non ammettere `NULL` (es. `NOT NULL`).
- **Diversi significati di** `NULL` **(concettuali):**
  - Valore sconosciuto.
  - Valore inesistente/non applicabile.
  - Valore non informativo.
- *SQL Insight:* Si interroga con `IS NULL` e `IS NOT NULL`.
- **Troppi** `NULL`**:** Possibile segno di progettazione non ottimale.

2.9 Vincoli di Integrità

Regole che i dati devono rispettare per garantire correttezza e consistenza.

- Un vincolo è una **funzione booleana (predicato)**: per ogni istanza, è vero o falso.
- **Perché usare i vincoli?**
  1. Descrizione accurata dello scenario reale.
  2. Supportano la "qualità dei dati".
  3. Utili nella progettazione del Database.
  4. Usati dal DBMS per l'ottimizzazione delle query.
- **Supporto dei DBMS:**
  - Molti tipi supportati nativamente (`NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK`).
  - Vincoli non supportati devono essere implementati a livello applicativo.

2.9.1 Tipi di Vincoli

1. **Vincoli Intra-relazionali (su una singola tabella):**
    - **Sui valori (o Vincoli di Dominio):** Valori ammissibili per una colonna.
    - Esempio: `Grade` tra 18 e 30.
    - *SQL Insight:* `CHECK (Grade >= 18 AND Grade <= 30)`.
    - **Sulle tuple:** Valori di più colonne *nella stessa riga*.
    - Esempio: `GrossPay = Deductions + Net`.
    - *SQL Insight:* `CHECK (GrossPay = Deductions + Net)`.
2. **Vincoli Inter-relazionali (tra più tabelle):**
    - Integrità referenziale (chiavi esterne).
    - Esempio: `IDStudente` in `ESAMI` deve esistere in `STUDENTI`.
    - *SQL Insight:* `FOREIGN KEY`.

2.9.2 Vincoli di Tupla (e di Dominio)

- **Vincoli di Tupla:** Regole sui valori di ogni tupla, indipendentemente dalle altre.
- **Vincoli di Dominio (caso specifico):** Coinvolgono un singolo attributo.
    - Sintassi: Espressioni booleane che confrontano valori del dominio o espressioni aritmetiche.

2.9.3 Chiavi (Superchiavi, Chiavi Candidate, Chiave Primaria)

Fondamentali per identificare univocamente le tuple e stabilire relazioni.

- **Superchiave (Superkey):**
    - Insieme di attributi K tali che non esistono due tuple distinte t_1, t_2 con $t_1[K] = t_2[K]$.
    - I valori combinati degli attributi in K sono unici per ogni riga.
    - Esempio: In `STUDENTS (Number, ...)`, `{Number}` è superchiave. Anche `{Number, Surname}` lo è. L'insieme di *tutti* gli attributi è sempre una superchiave.
- **Chiave (o Chiave Candidata - Candidate Key):**
    - Una superchiave **minimale** (rimuovendo un attributo, cessa di essere superchiave).
    - Una relazione può avere più chiavi candidate.
- **Vincoli, Schema e Istanze:**
    - Le chiavi sono proprietà dello **schema**, non dedotte da una particolare **istanza**.
- **Esistenza delle Chiavi:**
    - Ogni relazione DEVE avere almeno una chiave.
- **Importanza delle Chiavi:**
    1. Garantiscono identificazione univoca e accessibilità.
    2. Permettono di correlare tuple tra relazioni (modello "value-based").
- **Chiavi e Valori `NULL`:**
    - Attributi parte di una chiave candidata dovrebbero essere `NOT NULL`.
- **Chiave Primaria (Primary Key - PK):**
    - Una chiave candidata scelta come meccanismo principale di identificazione.
    - **NON PUÒ contenere valori** `NULL`.
    - Ogni relazione ha al massimo una PK. Spesso sottolineata.
    - *SQL Insight:* `PRIMARY KEY (attribute_list)` implica `UNIQUE` e `NOT NULL`.

2.9.4 Integrità Referenziale (Chiavi Esterne e Azioni Compensative)

Garantisce coerenza dei collegamenti tra tabelle.

- **Vincolo di Integrità Referenziale (o Chiave Esterna - Foreign Key - FK):**
  - Un insieme di attributi X in R_1 (tabella referenziante) è una FK che referenzia la PK (o una chiave candidata univoca) di R_2 (tabella referenziata) se:
    1. Gli attributi X in R_1 e la PK di R_2 hanno domini compatibili.
    2. Per ogni tupla in R_1, i valori di X devono:
       * Essere `NULL` (se permesso).
       * Oppure, corrispondere a un valore esistente nella PK di una tupla in R_2.
- Esempio:

```
-- Tabella POLICEMAN
-- ID (PK), Surname, Name

-- Tabella INFRINGEMENT
-- Code (PK), Date, Policeman_ID (FK -> POLICEMAN.ID), ...
CREATE TABLE INFRINGEMENT (
Code INT PRIMARY KEY,
EventDate DATE,
Policeman_ID INT,
-- ... altre colonne ...
FOREIGN KEY (Policeman_ID) REFERENCES POLICEMAN(ID)
);
```

- **Chiavi Esterne e `NULL`:**
  - Una FK può contenere `NULL` se la relazione è opzionale.
  - Esempio: `EMPLOYEE (ID, Name, Project_Code)`. Se `Project_Code` è FK, un impiegato può avere `Project_Code = NULL`.
- **Azioni Compensative (se si viola l'integrità referenziale):**
  - Azioni su DELETE/UPDATE sulla tabella referenziata (R_2):
    1. `RESTRICT` **(o** `NO ACTION` **- default):** Operazione rifiutata.
    2. `CASCADE`**:**
       * `ON DELETE CASCADE`: Elimina righe referenzianti in R_1.
       * `ON UPDATE CASCADE`: Aggiorna valori FK in R_1.
    3. `SET NULL`**:**
       * `ON DELETE SET NULL`: Imposta FK in R_1 a `NULL`.
       * `ON UPDATE SET NULL`: (simile).
    4. `SET DEFAULT`**:**
       * `ON DELETE SET DEFAULT`: Imposta FK in R_1 al valore di default.
- *SQL Insight:*

```
FOREIGN KEY (Project_Code) REFERENCES PROJECT(Code)
ON DELETE SET NULL
ON UPDATE CASCADE;
```

- **Vincoli su Attributi Multipli (Chiavi Composte):**
  - PK o FK possono essere composte da più attributi.
  - L'ordine degli attributi nella definizione della FK deve corrispondere a quello della PK referenziata.

Capitolo 3

Algebra Relazionale e Calcolo Relazionale

3.1 Introduzione ai Linguaggi per Database

I linguaggi per database si dividono principalmente in due categorie:

- **DDL (Data Definition Language):** Utilizzato per definire e modificare lo schema del database (es. creare tabelle, definire attributi e tipi).
- **DML (Data Manipulation Language):** Utilizzato per operare sui dati. Si suddivide ulteriormente in:
  - **Istruzioni di Query:** Per estrarre dati di interesse.
  - **Istruzioni di Aggiornamento:** Per inserire nuovi dati o modificare quelli esistenti.

I linguaggi di query possono essere:

- **Dichiarativi:** Specificano **cosa** si vuole ottenere, le proprietà del risultato. L'utente non si preoccupa di come il database recupererà i dati. SQL è prevalentemente dichiarativo.
- **Imperativi/Procedurali:** Specificano **come** il risultato deve essere ottenuto, descrivendo una sequenza di operazioni.

Panoramica dei linguaggi trattati:

- **Algebra Relazionale:** Procedurale (fondamento teorico).
- **Calcolo Relazionale:** Dichiarativo (fondamento teorico).
- **SQL (Structured Query Language):** Parzialmente dichiarativo (ampiamente implementato).
- **QBE (Query by Example):** Dichiarativo (implementato in alcuni sistemi).

3.2 Algebra Relazionale

L'algebra relazionale è un linguaggio di query formale, procedurale, che definisce un insieme di operatori che agiscono su relazioni (tabelle) per produrre nuove relazioni come risultato. Gli operatori possono essere composti.

3.2.1 Operatori dell'Algebra Relazionale

- Operatori insiemistici: **Unione ($\cup$)**, **Intersezione ($\cap$)**, **Differenza ($-$)**.
- **Ridenominazione ($\rho()$)**.
- **Selezione ($\sigma()$)**.
- **Proiezione ($\pi()$)**.
- **Join ($\bowtie$)**: natural join, prodotto cartesiano ($\times$), theta-join ($\bowtie_\theta$).

3.2.2 Operatori Insiemistici

Le relazioni sono insiemi di tuple. Questi operatori funzionano come le loro controparti nella teoria degli insiemi, ma richiedono che le relazioni coinvolte abbiano lo **stesso schema** (stessi nomi di attributi, nello stesso ordine e con tipi compatibili).

- **Unione (*R1* $\cup$ *R2*):** Restituisce una relazione contenente tutte le tuple che sono in *R1*, in *R2*, o in entrambe. Le tuple duplicate vengono eliminate.
- **Intersezione (*R1* $\cap$ *R2*):** Restituisce una relazione contenente solo le tuple che sono presenti sia in *R1* sia in *R2*.
- **Differenza (*R1* $-$ *R2*):** Restituisce una relazione contenente le tuple che sono in *R1* ma non in *R2*.

Esempio pratico: Se hai due tabelle, *StudentiMagistrale* e *StudentiDottorato*, entrambe con colonne IDStudente e Nome, puoi fare l'unione per ottenere una lista unica di tutti gli studenti post-laurea. Se le colonne avessero nomi diversi (es. Matricola vs IDStudente), dovresti prima usare l'operatore di ridenominazione.

3.2.3 Ridenominazione ($\rho_{nuovo \leftarrow vecchio}(R)$)

Operatore unario che cambia i nomi degli attributi o della relazione stessa, senza alterare i dati.

- $\rho_{\text{NuovoNomeAttr} \leftarrow \text{VecchioNomeAttr}}((R))$: Rinomina l'attributo VecchioNomeAttr in NuovoNomeAttr nella relazione *R*.
- $\rho_{\textit{NuovoNomeRel}}((R))$: Rinomina la relazione *R* in *NuovoNomeRel*.
- $\rho_{\textit{NuovoNomeRel}(\text{A1},\text{A2},\dots)}((R))$: Rinomina la relazione e i suoi attributi.

Esempio pratico: Se hai una tabella *Impiegati* con una colonna stip e vuoi renderla più chiara come StipendioAnnuale, useresti $\rho_{\text{StipendioAnnuale} \leftarrow \text{stip}}((\textit{Impiegati}))$.

3.2.4 Selezione ($\sigma_{predicato}(R)$)

Operatore unario che restituisce un sottoinsieme delle tuple di una relazione *R* che soddisfano un dato *predicato* (condizione). Lo schema del risultato è identico a quello di *R*. **Esempio pratico (SQL):** $\sigma_{\text{Eta} > 30 \wedge \text{Dipartimento} = \text{'IT'}}((\textit{Impiegati}))$ è equivalente a:

```sql
SELECT *
FROM Impiegati
WHERE Eta > 30 AND Dipartimento = 'IT';
```

3.2.5 Proiezione ($\pi_{lista\_attributi}(R)$)

Operatore unario che restituisce una nuova relazione contenente solo gli attributi specificati nella *lista_attributi* dalla relazione *R*. Le tuple duplicate nel risultato vengono eliminate (poiché le relazioni sono insiemi). **Esempio pratico (SQL):** $\pi_{\text{Nome},\text{Cognome}}((\textit{Impiegati}))$ è equivalente a:

```sql
SELECT DISTINCT Nome, Cognome
FROM Impiegati;
```

Nota l'uso di 'DISTINCT' in SQL per replicare il comportamento insiemistico della proiezione.

3.2.6 Combinazione di Selezione e Proiezione

Questi operatori sono spesso usati insieme per estrarre dati specifici. **Esempio pratico (SQL):** Trovare nome e cognome degli impiegati nel dipartimento 'Vendite' con stipendio superiore a 50000. Algebra: $\pi_{\text{Nome},\text{Cognome}}((\sigma_{\text{Dipartimento} = \text{'Vendite'} \wedge \text{Stipendio} > 50000}((\textit{Impiegati}))))$ SQL:

```sql
SELECT DISTINCT Nome, Cognome
FROM Impiegati
WHERE Dipartimento = 'Vendite' AND Stipendio > 50000;
```

3.2.7 Join

Il join è un operatore fondamentale per combinare informazioni da due o più relazioni.

- **Prodotto Cartesiano (*R1* $\times$ *R2*):** Combina ogni tupla di *R1* con ogni tupla di *R2*. Il numero di tuple risultanti è $|R1| \times |R2|$. Lo schema è la concatenazione degli schemi di *R1* e *R2*. In SQL, è spesso scritto come 'FROM R1, R2' (sintassi più vecchia) o 'FROM R1 CROSS JOIN R2'. Di solito è seguito da una selezione per filtrare le combinazioni significative.
- **Theta-Join (*R1* $\bowtie_{condizione}$ *R2*):** È un prodotto cartesiano seguito da una selezione. La *condizione* è un predicato che coinvolge attributi di *R1* e *R2*. Sintassi formale: $\sigma_{condizione}((R1 \times R2))$. **Esempio pratico (SQL):** *Impiegati* $\bowtie_{\text{Impiegati.IDDip=Dipartimenti.ID}}$ *Dipartimenti* è equivalente a:

  ```sql
  SELECT *
  FROM Impiegati, Dipartimenti -- o Impiegati JOIN Dipartimenti
  WHERE Impiegati.IDDip = Dipartimenti.ID;
  ```

- **Equi-Join:** Un Theta-Join in cui la condizione contiene solo confronti di uguaglianza ($=$). L'esempio precedente è un equi-join.
- **Natural Join (*R1* $\bowtie$ *R2*):** Un tipo speciale di equi-join. Le relazioni vengono combinate basandosi sull'uguaglianza dei valori degli attributi che hanno lo **stesso nome** in entrambe le relazioni. Gli attributi comuni appaiono una sola volta nel risultato. **Esempio pratico (SQL):** Se *Impiegati* e *Assegnazioni* hanno entrambe una colonna IDProgetto. *Impiegati* $\bowtie$ *Assegnazioni* è equivalente a:

  ```sql
  SELECT *
  FROM Impiegati NATURAL JOIN Assegnazioni;
  ```

  Attenzione: Il Natural Join può essere pericoloso se ci sono attributi con lo stesso nome ma significato diverso, o se si aggiungono/rimuovono colonne. È spesso preferibile usare join espliciti con clausola 'ON'.

3.2.8 Outer Join (Join Esterni)

I join visti finora (inner join) scartano le tuple che non trovano una corrispondenza nell'altra relazione. Gli outer join includono queste tuple, riempiendo con 'NULL' gli attributi mancanti.

- **Left Outer Join (*R1* $\bowtie$ *R2*):** Mantiene tutte le tuple di *R1*. Se una tupla di *R1* non ha corrispondenze in *R2*, viene inclusa nel risultato con valori 'NULL' per gli attributi di *R2*. SQL: 'R1 LEFT OUTER JOIN R2 ON condizione'
- **Right Outer Join (*R1* $\bowtie$ *R2*):** Mantiene tutte le tuple di *R2*. Simmetrico al left outer join. SQL: 'R1 RIGHT OUTER JOIN R2 ON condizione'
- **Full Outer Join (*R1* $\bowtie$ *R2*):** Mantiene tutte le tuple di entrambe le relazioni. Se non c'è corrispondenza, i campi dell'altra relazione sono 'NULL'. SQL: 'R1 FULL OUTER JOIN R2 ON condizione'

Esempio pratico: Trovare tutti gli impiegati e, se assegnati a un dipartimento, il nome del dipartimento. Se un impiegato non ha dipartimento, vogliamo comunque vederlo. *Impiegati* $\bowtie$ *Dipartimenti* (assumendo un join su IDDip) SQL:

```sql
SELECT Impiegati.Nome, Dipartimenti.NomeDip
FROM Impiegati
LEFT OUTER JOIN Dipartimenti ON Impiegati.IDDip = Dipartimenti.ID;
```

3.2.9 Espressioni Equivalenti e Ottimizzazione

Esistono diverse espressioni algebriche che producono lo stesso risultato. I DBMS (Database Management Systems) utilizzano regole di equivalenza per trasformare una query in una forma equivalente ma più efficiente da eseguire. Ad esempio, "spingere" le selezioni il più presto possibile ('pushdown selection') riduce la dimensione delle relazioni intermedie, velocizzando i join successivi. **Esempio:** $\sigma_{\text{Stipendio}>100K}((\textit{Impiegati} \bowtie \textit{Dipartimenti}))$ potrebbe essere più efficiente se riscritta come: $(\sigma_{\text{Stipendio}>100K}(\textit{Impiegati})) \bowtie \textit{Dipartimenti}$ (se Stipendio è solo in *Impiegati*). Il DBMS fa queste ottimizzazioni automaticamente.

3.2.10 Selezione con Valori NULL

I valori 'NULL' rappresentano l'assenza di un valore o un valore sconosciuto. Nei predicati di selezione:

- Un confronto con 'NULL' (es. Eta $>$ NULL o Stipendio $=$ NULL) restituisce 'UNKNOWN'.
- L'operatore σ seleziona solo le tuple per cui il predicato è 'TRUE'.
- Per testare esplicitamente i 'NULL', si usano i predicati Attributo IS NULL e Attributo IS NOT NULL.

Esempio pratico (SQL): Trovare gli impiegati senza un numero di telefono assegnato. $\sigma_{\text{Telefono IS NULL}}((\textit{Impiegati}))$ SQL:

```sql
SELECT * FROM Impiegati WHERE Telefono IS NULL;
```

3.3 Views (Viste)

Una vista è una tabella virtuale il cui contenuto è definito da una query sull'algebra relazionale (o SQL). Non memorizza dati propriamente, ma li deriva dalle tabelle base al momento della query sulla vista.

- **Tabelle Base:** Tabelle che contengono fisicamente i dati.
- **Tabelle Derivate (Viste):** Definite da query.

Le viste sono utili per:

- **Schema Esterno:** Fornire diverse rappresentazioni dei dati a utenti diversi, semplificando la complessità e implementando la sicurezza (mostrando solo dati pertinenti).
- **Strumento di Programmazione:** Semplificare query complesse riutilizzando sotto-espressioni comuni, o per mantenere la compatibilità con applicazioni esistenti quando lo schema delle tabelle base cambia.

Esempio pratico (SQL): Creare una vista che mostra solo gli impiegati del dipartimento IT. Algebra: $\textit{ImpiegatiIT} := \sigma_{\text{Dipartimento}='\text{IT}'}((\textit{Impiegati}))$ SQL:

```sql
CREATE VIEW ImpiegatiIT AS
SELECT *
FROM Impiegati
WHERE Dipartimento = 'IT';

-- Successivamente si può interrogare la vista:
SELECT Nome, Cognome FROM ImpiegatiIT WHERE Stipendio > 60000;
```

Il DBMS traduce la query sulla vista in una query sulle tabelle base (es. $\sigma_{\text{Stipendio}>60000}((\sigma_{\text{Dipartimento}='\text{IT}'}((\textit{Impiegati}))))$).

3.4 Esempi Completi di Algebra Relazionale

In questa sezione presentiamo esempi completi di espressioni in algebra relazionale, simili a quelli che potrebbero apparire negli esercizi d'esame. Utilizzeremo uno schema di database di esempio relativo a concerti ed eventi musicali.

3.4.1 Schema di Esempio

PERSONA	
IDPersona	Chiave primaria
Nome	Nome della persona
Cognome	Cognome della persona
DataNascita	Data di nascita

CONCERTO	
IDConcerto	Chiave primaria
Titolo	Titolo del concerto
DataEvento	Data del concerto
Genere	Genere musicale
Citta	Città dove si svolge

BIGLIETTO	
IDBiglietto	Chiave primaria
IDConcerto	Chiave esterna $\rightarrow$ CONCERTO
IDPersona	Chiave esterna $\rightarrow$ PERSONA
Prezzo	Prezzo del biglietto
Settore	Settore del posto

ARTISTA	
IDArtista	Chiave primaria
Nome	Nome dell'artista/band
Nazionalita	Nazionalità dell'artista

PARTECIPAZIONE	
IDArtista	Chiave esterna $\rightarrow$ ARTISTA
IDConcerto	Chiave esterna $\rightarrow$ CONCERTO
Cachet	Compenso dell'artista

Tabella 3.1: Schema del database dei concerti

3.4.2 Esempi di Query in Algebra Relazionale

Esempio 1: Persone che hanno acquistato biglietti per concerti pop a Berlino

Vogliamo trovare i nominativi (nome e cognome) delle persone che hanno acquistato biglietti per concerti pop tenutisi a Berlino.

Step 1: Selezionare i concerti pop a Berlino

$$\text{ConcertiPopBerlino} \leftarrow \sigma_{\text{Genere='Pop'} \wedge \text{Citta='Berlino'}}((\textit{CONCERTO}))$$

Step 2: Trovare i biglietti per questi concerti

$$\text{BigliettiPopBerlino} \leftarrow \textit{BIGLIETTO} \bowtie \text{ConcertiPopBerlino}$$

Step 3: Trovare le persone che hanno acquistato questi biglietti

$$\text{PersonePopBerlino} \leftarrow \text{BigliettiPopBerlino} \bowtie \textit{PERSONA}$$

Step 4: Proiettare solo nome e cognome

$$\text{Risultato} \leftarrow \pi_{\text{Nome,Cognome}}((\text{PersonePopBerlino}))$$

Oppure, in un'unica espressione compatta:

$$\text{Risultato} \leftarrow \pi_{\text{Nome,Cognome}}((\textit{PERSONA} \bowtie (\textit{BIGLIETTO} \bowtie \sigma_{\text{Genere='Pop'} \wedge \text{Citta='Berlino'}}((\textit{CONCERTO})))))$$

Esempio 2: Città che non hanno mai ospitato un concerto rock

Vogliamo trovare i nomi delle città che non hanno mai ospitato un concerto rock. Per questo abbiamo bisogno di una tabella aggiuntiva *CITTA* con tutte le città.

Step 1: Trovare le città che hanno ospitato concerti rock

$$\text{CittaRock} \leftarrow \pi_{\text{Citta}}((\sigma_{\text{Genere='Rock'}}((\textit{CONCERTO}))))$$

Step 2: Trovare le città che non sono in CittaRock

$$\text{Risultato} \leftarrow \pi_{\text{Nome}}((\textit{CITTA})) - \text{CittaRock}$$

Alternativamente, assumendo che non abbiamo una tabella *CITTA* separata, possiamo usare la differenza con l'insieme di tutte le città presenti nella tabella *CONCERTO*:

$$\text{Risultato} \leftarrow \pi_{\text{Citta}}((\textit{CONCERTO})) - \pi_{\text{Citta}}((\sigma_{\text{Genere='Rock'}}((\textit{CONCERTO}))))$$

Esempio 3: Artisti che hanno partecipato a tutti i concerti in Italia

Questo è un esempio di divisione relazionale (implementata con differenza e proiezione).

Step 1: Trovare tutti i concerti in Italia

$$\text{ConcertiItalia} \leftarrow \pi_{\text{IDConcerto}}((\sigma_{\text{Citta in }\{\text{'Roma','Milano','Napoli',}\ldots\}}((\textit{CONCERTO}))))$$

Step 2: Per ogni artista, trovare le coppie (artista, concerto) che NON esistono in PARTECIPAZIONE

$$\text{TutteCombinazioniPossibili} \leftarrow \pi_{\text{IDArtista}}((\textit{ARTISTA})) \times \text{ConcertiItalia}$$

$$\text{CombinazioniMancanti} \leftarrow \text{TutteCombinazioniPossibili} - \pi_{\text{IDArtista,IDConcerto}}((\textit{PARTECIPAZIONE}))$$

Step 3: Trovare gli artisti che non hanno combinazioni mancanti (hanno partecipato a tutti i concerti)

$$\text{ArtistiConMancanze} \leftarrow \pi_{\text{IDArtista}}((\text{CombinazioniMancanti}))$$

$$\text{Risultato} \leftarrow \pi_{\text{IDArtista,Nome}}((\textit{ARTISTA})) - \pi_{\text{IDArtista,Nome}}((\textit{ARTISTA} \bowtie \text{ArtistiConMancanze}))$$

Esempio 4: Persone che hanno acquistato biglietti per tutti gli artisti di nazionalità francese

Step 1: Trovare tutti gli artisti francesi
ArtistiFrancesi $\leftarrow \pi_{\text{IDArtista}}((\sigma_{\text{Nazionalita='Francese'}}((\textit{ARTISTA}))))$

Step 2: Trovare i concerti dove si esibiscono artisti francesi
ConcertiFrancesi $\leftarrow \pi_{\text{IDConcerto}}((\textit{PARTECIPAZIONE} \bowtie \text{ArtistiFrancesi}))$

Step 3: Trovare tutte le combinazioni possibili (persona, artista francese)
TutteCombinazioniPossibili $\leftarrow \pi_{\text{IDPersona}}((\textit{PERSONA})) \times \text{ArtistiFrancesi}$

Step 4: Trovare le persone che hanno visto ciascun artista francese
PersoneArtistiFrancesi $\leftarrow \pi_{\text{IDPersona,IDArtista}}((\textit{PERSONA} \bowtie \textit{BIGLIETTO} \bowtie \textit{CONCERTO} \bowtie \textit{PARTECIPAZIONE} \bowtie \sigma_{\text{Nazionalita='Fra}}$

Step 5: Trovare le combinazioni mancanti
CombinazioniMancanti $\leftarrow$ TutteCombinazioniPossibili $-$ PersoneArtistiFrancesi

Step 6: Trovare le persone senza combinazioni mancanti
PersoneConMancanze $\leftarrow \pi_{\text{IDPersona}}((\text{CombinazioniMancanti}))$
Risultato $\leftarrow \pi_{\text{Nome,Cognome}}((\textit{PERSONA})) - \pi_{\text{Nome,Cognome}}((\textit{PERSONA} \bowtie \text{PersoneConMancanze}))$

Esempio 5: Concerti con prezzo medio dei biglietti superiore alla media generale

Nota: Questo esempio illustra un limite dell'algebra relazionale standard, che non supporta funzioni di aggregazione. In pratica, si userebbero estensioni dell'algebra relazionale o SQL.

In SQL questa query sarebbe:

```
SELECT C.IDConcerto, C.Titolo, AVG(B.Prezzo) as PrezzoMedio
FROM CONCERTO C JOIN BIGLIETTO B ON C.IDConcerto = B.IDConcerto
GROUP BY C.IDConcerto, C.Titolo
HAVING AVG(B.Prezzo) > (SELECT AVG(Prezzo) FROM BIGLIETTO)
```

Quest'ultimo esempio evidenzia perché SQL, con il suo supporto per aggregazioni e costrutti come GROUP BY e HAVING, è spesso più conveniente dell'algebra relazionale pura per query complesse.

3.5 Calcolo Relazionale

Il calcolo relazionale è un linguaggio di query formale, **dichiarativo**, basato sulla logica dei predicati del primo ordine. Specifica *cosa* si vuole ottenere, non *come*. Esistono due forme principali:

- **Domain Relational Calculus (DRC):** Le variabili assumono valori dai domini degli attributi.
- **Tuple Relational Calculus (TRC):** Le variabili rappresentano tuple di relazioni.

3.5.1 Domain Relational Calculus (DRC)

Una query DRC ha la forma: $\{\text{A}_1 : x_1, \ldots, \text{A}_k : x_k \mid \text{Formula}(x_1, \ldots, x_k)\}$ dove:

- $\text{A}_1 : x_1, \ldots, \text{A}_k : x_k$ è la **target list**: specifica gli attributi del risultato e le variabili che ne conterranno i valori.
- $x_1, \ldots, x_k$ sono variabili che variano sui domini dei rispettivi attributi.

- Formula$(x_1, \ldots, x_k)$ è una formula della logica del primo ordine che usa:
  - Predicati corrispondenti alle relazioni nel database (es. *IMPIEGATO*(Num : m, Nome : $n, \ldots$)).
  - Operatori di confronto ($=, >, <, \ldots$).
  - Operatori logici ($\wedge$ AND, $\vee$ OR, $\neg$ NOT).
  - Quantificatori ($\forall$ per tutti, $\exists$ esiste).

Il risultato è l'insieme di tuple $(\mathsf{A}_1 : v_1, \ldots, \mathsf{A}_k : v_k)$ tali che, quando le variabili x_i assumono i valori v_i, la Formula è vera.

Esempio DRC: Trovare numero, nome, età e salario degli impiegati che guadagnano più di 40.

```
{ Numero:m, Nome:n, Eta:a, Salario:w |
 IMPIEGATO(Numero:m, Nome:n, Eta:a, Salario:w)  w > 40 }
```

Se vogliamo solo nome ed età:

```
{ Nome:n, Eta:a |
 existsop w (IMPIEGATO(Numero:m, Nome:n, Eta:a, Salario:w)  w > 40) }
```

(Il 'Numero:m' nella seconda query è una variabile libera nella formula 'IMPIEGATO', ma non nella target list, quindi la sua esistenza è implicitamente richiesta. Le slide mostrano 'EMPLOYEE(Number:m, Name:n, Age:a, Wage:w)' anche quando 'm' non è nella target list; questo significa che deve esistere una tupla con qualche 'm' che soddisfi il resto. Per essere più precisi, si quantificherebbero le variabili non nella target list).

3.5.2 Tuple Relational Calculus (TRC) with Range Declarations

Una query TRC ha la forma: {TargetList | RangeList | Formula} dove:

- TargetList: Specifica gli attributi da restituire, spesso nella forma t.A (attributo A della tupla t).
- RangeList: Dichiara le variabili di tupla e le relazioni a cui appartengono (es. e(*IMPIEGATO*), s(*SUPERVISORE*)).
- Formula: Una condizione sulle variabili di tupla dichiarate.

Le variabili variano sull'insieme delle tuple della relazione specificata.

Esempio TRC: Trovare tutte le informazioni sugli impiegati che guadagnano più di 40.

```
{ e.* | e(IMPIEGATO) | e.Salario > 40 }
```

Trovare nome ed età degli impiegati che guadagnano più di 40:

```
{ e.Nome, e.Eta | e(IMPIEGATO) | e.Salario > 40 }
```

Il TRC è spesso considerato più vicino a come si pensa in SQL, poiché si ragiona in termini di "tuple che soddisfano certe condizioni".

3.5.3 Equivalenza tra Algebra e Calcolo

Per le query "safe" (che non producono risultati infiniti e dipendono solo dai dati nel database), l'Algebra Relazionale, il Domain Relational Calculus (safe) e il Tuple Relational Calculus (safe) sono **espressivamente equivalenti**. Ciò significa che qualsiasi query esprimibile in uno di questi linguaggi può essere espressa anche negli altri. Questo è un risultato teorico importante (Teorema di Codd).

3.6 Limiti dell'Algebra e del Calcolo Relazionale Standard

Nonostante la loro potenza, l'algebra e il calcolo relazionale standard hanno dei limiti:

- **No Calcoli Aritmetici/Nuovi Valori:** Non possono calcolare nuovi valori (es. stipendio + bonus) o eseguire aggregazioni (somma, media, conteggio). SQL estende queste capacità con funzioni aritmetiche e di aggregazione ('SUM()', 'AVG()', 'COUNT()', 'GROUP BY').
- **No Chiusura Transitiva (Recursion):** Non possono esprimere query ricorsive, come trovare tutti i superiori di un impiegato (il capo, il capo del capo, ecc.) o tutte le tratte aeree possibili tra due città (dirette e indirette). Questo richiederebbe un numero potenzialmente illimitato di join.

3.7 Datalog

Datalog è un linguaggio di query e programmazione logica orientato ai database, che supera alcuni limiti di RA/RC, in particolare per le query ricorsive. È un sottoinsieme di Prolog. Concetti chiave:

- **Predicati Estensionali (EDB - Extensional Database):** Corrispondono alle relazioni base del database (fatti).
- **Predicati Intensionali (IDB - Intensional Database):** Corrispondono a viste o relazioni derivate, definite tramite **regole**.
- **Regole:** Hanno la forma 'testa :- corpo.' (o 'testa <- corpo' nelle slide). Significa: "la *testa* è vera se il *corpo* è vero". La testa è un singolo predicato intensionale. Il corpo è una congiunzione (AND) di predicati (estensionali o intensionali) e condizioni.
- **Query:** Indicate con un prefisso ? davanti a un predicato.

Esempio Datalog (non ricorsivo): Trovare i capi degli impiegati che guadagnano più di 40. Relazioni EDB: *IMPIEGATO*(Num, Nome, Eta, Salario), *SUPERVISORE*(Capo, Impiegato)

```
% Predicato intensionale: IMPIEGATO_RICCO
IMPIEGATO_RICCO(Num, Nome, Eta, Salario) :- IMPIEGATO(Num, Nome, Eta, Salario), Salario > 40.

% Predicato intensionale: CAPO_DI_IMPIEGATO_RICCO
CAPO_DI_IMPIEGATO_RICCO(NumCapo) :- IMPIEGATO_RICCO(NumImp, _, _, _),
SUPERVISORE(NumCapo, NumImp).

% Query
? CAPO_DI_IMPIEGATO_RICCO(X).
```

Esempio Datalog (Ricorsione - Chiusura Transitiva): Trovare tutti i superiori (diretti e indiretti) di un impiegato. Relazione EDB: *CAPO_DIRETTO*(Superiore, Subordinato)

```
% Caso base: un capo diretto e' un superiore
SUPERIORE(X, Y) :- CAPO_DIRETTO(X, Y).

% Caso ricorsivo: il superiore di un mio superiore e' anche mio superiore
SUPERIORE(X, Y) :- CAPO_DIRETTO(X, Z), SUPERIORE(Z, Y).

% Query: trovare tutti i superiori di 'Rossi' (assumendo che 'Rossi' sia un ID)
? SUPERIORE(Capo, 'Rossi').
```

Datalog, grazie alla sua capacità di esprimere la ricorsione, è più potente dell'algebra e del calcolo relazionale standard. Le estensioni ricorsive di SQL (come 'WITH RECURSIVE') sono ispirate a Datalog.

3.8 Conclusioni

L'Algebra Relazionale e il Calcolo Relazionale forniscono le fondamenta teoriche per i linguaggi di query dei database relazionali come SQL.

- L'**Algebra Relazionale** è procedurale e definisce come costruire il risultato passo dopo passo. È cruciale per l'implementazione interna e l'ottimizzazione delle query nei DBMS.
- Il **Calcolo Relazionale** è dichiarativo, permettendo di specificare le proprietà del risultato desiderato senza dettagliarne il processo di ottenimento.
- Entrambi (nelle loro forme "safe") hanno lo stesso potere espressivo ma non possono gestire calcoli complessi o ricorsione.
- **Datalog** estende questi concetti introducendo la ricorsione, aumentando il potere espressivo.

Comprendere questi modelli teorici aiuta a capire meglio il funzionamento e le potenzialità di SQL e dei sistemi di gestione di database moderni.

Capitolo 4

SQL Base

4.1 Introduzione a SQL

4.1.1 Cos'è SQL

- Acronimo di "Structured Query Language", oggi considerato un "nome proprio".

4.1.2 Caratteristiche Principali

- Implementa sia **DDL (Data Definition Language)**: comandi per definire la struttura del database (tabelle, schemi, indici, ecc.).
- Implementa sia **DML (Data Manipulation Language)**: comandi per interrogare e modificare i dati.

4.1.3 Standard vs. Dialetti

- Esiste uno standard ISO, ma ogni DBMS (PostgreSQL, MySQL, SQL Server, Oracle, SQLite) ha le sue piccole variazioni ed estensioni grammaticali.
- *Esempio Pratico:* La sintassi per l'auto-incremento di un ID può variare (`SERIAL` in PostgreSQL, `AUTO_INCREMENT` in MySQL).

4.1.4 Storia

- Predecessore: SEQUEL (1974).
- Prime implementazioni: SQL/DS e Oracle (1981).
- Standard "de facto" dal 1983, con molte evoluzioni (SQL-86, SQL-89, SQL-92, SQL:1999, ecc.) che hanno introdotto:
  - Integrità referenziale (SQL-89)
  - Funzioni come `COALESCE`, `NULLIF`, `CASE` (SQL-92)
  - Concetti object-relational, trigger, funzioni esterne (SQL:1999)
  - Supporto per Java e XML (SQL:2003, SQL:2006)

4.2 DDL (Data Definition Language) - Definire la Struttura

4.2.1 Database e Schemi

- `CREATE DATABASE db_name;`
  - Crea un nuovo database, che è un contenitore per tabelle, viste, trigger, ecc.

  - *Esempio Pratico (SQLite):* Quando esegui `sqlite3 miodatabase.db`, stai creando un file che funge da database.
  - *Nota:* In alcuni DBMS come MySQL, `CREATE SCHEMA` è un sinonimo di `CREATE DATABASE`.
- `CREATE SCHEMA schema_name [AUTHORIZATION 'user_name'];`
  - Uno schema è uno spazio dei nomi all'interno di un database. Serve a organizzare gli oggetti del database.
  - L'utente che esegue il comando diventa il proprietario dello schema, a meno che non sia specificato con `AUTHORIZATION`.
  - *Esempio Pratico (PostgreSQL):* Spesso si usa lo schema `public` di default, ma potresti creare schemi come `accounting`, `sales` per separare logicamente le tabelle.

4.2.2 Tabelle

- Sintassi base:

```
CREATE TABLE table_name (
colonna1 TIPO_DATI [vincoli],
colonna2 TIPO_DATI [vincoli],
...
);
```

- Definisce una nuova tabella (relazione) con le sue colonne (attributi), i tipi di dato per ciascuna colonna e i vincoli iniziali.
- *Esempio Pratico (corrispettivo User con Prisma):*

```
// Prisma Schema
model User {
id        Int      @id @default(autoincrement())
email     String   @unique
name      String?
}
```

Equivalente a:

```
-- SQL (es. PostgreSQL)
CREATE TABLE User (
id SERIAL PRIMARY KEY,
email VARCHAR(255) UNIQUE NOT NULL,
name VARCHAR(255)
);
```

- Esempio dalla slide:

```
CREATE TABLE EMPLOYEE (
Number CHARACTER(6) PRIMARY KEY,
Name CHARACTER(20) NOT NULL,
Surname CHARACTER(20) NOT NULL,
Dept CHARACTER(15),
Wage NUMERIC(9) DEFAULT 0,
FOREIGN KEY(Dept) REFERENCES DEPARTMENT(Dept)
);
```

4.2.3 Tipi di Dati

Tipi Base

- `CHARACTER(n)`, `VARCHAR(n)`: Stringhe di caratteri a lunghezza fissa o variabile.
- `NUMERIC(p,s)`, `INTEGER`, `SMALLINT`, `DECIMAL`: Numeri interi o decimali.
- `DATE`, `TIME`, `TIMESTAMP`, `INTERVAL`: Per date e orari.
- `BOOLEAN`: Valori vero/falso.
- `BLOB`, `CLOB`: (Binary/Character Large Object) Per grandi quantità di dati binari o testuali.

Tipi Personalizzati (Domini)

- `CREATE DOMAIN domain_name AS tipo_base [DEFAULT valore_default] [CHECK (condizione)];`
- Permette di definire un tipo di dato riutilizzabile con vincoli e valori di default specifici.
- Esempio dalla slide:

```sql
CREATE DOMAIN Grade
AS SMALLINT DEFAULT NULL
CHECK (value >= 18 AND value <= 30);
```

  Questo `Grade` può poi essere usato come tipo di dato per una colonna.

4.2.4 Vincoli (Constraints)

Servono a garantire l'integrità e la coerenza dei dati.

Vincoli comuni

- `NOT NULL`: La colonna non può contenere valori `NULL`.
- `UNIQUE`: I valori nella colonna (o combinazione di colonne) devono essere unici.
- `PRIMARY KEY`:
  - Identifica univocamente ogni riga. Implica `NOT NULL` e `UNIQUE`.
  - Solo una per tabella. Può essere su colonna singola o multipla.
  - Esempio (inline): `Number CHARACTER(6) PRIMARY KEY`
  - Esempio (standalone): `PRIMARY KEY (Number)`
- **Attenzione (Slide 23):**
  - `UNIQUE (Surname, Name)`: La *combinazione* di cognome e nome deve essere unica.
  - `Surname CHARACTER(20) UNIQUE, Name CHARACTER(20) UNIQUE`: Il cognome deve essere unico **e** il nome deve essere unico (indipendentemente).

`FOREIGN KEY` e Integrità Referenziale

- `FOREIGN KEY (colonna_fk) REFERENCES tabella_riferita (colonna_pk_riferita)`
- Garantisce che i valori nella `colonna_fk` esistano nella `colonna_pk_riferita` della `tabella_riferita`.
- *Esempio Pratico (Relazione Post-User con Prisma):*

```
// Prisma Schema
model User {
id    Int     @id @default(autoincrement())
posts Post[]
}
model Post {
id        Int     @id @default(autoincrement())
author    User    @relation(fields: [authorId], references: [id])
authorId  Int // Foreign Key
}
```

Azioni Referenziali Triggerate

Cosa succede se un record referenziato viene cancellato o aggiornato: `ON DELETE | ON UPDATE`

- `CASCADE`: Propaga l'azione (es. se cancello un utente, cancella anche i suoi post).
- `SET NULL`: Imposta la foreign key a `NULL`.
- `SET DEFAULT`: Imposta la foreign key al suo valore di default.
- `NO ACTION` / `RESTRICT`: Impedisce l'operazione (spesso il default).

`CHECK`

- `CHECK (condizione)`: Specifica una condizione che deve essere vera per ogni riga.
- Esempio: `CHECK (Wage > 0)`
- Esempio aggiuntivo:

```
Age INTEGER CHECK (Age >= 0 AND Age <= 120)
```

4.2.5 Modificare la Struttura

- `ALTER DOMAIN domain_name [...opzioni...];`
- `ALTER TABLE table_name [...opzioni...];`
  - Opzioni: `ADD COLUMN`, `DROP COLUMN col_name [RESTRICT|CASCADE]`, `ALTER COLUMN`, `ADD CONSTRAINT`, `DROP CONSTRAINT`.
- `DROP DOMAIN domain_name;`
- `DROP TABLE table_name;` (cancella la tabella e tutti i suoi dati!)

Esempio aggiuntivo:

```
ALTER TABLE Student ADD COLUMN Email VARCHAR(100);
ALTER TABLE Student DROP COLUMN Email;
ALTER TABLE Student ADD CONSTRAINT chk_age CHECK (Age >= 18);
```

4.2.6 Indici

- `CREATE INDEX index_name ON table_name (colonna1, [colonna2, ...]);`
- Migliorano le performance delle query.
- Strutture dati fisiche, non logiche.
- Le `PRIMARY KEY` e le colonne `UNIQUE` creano automaticamente un indice.
- *Esempio Pratico:* Se fai spesso ricerche di utenti per `email`, un indice su `User(email)` velocizzerà molto.
- Esempio aggiuntivo:

```
CREATE INDEX idx_salary ON Employee(Salary DESC);
```

Riepilogo tabellare:

Comando	Scopo	Esempio Sintetico
CHECK	Vincolo su valori	Wage INTEGER CHECK (Wage > 0)
ALTER	Modifica struttura tabella	ALTER TABLE T ADD COLUMN C INT;
CREATE INDEX	Crea indice per velocizzare query	CREATE INDEX idx ON T(C);

Riepilogo:
`CHECK` serve per vincoli sui dati.
`ALTER` modifica la struttura di tabelle/domìni.
`CREATE INDEX` velocizza le ricerche su una o più colonne.

4.3 DML (Data Manipulation Language) - Interrogare e Modificare i Dati

4.3.1 Interrogazioni (Query) - `SELECT`

La struttura base è:

```
SELECT [DISTINCT] {* | lista_colonne | espressioni [AS alias_colonna]}
FROM tabella1 [AS alias_tabella1]
[, tabella2 [AS alias_tabella2] ... |
JOIN_TYPE tabella2 ON condizione_join]
[WHERE condizione_filtro_righe]
[GROUP BY lista_colonne_raggruppamento]
[HAVING condizione_filtro_gruppi]
[ORDER BY lista_colonne_ordinamento [ASC|DESC]];
```

Ordine Concettuale di Esecuzione

1. `FROM` (e `JOIN`s)
2. `WHERE`
3. `GROUP BY`
4. `HAVING`
5. `SELECT`
6. `DISTINCT`
7. `ORDER BY`

Clausole base e alias

- `SELECT *`: Seleziona tutte le colonne.
- Rinominare Colonne e Tabelle (Alias): `AS nome_alias`

```
SELECT P.Name AS GivenName FROM PEOPLE AS P;
```

Condizioni `WHERE`

- Operatori logici: `AND`, `OR`, `NOT`.
- Operatori di confronto: `=`, `<>`, `<`, `>`, `<=`, `>=`.
- `LIKE`: Pattern matching (`%` per zero o più caratteri, `_` per un singolo carattere).

```
WHERE Name LIKE 'J_m%';
```

- Wildcard nel pattern matching:
  - `%`: Matcha zero o più caratteri qualsiasi

```
-- Trova tutti i nomi che iniziano con 'Jo'
WHERE Name LIKE 'Jo%';  -- Match: 'John', 'Joseph', 'Joanna'
-- Trova tutti i nomi che finiscono con 'son'
WHERE Name LIKE '%son';  -- Match: 'Johnson', 'Jackson', 'Wilson'
-- Trova tutti i nomi che contengono 'an'
WHERE Name LIKE '%an%';  -- Match: 'Frank', 'Andrew', 'Sandra'
```

  - `_`: Matcha esattamente un singolo carattere

```
-- Trova nomi di 4 lettere che iniziano con 'J'
WHERE Name LIKE 'J___';  -- Match: 'John', 'Jane', 'Jake'
-- Trova nomi che hanno 'a' come seconda lettera
WHERE Name LIKE '_a%';   -- Match: 'Sam', 'Paul', 'Mary'
```

  - `[...]`: Matcha un singolo carattere dalla lista (non supportato in tutti i DBMS)

```
-- Trova nomi che iniziano con 'A' o 'B'
WHERE Name LIKE '[AB]%';  -- Match: 'Alice', 'Bob', 'Anna'
```

  - `[^...]`: Matcha un singolo carattere NON nella lista (non supportato in tutti i DBMS)

```
-- Trova nomi che iniziano con qualsiasi lettera tranne 'A' e 'B'
WHERE Name LIKE '[^AB]%';  -- Match: 'Charlie', 'David', 'Emma'
```

- `IS NULL` / `IS NOT NULL`: Per verificare valori `NULL`.

`DISTINCT`

- Rimuove le righe duplicate dal risultato.

JOINs

Combinano righe da due o più tabelle.

- **Implicit JOIN** (sconsigliato):

```
SELECT ... FROM TableA, TableB WHERE TableA.id = TableB.a_id;
```

- **Explicit JOIN** (preferito):
  - `INNER JOIN` (o solo `JOIN`): Solo righe con corrispondenza in entrambe.

```
SELECT ... FROM TableA INNER JOIN TableB ON TableA.id = TableB.a_id;
```

  - `LEFT [OUTER] JOIN`: Tutte le righe da sinistra, e le corrispondenti da destra (o `NULL`).
  - `RIGHT [OUTER] JOIN`: Tutte le righe da destra, e le corrispondenti da sinistra (o `NULL`).
  - `FULL [OUTER] JOIN`: Tutte le righe da entrambe; `NULL` dove non c'è corrispondenza.
  - `NATURAL JOIN`: Join automatico su colonne con lo stesso nome (usare con cautela).
- *Esempio Pratico (Left Join):* Trovare tutti gli utenti e i loro post.

```
SELECT U.name, P.title
FROM User U LEFT JOIN Post P ON U.id = P.authorId;
```

Espressioni nella Target List

```
SELECT Income / 2 AS halvedIncome FROM PEOPLE;
```

Ordinamento

- `ORDER BY colonna [ASC|DESC];` (`ASC` è il default).

Operazioni sugli Insiemi (Set Operations)

Le query devono avere lo stesso numero di colonne e tipi compatibili.

- `UNION`: Combina risultati, rimuovendo duplicati.
- `UNION ALL`: Come `UNION`, ma mantiene i duplicati.
- `INTERSECT`: Righe presenti in entrambi i risultati.
- `EXCEPT` (o `MINUS`): Righe nel primo risultato ma non nel secondo.
- *Nota sulla denominazione delle colonne:* I nomi sono presi dalla prima query `SELECT`.

4.3.2 Subquery (Query Annidate)

Una query all'interno di un'altra.

Nelle clausole `WHERE`

- Con operatori di confronto: la subquery deve restituire un valore scalare.

```sql
-- Trova il dipendente con lo stipendio più alto
SELECT Name FROM EMPLOYEE
WHERE Salary = (SELECT MAX(Salary) FROM EMPLOYEE);

-- Trova i dipendenti che guadagnano più della media
SELECT Name, Salary FROM EMPLOYEE
WHERE Salary > (SELECT AVG(Salary) FROM EMPLOYEE);
```

- `IN`: Verifica se un valore è nel set di risultati della subquery.

```sql
-- Trova i dipendenti che lavorano in dipartimenti con budget > 1000000
SELECT Name FROM EMPLOYEE
WHERE Dept IN (SELECT Dept FROM DEPARTMENT WHERE Budget > 1000000);

-- Trova i clienti che hanno fatto almeno un ordine
SELECT Name FROM CUSTOMER
WHERE CustomerID IN (SELECT DISTINCT CustomerID FROM ORDERS);
```

- `ANY` / `SOME`, `ALL`: Usati con operatori di confronto.
  - `valore > ANY (subquery)`: vero se `valore` $>$ di almeno un valore della subquery.

```sql
-- Trova i prodotti più costosi di almeno un prodotto nella categoria 'Electronics'
SELECT Name, Price FROM PRODUCT
WHERE Price > ANY (SELECT Price FROM PRODUCT WHERE Category = 'Electronics');
```

  - `valore > ALL (subquery)`: vero se `valore` $>$ di tutti i valori della subquery.

```sql
-- Trova i prodotti più costosi di tutti i prodotti nella categoria 'Electronics'
SELECT Name, Price FROM PRODUCT
WHERE Price > ALL (SELECT Price FROM PRODUCT WHERE Category = 'Electronics');
```

- `EXISTS`: Vero se la subquery restituisce almeno una riga.

```sql
-- Trova i dipendenti che hanno almeno un progetto assegnato
SELECT Name FROM EMPLOYEE E
WHERE EXISTS (SELECT * FROM PROJECT P WHERE P.LeaderID = E.ID);

-- Trova i clienti che hanno fatto ordini nel 2023
SELECT Name FROM CUSTOMER C
WHERE EXISTS (
SELECT * FROM ORDERS O
WHERE O.CustomerID = C.ID
AND YEAR(O.OrderDate) = 2023
);
```

- `NOT EXISTS`: Vero se la subquery non restituisce righe.

```sql
-- Trova i dipendenti che non hanno progetti assegnati
SELECT Name FROM EMPLOYEE E
WHERE NOT EXISTS (SELECT * FROM PROJECT P WHERE P.LeaderID = E.ID);
```

```sql
-- Trova i prodotti che non sono mai stati ordinati
SELECT Name FROM PRODUCT P
WHERE NOT EXISTS (SELECT * FROM ORDER_ITEMS OI WHERE OI.ProductID = P.ID);
```

Visibilità (Scope)

- Una subquery può fare riferimento a colonne della query esterna (subquery correlata).
- La query esterna non può fare riferimento a colonne definite solo nella subquery.
- Se un nome di colonna è ambiguo, si assume quello dello scope più interno.

Nelle clausole `FROM` (Derived Tables)

La subquery agisce come una tabella temporanea e deve avere un alias.

```sql
SELECT P.Name, J.Child
FROM PEOPLE P, (SELECT Child FROM FATHERHOOD WHERE Father='Jim') AS J
WHERE P.Name = J.Child;
```

Nelle clausole `SELECT` (Scalar Subqueries)

La subquery deve restituire un singolo valore per ogni riga della query esterna.

```sql
SELECT C.Num, (SELECT COUNT(*) FROM ORDERS O WHERE O.CustomerNum = C.Num) AS OrderCount
FROM CUSTOMER C;
```

4.3.3 Funzioni Aggregate e Raggruppamento

Funzioni Aggregate

- `COUNT()`, `SUM()`, `AVG()`, `MIN()`, `MAX()`.
- Operano su un insieme di righe e restituiscono un singolo valore.
  - `COUNT(*)`: conta tutte le righe.
  - `COUNT(colonna)`: conta le righe dove `colonna` non è `NULL`.
  - `COUNT(DISTINCT colonna)`: conta i valori unici non `NULL`.
  - Le altre funzioni ignorano i `NULL`.
- **Attenzione:** Non mischiare colonne non aggregate con funzioni aggregate nella `SELECT` list a meno che le colonne non aggregate non siano nella `GROUP BY`.
  - Errato: `SELECT Name, MAX(Income) FROM PEOPLE;`
  - Corretto: `SELECT MAX(Income) FROM PEOPLE;`

`GROUP BY lista_colonne_raggruppamento`

- Raggruppa le righe che hanno gli stessi valori nelle colonne specificate.
- Le funzioni aggregate vengono applicate a ciascun gruppo.
- Esempio:

```sql
SELECT Dept, AVG(Wage) FROM EMPLOYEE GROUP BY Dept;
```

`HAVING condizione_filtro_gruppi`

- Filtra i gruppi creati da `GROUP BY`. La condizione in `HAVING` di solito coinvolge funzioni aggregate.
- `WHERE` filtra le righe *prima* del raggruppamento, `HAVING` filtra i gruppi *dopo*.
- Esempio:

```sql
SELECT Dept, AVG(Wage) FROM EMPLOYEE
GROUP BY Dept
HAVING AVG(Wage) > 50000;
```

NULLs e Raggruppamento

- I valori `NULL` in una colonna di raggruppamento formano un gruppo a sé stante.

4.3.4 Modifica dei Dati

`INSERT`

- ```sql
  INSERT INTO table_name [(colonna1, colonna2, ...)]
  VALUES (valore1, valore2, ...);
  ```

- Aggiunge una nuova riga. Se la lista colonne è omessa, fornire valori per tutte le colonne nell'ordine definito.

- ```sql
  INSERT INTO table_name [(colonna1, ...)]
  SELECT query_che_restituisce_righe_compatibili;
  ```

`UPDATE`

- ```sql
  UPDATE table_name
  SET colonna1 = valore1, colonna2 = valore2, ...
  [WHERE condizione];
  ```

- Modifica righe che soddisfano la `condizione`. **ATTENZIONE:** Senza `WHERE`, aggiorna tutte le righe!
- Il valore può essere un'espressione, `NULL`, `DEFAULT`, o una subquery scalare.

```sql
UPDATE PEOPLE SET Income = Income * 1.1 WHERE Age < 30;
```

`DELETE`
```

- ```
  DELETE FROM table_name [WHERE condizione];
  ```

- Cancella righe che soddisfano la `condizione`. **ATTENZIONE:** Senza `WHERE`, cancella tutte le righe!
- Può innescare azioni referenziali.

4.4 Concetti Chiave da Ricordare

1. **SQL è Dichiarativo:** Dici *cosa* vuoi, non *come* ottenerlo.
2. **Integrità dei Dati:** I vincoli sono fondamentali.
3. **NULL è Speciale:** Rappresenta assenza di valore. Va trattato con `IS NULL` / `IS NOT NULL`.
4. **JOINs sono Potenti:** Cuore delle query relazionali. Comprendere `INNER` vs `OUTER` JOINs è cruciale.
5. **Aggregazione e Raggruppamento:** `GROUP BY`, funzioni aggregate e `HAVING` permettono calcoli sui dati.
6. **Subquery:** Offrono flessibilità per query complesse.
  ```

Capitolo 5

SQL Avanzato

5.1 Vincoli (Constraints)

5.1.1 `CHECK`

- **Concetto**: Specifica vincoli sui valori che una tupla (riga) può assumere. È una forma di validazione dei dati a livello di database.
- **Sintassi**: `CHECK ( Predicate )`
- **Esempi**:
  - Semplice:

```
Gender CHARACTER NOT NULL CHECK (Gender IN ('M', 'F'))
```

  - Semplice:

```
Salary INTEGER CHECK (Salary >= 0)
```

  - Complesso (con subquery):

```
-- Assicura che lo stipendio di un impiegato non superi
-- quello del suo supervisore.
-- Nota: le subquery nei CHECK non sono supportate da tutti i DBMS.
CHECK (Salary <= (SELECT Salary
FROM EMPLOYEE J
WHERE Supervisor = J.Number))
```

  - Derivato:

```
-- Assicura la coerenza per campi calcolati.
CHECK (Net = Salary - Withholding)
```

- **Importanza**: Se un `INSERT` o `UPDATE` viola un vincolo `CHECK`, l'operazione fallisce, mantenendo l'integrità dei dati.

5.1.2 ASSERTION

- **Concetto**: Definisce vincoli a livello di schema, cioè che coinvolgono potenzialmente più tabelle o l'intero database, non solo una singola tupla.
- **Sintassi**: `CREATE ASSERTION NomeAsserzione CHECK ( Predicate )`
- **Esempio**:

```
-- Questa asserzione garantisce che la tabella EMPLOYEE
-- non sia mai completamente vuota.
CREATE ASSERTION AtLeastOneEmployee
CHECK (1 <= (SELECT COUNT(*) FROM EMPLOYEE));
```

- *Nota Pratica*: Anche qui, il supporto completo (specialmente con subquery complesse) varia tra i DBMS.

5.2 Viste (Views)

- **Concetto**: Una vista è una tabella virtuale il cui contenuto è definito da una query. Non memorizza dati fisicamente (generalmente), ma esegue la sua query sottostante ogni volta che viene interrogata.
- **Sintassi**: `CREATE VIEW NomeVista [(ListaAttributi)] AS SelectStatement`
- **Esempio**:

```
CREATE VIEW ADMINEMPLOYEES (Name, Surname, Salary) AS
SELECT Name, Surname, Salary
FROM EMPLOYEE
WHERE Dept = 'Administration' AND Salary > 10;
```

- **Utilizzi**:
  - **Semplificazione**: Nascondere la complessità di query complesse.
  - **Sicurezza**: Limitare l'accesso a determinate colonne o righe di una tabella.
  - **Indipendenza logica dei dati**: Se la struttura delle tabelle sottostanti cambia, la vista può essere modificata per mantenere la stessa interfaccia per gli utenti/applicazioni.

5.2.1 Aggiornamento delle Viste e `WITH CHECK OPTION`

- Le viste possono essere aggiornabili (tramite `INSERT`, `UPDATE`, `DELETE`) se definite su una singola tabella e soddisfano certe condizioni.
- `WITH CHECK OPTION`: Se specificato, qualsiasi `INSERT` o `UPDATE` eseguito tramite la vista deve soddisfare la clausola `WHERE` della vista stessa.
  - **Esempio**:

```
CREATE VIEW POORADMINEMPLOYEES AS
SELECT *
FROM ADMINEMPLOYEES -- Supponiamo sia una vista o tabella
WHERE Salary < 50
WITH CHECK OPTION;
```

Se si tenta di fare `UPDATE POORADMINEMPLOYEES SET Salary = 60 WHERE Name = 'Ann'`, l'operazione fallirà.

- `LOCAL` vs `CASCADED` (per viste su viste):
  - `LOCAL`: Il `CHECK OPTION` si applica solo alla definizione della vista corrente.
  - `CASCADED`: Il `CHECK OPTION` si applica alla vista corrente E a tutte le viste sottostanti.

5.2.2 Interrogare le Viste

- Si interrogano come normali tabelle. Il DBMS sostituisce la vista con la sua definizione.
- **Utilità per query complesse**:
  - **Problema**: "Calcolare la media del numero di uffici distinti per dipartimento". Una query come `SELECT AVG(COUNT(DISTINCT Office)) FROM EMPLOYEE GROUP BY Dept` è errata perché non si possono annidare funzioni aggregate direttamente.
  - **Soluzione con Vista**:

```sql
CREATE VIEW DEPTOFFICES (NameDept, OffNum) AS
SELECT Dept, COUNT(DISTINCT Office)
FROM EMPLOYEE
GROUP BY Dept;

SELECT AVG(OffNum) FROM DEPTOFFICES;
```

5.3 Query Ricorsive (`WITH RECURSIVE`)

- **Concetto**: Permettono di interrogare dati gerarchici o grafi. SQL:1999 ha introdotto le Common Table Expressions (CTE) ricorsive.
- **Sintassi Base**:

```sql
WITH RECURSIVE NomeCTE (colonne) AS (
-- Membro Ancora (non ricorsivo, caso base)
SELECT ...
UNION ALL
-- Membro Ricorsivo (richiama NomeCTE)
SELECT ... FROM NomeCTE JOIN ...
)
SELECT * FROM NomeCTE;
```

- **Esempio (Trovare tutti gli antenati)**: Data una tabella `FATHERHOOD(Father, Child)`

```sql
WITH RECURSIVE ANCESTORS (Ancestor, Descendant) AS (
-- Caso base: padri diretti
SELECT Father, Child FROM FATHERHOOD
UNION ALL
-- Passo ricorsivo: il padre di un antenato
-- è anche un antenato
SELECT FH.Father, A.Descendant
FROM FATHERHOOD FH, ANCESTORS A
WHERE FH.Child = A.Ancestor
)
SELECT * FROM ANCESTORS;
```

5.4 Funzioni Scalari

Funzioni che operano su valori singoli e restituiscono un singolo valore per tupla.

5.4.1 Temporali

- `CURRENT_DATE()`: Data corrente.
- `EXTRACT(parte FROM espressione_data)`: Estrae una parte da una data (es. `EXTRACT(YEAR FROM OrderDate)`).
- Esempio:

```
SELECT EXTRACT(YEAR FROM OrderDate) AS OrderYear
FROM ORDERS
WHERE DATE(OrderDate) = CURRENT_DATE();
```

5.4.2 Stringhe

- `CHAR_LENGTH(stringa)`: Lunghezza della stringa.
- `LOWER(stringa)`: Stringa in minuscolo.

5.4.3 Casting

- `CAST(espressione AS NuovoTipo)`: Converte un valore in un altro tipo di dato.

5.4.4 Condizionali

- `COALESCE(expr1, expr2, ..., default)`: Restituisce la prima espressione non-NULL nella lista.
  - Esempio: `SELECT COALESCE(Mobile, PhoneHome, 'N/A') FROM EMPLOYEE;`
- `NULLIF(expr1, expr2)`: Restituisce `NULL` se `expr1 = expr2`, altrimenti restituisce `expr1`.
  - Esempio: `SELECT NULLIF(Dept, 'Unknown') FROM EMPLOYEE;`
- `CASE`: Struttura if-then-else in SQL.
  - **Sintassi "Searched"**:

```
CASE
WHEN condizione1 THEN risultato1
WHEN condizione2 THEN risultato2
...
ELSE risultato_default
END
```

  - **Esempio**: Calcolo tasse veicoli

```
SELECT PlateNum,
(CASE Type
WHEN 'Car' THEN 2.58 * KWatt
WHEN 'Moto' THEN (22.00 + 1.00 * KWatt)
ELSE NULL
END) AS Tax
FROM VEHICLE
WHERE Year > 1975;
```

5.5 Sicurezza del Database

5.5.1 Privilegi

- SQL permette di concedere privilegi specifici (es. `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `REFERENCES`, `USAGE`) agli utenti.
- I privilegi possono essere su: intero DB, tabelle, viste, colonne, domini.

5.5.2 `GRANT` e `REVOKE`

- `GRANT`: Concede privilegi.
  - Sintassi: `GRANT <Privilegi | ALL PRIVILEGES> ON Risorsa TO Utenti [WITH GRANT OPTION];`
  - `WITH GRANT OPTION`: Permette all'utente ricevente di propagare quel privilegio ad altri.
  - Esempio: `GRANT SELECT ON DEPARTMENT TO Jack;`
- `REVOKE`: Rimuove privilegi.
  - Sintassi: `REVOKE Privilegi ON Risorsa FROM Utenti [RESTRICT | CASCADE];`
  - `RESTRICT` (default): La revoca fallisce se altri utenti dipendono da quel grant.
  - `CASCADE`: La revoca si estende a tutti gli utenti a cui il privilegio è stato propagato.

5.5.3 Discussione sui Privilegi

- Il sistema dovrebbe nascondere le parti del DB non accessibili senza dare indizi sulla loro esistenza.
- Le **viste** sono uno strumento chiave per la sicurezza: si possono concedere privilegi su una vista che mostra solo certe righe/colonne.

5.6 Autorizzazioni: RBAC (Role-Based Access Control)

- **Concetto**: SQL-3 introduce RBAC. Un `ROLE` (ruolo) è un contenitore di privilegi.
  1. Si creano ruoli.
  2. Si concedono privilegi AI RUOLI.
  3. Si concedono I RUOLI AGLI UTENTI.
- **Comandi RBAC**:
  - `CREATE ROLE NomeRuolo;`
  - `GRANT Privilegio ON Risorsa TO NomeRuolo;`
  - `GRANT NomeRuolo TO NomeUtente;`
  - `SET ROLE NomeRuolo;`
- **Esempio RBAC**:

```
-- 1. Crea il ruolo
CREATE ROLE Employee;
-- 2. Concedi un privilegio al ruolo
GRANT CREATE TABLE TO Employee;
-- 3. Assegna il ruolo a un utente
GRANT Employee TO 'specific_user';
```

5.7 Transazioni

- **Concetto**: Una transazione è un'unità logica di elaborazione del database, trattata come un'operazione atomica.
- **Proprietà ACID**:
  - **Atomicity (Atomicità)**: O tutto o niente.
  - **Consistency (Consistenza)**: Porta il DB da uno stato valido a un altro.
  - **Isolation (Isolamento)**: Le transazioni concorrenti non interferiscono.
  - **Durability (Durabilità)**: Le modifiche confermate (`COMMIT`) sono permanenti.
- **Supporto SQL per Transazioni**:
  - `START TRANSACTION;` (o `BEGIN TRANSACTION;`)
  - `COMMIT [WORK];`: Salva permanentemente le modifiche.
  - `ROLLBACK [WORK];`: Annulla tutte le modifiche.
  - `AUTOCOMMIT`: Modalità in cui ogni singola istruzione SQL è una transazione.
- **Esempio Transazione**:

```sql
START TRANSACTION;
UPDATE BANKACCOUNT SET Balance = Balance - 10
WHERE AccountNumber = 42177;
UPDATE BANKACCOUNT SET Balance = Balance + 10
WHERE AccountNumber = 12202;
-- Se tutto va bene:
COMMIT WORK;
-- Se c'è un errore (da verificare in logica applicativa):
-- ROLLBACK WORK;
```

Capitolo 6

Modellazione Concettuale dei Dati

6.1 Perché la Modellazione Concettuale?

Partire direttamente a definire tabelle SQL (modello logico) è difficile e rischioso. I problemi principali sono:

- Ci si perde nei dettagli troppo presto.
- Il modello relazionale (tabelle, colonne, tipi) è troppo *rigido* per le fasi iniziali di brainstorming e analisi dei requisiti.

La soluzione è il **Modello Concettuale** (ad esempio, il diagramma Entità-Relazione - ERD):

- Permette di ragionare sulla *realtà di interesse* in modo **indipendente dall'implementazione** specifica (quale DBMS useremo, come saranno le tabelle, ecc.).
- Aiuta a definire le **classi di oggetti** (entità) e le loro **relazioni**.
- Fornisce una **rappresentazione visuale** chiara, utile per la documentazione e la comunicazione con gli stakeholder (anche non tecnici).

Esempio Pratico: Immagina di dover creare un sistema per una biblioteca. Invece di pensare subito a `CREATE TABLE Libri (...)`, con il modello concettuale pensi: "Ok, ho bisogno di `Libri`, `Utenti`, e una relazione che dice *Un Utente prende in prestito un Libro*". Questo è più astratto e flessibile.

6.2 Il Ciclo di Vita del Design del Database

Il design del database è una fase cruciale nello sviluppo di Sistemi Informativi (SI). Le fasi principali del design sono:

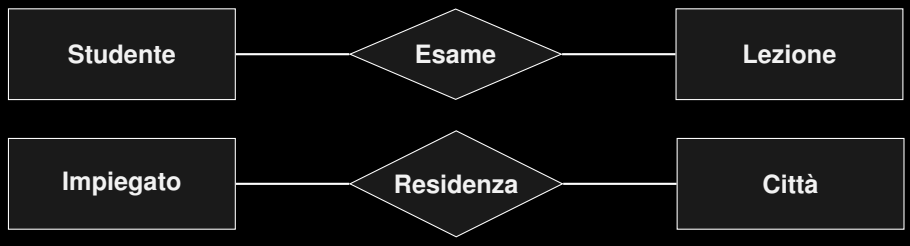
6.2.1 Design Concettuale

- **Input:** Requisiti del database (cosa deve fare il sistema?).
- **Output: Schema Concettuale** (es. un diagramma Entità-Relazione - ERD).
- **Focus:** Il "COSA?" – quali informazioni ci servono e come sono collegate, ad alto livello.
- *Esempio Pratico:* "Abbiamo Entità `Studente` e `Corso`. Uno `Studente` si `IscriveA` un `Corso`."

6.2.2 Design Logico

- **Input:** Schema Concettuale.
- **Output: Schema Logico** (es. definizione di tabelle per un DB relazionale, o collezioni per un DB NoSQL come MongoDB).

- **Focus:** Il “COME?” – come traduciamo il modello concettuale in un modello supportato da un tipo di DBMS (es. relazionale, a documenti, a grafo). È indipendente dal DBMS specifico, ma non dal *tipo* di DBMS.
- *Esempio Pratico:*



- Dallo schema concettuale sopra, con un po’ d’immaginazione sulle releazioni, con SQL potrebbe essere:

```sql
-- Table: Studente
CREATE TABLE Studente (
id INT PRIMARY KEY,
nome VARCHAR(100)
);

-- Table: Lezione
CREATE TABLE Lezione (
id INT PRIMARY KEY,
titolo VARCHAR(100)
);

-- Join Table: Esame (between Studente and Lezione)
CREATE TABLE Esame (
studente_id INT,
lezione_id INT,
data DATE,
voto INT,
PRIMARY KEY (studente_id, lezione_id),
FOREIGN KEY (studente_id) REFERENCES Studente(id),
FOREIGN KEY (lezione_id) REFERENCES Lezione(id)
);

-- Table: Impiegato
CREATE TABLE Impiegato (
id INT PRIMARY KEY,
nome VARCHAR(100)
);

-- Table: Città
CREATE TABLE Città (
id INT PRIMARY
)
```

6.2.3 Design Fisico

- **Input:** Schema Logico.
- **Output: Schema Fisico** (definizioni specifiche per il DBMS scelto: indici, partizionamento, filegroup, ecc.).
- **Focus:** Ottimizzazione delle performance e dello storage.

- *Esempio Pratico:* “Sulla tabella `Studenti`, creiamo un indice sulla colonna `Cognome` per velocizzare le ricerche.”

6.3 Modelli di Dati: Costrutti, Schemi e Istanze

- **Modello di Dati:** Una collezione di “costrutti” (come i tipi di dato in programmazione) per categorizzare i dati e descrivere le operazioni su di essi.
  - Esempio: il modello relazionale usa il costrutto `relazione` (tabella) per insiemi uniformi di `tuple` (righe).
- **Schema:** La struttura invariante nel tempo dei dati (aspetto *intensionale*).
  - SQL: `CREATE TABLE Users (id INT, name VARCHAR(255));`
  - Prisma: `model User { id Int @id; name String; }`
- **Istanza:** I valori attuali dei dati in un certo momento, che cambiano nel tempo (aspetto *estensionale*).
  - SQL: Le righe effettive nella tabella `Users`: `(1, 'Alice'), (2, 'Bob')`.
  - MongoDB: I documenti effettivi nella collezione `users`.

6.4 Il Modello Entità-Relazione (ER Model)

È il modello concettuale più usato. Ecco i suoi costrutti principali:

6.4.1 Entità (Entity)

- Rappresenta una classe di “oggetti” (cose, persone, luoghi) del mondo reale che hanno proprietà comuni e un’esistenza autonoma.
- **Esempi:** `Studente`, `Prodotto`, `Dipartimento`.
- **Rappresentazione Grafica:** Rettangolo.
- **Convenzioni:** Nomi singolari, significativi.
- *Paragone Pratico:* Simile a una classe in OOP, un `model` in Prisma, o una collezione in MongoDB.

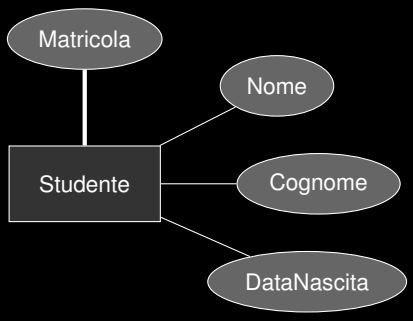


Figura 6.1: Esempio di entità Studente con i suoi attributi. La linea spessa indica l’identificatore (Matricola).

6.4.2 Relazione (Relationship)

- Un legame, un'associazione logica tra due o più tipi di entità.
- **Esempi:** `Studente` *Frequenta* `Corso`; `Impiegato` *LavoraIn* `Dipartimento`.
- **Rappresentazione Grafica:** Rombo.
- **Convenzioni:** Nomi singolari (se possibile, nomi invece di verbi).
- **Tipi:**
  - **Binarie:** Coinvolgono due entità.
  - **N-arie:** Coinvolgono più di due entità (es. `Fornitore` *Fornisce* `Prodotto` a un `Dipartimento`). Spesso si cerca di scomporle in binarie.
  - **Ricorsive:** Un'entità è in relazione con se stessa (es. `Impiegato` *Supervisiona* `Impiegato`).
    - *Paragone Pratico (Ricorsiva):* In SQL, una tabella `Impiegati` con una colonna `ID_Manager` che è una foreign key a `Impiegati.ID`.
- **Ruoli:** Utili nelle relazioni ricorsive per chiarire il significato (es. `Presidente` -(Precedente/Successivo)-> `Successione`).

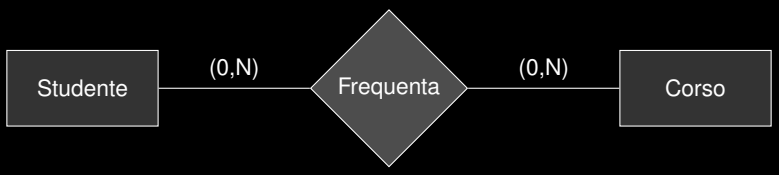


Figura 6.2: Esempio di relazione molti-a-molti tra Studente e Corso.

6.4.3 Promozione di Relazioni a Entità

Quando?

- Se una relazione ha attributi propri (es. la relazione `Iscrizione` tra `Studente` e `Corso` ha attributi come `DataIscrizione` e `VotoEsame`).
- Se uno studente può sostenere lo *stesso* esame più volte (es. per migliorare il voto). La semplice relazione `Studente-Esame-Corso` non cattura i tentativi multipli.

Come? La relazione diventa un'entità "associativa".

- *Esempio Pratico:* La relazione `Studente-Iscrizione-Corso` diventa: Entità `Studente` — Relazione `HaSostenuto` — Entità `IstanzaEsame` — Relazione `Riguarda` — Entità `Corso`. L'entità `IstanzaEsame` avrà attributi come `Data`, `Voto`.
- *SQL:* Questo si traduce in una "join table" o "tabella associativa":

```
CREATE TABLE EsamiSostenuti (
ID_Studente INT,
ID_Corso INT,
Data DATE,
Voto INT,
PRIMARY KEY (ID_Studente, ID_Corso, Data) -- Data inclusa per tentativi multipli
);
```

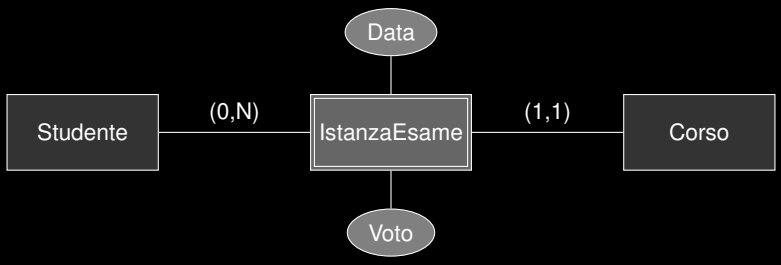


Figura 6.3: Esempio di promozione della relazione Esame a entità debole (doppio bordo) con attributi propri.

6.4.4 Attributi (Attribute)

- Una proprietà o caratteristica di un'entità o di una relazione.
- Collega ogni istanza dell'entità/relazione a un valore da un "dominio" (insieme di valori possibili).
- **Esempi:** `Nome` dell'entità `Studente`; `Data` della relazione `Esame`.
- **Rappresentazione Grafica:** Ovale.
- **Tipi:**
  - **Semplici:** Atomici (es. `Età`).
  - **Composti:** Possono essere scomposti in sotto-attributi (es. `Indirizzo` composto da `Via`, `NumeroCivico`, `Città`).
    - *Paragone Pratico (Composto):* In MongoDB è naturale: `address: { street: "...", city: "..." }`. In SQL, spesso si "appiattiscono" in colonne separate (`Via`, `NumeroCivico`, `Citta`) o, se complesso, si mette in una tabella separata.

6.4.5 Cardinalità (Cardinality)

Specifica il numero minimo e massimo di istanze di un'entità che possono partecipare a una relazione, o il numero di valori che un attributo può assumere.

- **Notazione comune:** `(min, max)`
  - `min = 0`: partecipazione opzionale.
  - `min = 1` (o più): partecipazione obbligatoria.
  - `max = 1`: al massimo una.
  - `max = N` (o $*$): molte.

Cardinalità delle Relazioni

- **Esempio:** `Impiegato (1,1)` — `LavoraPer` — `(0,N) Dipartimento`
  - Un `Impiegato` deve lavorare per **esattamente un** `Dipartimento`.
  - Un `Dipartimento` può avere **da zero a molti** `Impiegati`.
- **Tipi comuni (basati su max):**
  - **Uno-a-Uno (1:1):** Es. `Persona (0,1)` — `Possiede` — `(0,1) Pacemaker`.
  - **Uno-a-Molti (1:N):** Es. `Cliente (1,1)` — `Effettua` — `(0,N) Ordine`.
  - **Molti-a-Molti (M:N):** Es. `Studente (0,N)` — `Frequenta` — `(0,N) Corso`.
    - *Paragone Pratico (M:N):* In SQL, le relazioni M:N si implementano sempre con una tabella associativa intermedia. Prisma gestisce questo in modo più astratto.

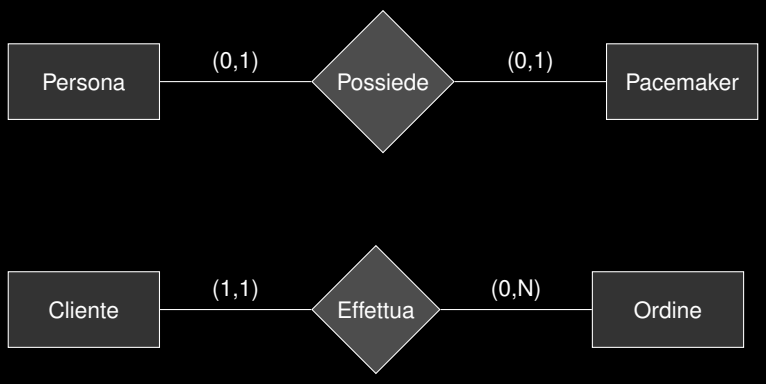


Figura 6.4: Esempi di relazioni uno-a-uno (Persona-Pacemaker) e uno-a-molti (Cliente-Ordine).

Cardinalità degli Attributi

- `(0,1)`: Attributo opzionale (può essere `NULL`). Es. `NumeroTelefonoSecondario`.
- `(1,1)`: Attributo obbligatorio, singolo valore (default). Es. `CodiceFiscale`.
- `(0,N)` o `(1,N)`: Attributo multivalore (un'entità può avere più valori per quell'attributo). Es. `NumeriTelefono` (una persona può avere più numeri).
  - *Paragone Pratico (Multivalore):* In SQL, si usa una tabella separata: `Persona(ID_Persona)`, `NumeriTelefono(ID_Persona_FK, Numero)`. In MongoDB, si usa un array: `telefoni: ["123", "456"]`.

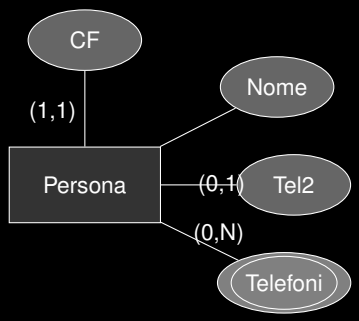


Figura 6.5: Esempi di attributi con diverse cardinalità: obbligatorio (CF), opzionale (Tel2), e multivalore (Telefoni).

6.4.6 Identificatori (Chiavi - Keys)

- Un attributo o un insieme di attributi che identificano univocamente ogni istanza di un'entità.
- **Rappresentazione Grafica:** Attributo sottolineato.
- **Tipi:**
  - **Identificatore Interno:** Formato da attributi della stessa entità.
    - Es. `codiceFiscale` per l'entità `Persona`.
    - *Paragone Pratico:* `PRIMARY KEY` in SQL; `_id` in MongoDB; `@id` in Prisma.
  - **Identificatore Esterno:** Formato da attributi dell'entità più l'identificatore di un'entità esterna a cui è collegata tramite una relazione con cardinalità `(1,1)` dal lato dell'entità da identificare. Usato per "entità deboli" che non possono esistere o essere identificate senza l'entità "forte".
    - Es. `lineId` (attributo di `OrderItem`) + `orderId` (dall'entità `Order`) identifica univocamente un `OrderItem`. `OrderItem` è un'entità debole rispetto a `Order`.

- Ogni entità deve avere almeno un identificatore.
- Le relazioni di solito non hanno identificatori (se ne hanno bisogno, si promuovono a entità).

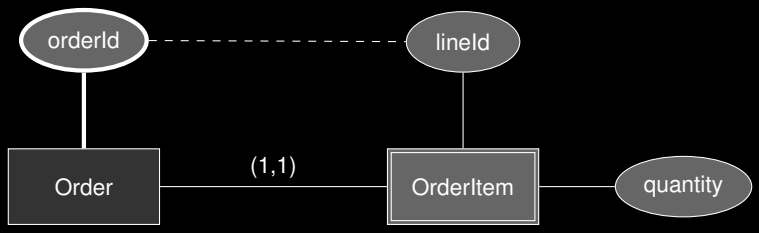


Figura 6.6: Esempio di entità forte (Order) con identificatore interno e entità debole (OrderItem) con identificatore esterno.

```
// Esempio con Prisma
model Order {
orderId    Int         @id @default(autoincrement())
// altri campi dell'ordine
orderItems OrderItem[]
}

model OrderItem {
orderId     Int
lineId      Int
quantity    Int
order       Order  @relation(fields: [orderId], references: [orderId])

@@id([orderId, lineId]) // Chiave primaria composta
}
```

```
-- Definizione SQL dello schema
CREATE TABLE "Order" (
orderId    SERIAL PRIMARY KEY,
// altri campi dell'ordine
);

CREATE TABLE OrderItem (
orderId     INT,
lineId      INT,
quantity    INT,
PRIMARY KEY (orderId, lineId),
FOREIGN KEY (orderId) REFERENCES "Order"(orderId) ON DELETE CASCADE
);
```

6.4.7 Generalizzazione/Specializzazione (Inheritance)

- Una relazione tra un'entità genitore (superclasse, es. `Veicolo`) e una o più entità figlie (sottoclassi, es. `Automobile`, `Motocicletta`).
- Le figlie sono "tipi di" genitore: ereditano attributi e relazioni del genitore e possono averne di propri.
- **Rappresentazione Grafica:** Freccia (triangolo vuoto) dalle figlie al genitore.
- **Proprietà:**
  - **Ereditarietà:** Le proprietà del genitore sono implicitamente presenti nelle figlie.

  - **Copertura (Total/Partial):**
    - **Totale:** Ogni istanza del genitore DEVE essere un'istanza di (almeno) una delle figlie. (Es. `Persona` -> `Maschio`, `Femmina`).
    - **Parziale:** Un'istanza del genitore PUÒ essere un'istanza di una figlia (o solo del tipo genitore). (Es. `Veicolo` -> `Automobile`, `Motocicletta`).
  - **Disgiunzione (Disjoint/Overlapping):**
    - **Disgiunta:** Un'istanza del genitore può essere al massimo un tipo di figlia. (Es. `Persona` è `Maschio` O `Femmina`).
    - **Sovrapposta:** Un'istanza del genitore può essere più tipi di figlia contemporaneamente (raro e più complesso da modellare).
  - Di solito ci si concentra su generalizzazioni **Disgiunte (Totali o Parziali)**.
- *Paragone Pratico:*
  - OOP: `class Veicolo {}`, `class Automobile extends Veicolo {}`.
  - SQL: Ci sono diversi pattern per implementare l'ereditarietà.
  - Prisma: Può essere modellato con campi discriminatori o modelli separati con relazioni.

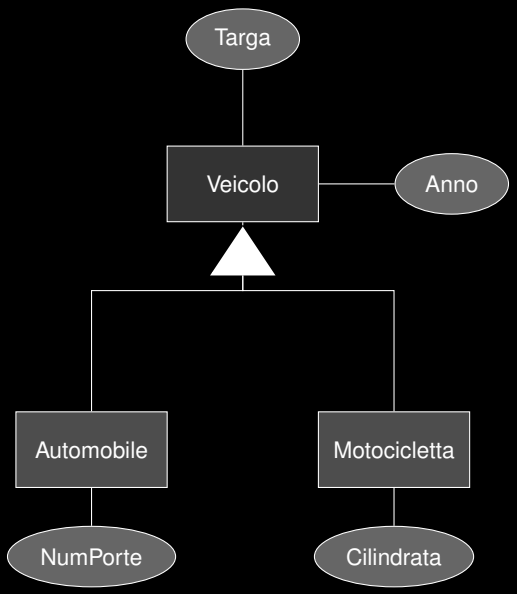


Figura 6.7: Esempio di generalizzazione/specializzazione: Veicolo come superclasse e Automobile/Motocicletta come sottoclassi con simbolo di ereditarietà (triangolo vuoto).

6.5 Documentazione

- **Dizionario dei Dati:** Descrive in dettaglio ogni entità, relazione e attributo.
- **Vincoli Non Esprimibili:** Alcuni vincoli non possono essere rappresentati graficamente nell'ERD (es. "Lo stipendio di un impiegato non può superare quello del suo manager"). Vanno documentati a parte.
  - *Paragone Pratico:* Questi vincoli si implementano spesso con `CHECK constraints` in SQL, `triggers`, o a livello applicativo.

6.6 UML (Unified Modeling Language) come Alternativa

- UML è un linguaggio di modellazione più ampio, usato per vari aspetti dello sviluppo software.
- Per la modellazione dei dati, si usano principalmente i **Diagrammi delle Classi (Class Diagrams)**.

- Molti concetti ER hanno un equivalente in UML:
  - **Entità** -> **Classe**
  - **Relazione** -> **Associazione**
  - **Relazione con attributi** -> **Classe di Associazione**
  - **Cardinalità:** `1`, `0..1`, `*`, `1..*`
  - **Identificatori:** `{id}` accanto all'attributo.
  - **Generalizzazione/Specializzazione:** Freccia con triangolo vuoto verso la superclasse.
  - **Concetti specifici UML:** Aggregazione (rombo vuoto), Composizione (rombo pieno).

6.7 Modellazione Concettuale con UML (Unified Modeling Language)

UML è un linguaggio di modellazione standardizzato e ampiamente utilizzato per specificare, visualizzare, costruire e documentare gli artefatti di un sistema software. Sebbene l'ERD sia specifico per i database, UML offre un approccio più generale e può essere utilizzato anche per la modellazione concettuale dei dati, principalmente attraverso i **Diagrammi delle Classi (Class Diagrams)**.

6.7.1 Classi (Classes)

In UML, un'entità del modello ER corrisponde a una **Classe**. Una classe è rappresentata come un rettangolo, tipicamente diviso in tre sezioni:

1. **Nome della Classe:** In alto, in grassetto.
2. **Attributi (Attributes):** Al centro, elencano le proprietà della classe.
3. **Operazioni (Operations/Methods):** In basso (spesso omessa nella modellazione concettuale dei dati puri, poiché ci si concentra sulla struttura).

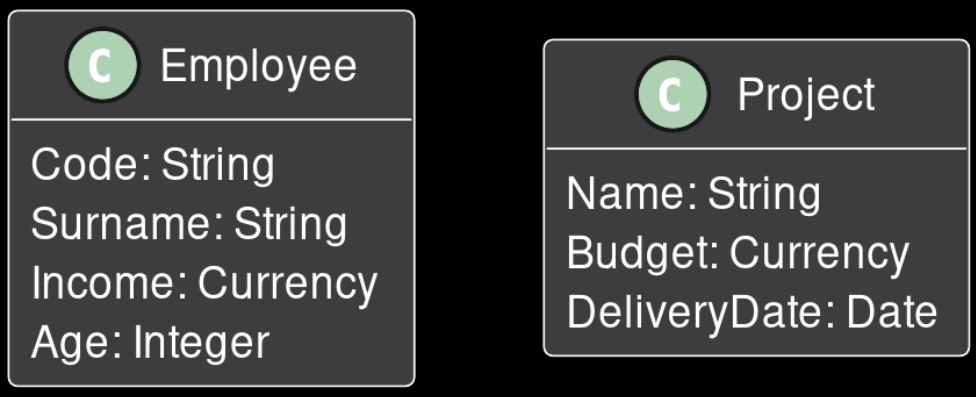


Figura 6.8: Esempio di Classi UML: Employee e Project.

6.7.2 Associazioni (Associations)

Le relazioni del modello ER sono chiamate **Associazioni** in UML. Un'associazione rappresenta una relazione semantica tra due o più classi. È disegnata come una linea continua che connette le classi.

- **Nome dell'Associazione (opzionale):** Può essere scritto vicino alla linea, spesso con una freccetta che indica la direzione di lettura (se il nome è un verbo).
- **Ruoli (opzionale):** Nomi posti alle estremità della linea di associazione per chiarire il ruolo che una classe gioca nell'associazione.
- **Molteplicità (Multiplicity):** Equivalente alla cardinalità ER, indica quante istanze di una classe possono essere collegate a un'istanza dell'altra classe. Notazioni comuni:

- `1` (esattamente uno)
- `0..1` (zero o uno)
- `*` (zero o molti)
- `1..*` (uno o molti)
- `m..n` (da m a n)

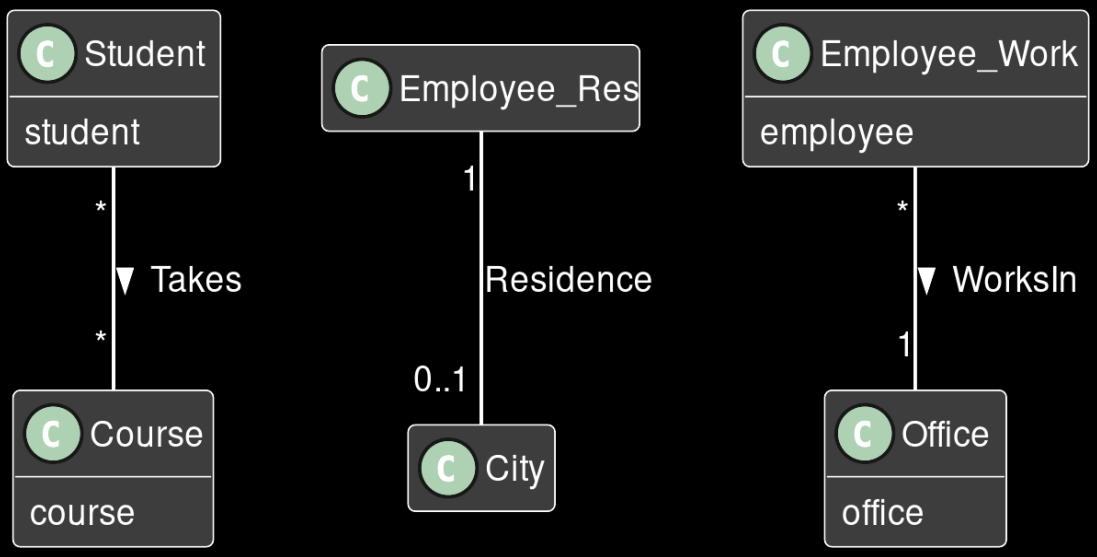


Figura 6.9: Esempi di Associazioni UML con nomi, ruoli e molteplicità.

6.7.3 Classe di Associazione (Association Class)

Quando un'associazione stessa ha attributi o operazioni, può essere modellata come una **Classe di Associazione**. È rappresentata come una classe normale collegata da una linea tratteggiata all'associazione che descrive. *Esempio:* L'associazione "Sostiene Esame" tra `Student` e `Course` può avere attributi come `Date` e `Degree`. `Exam` diventa una classe di associazione.

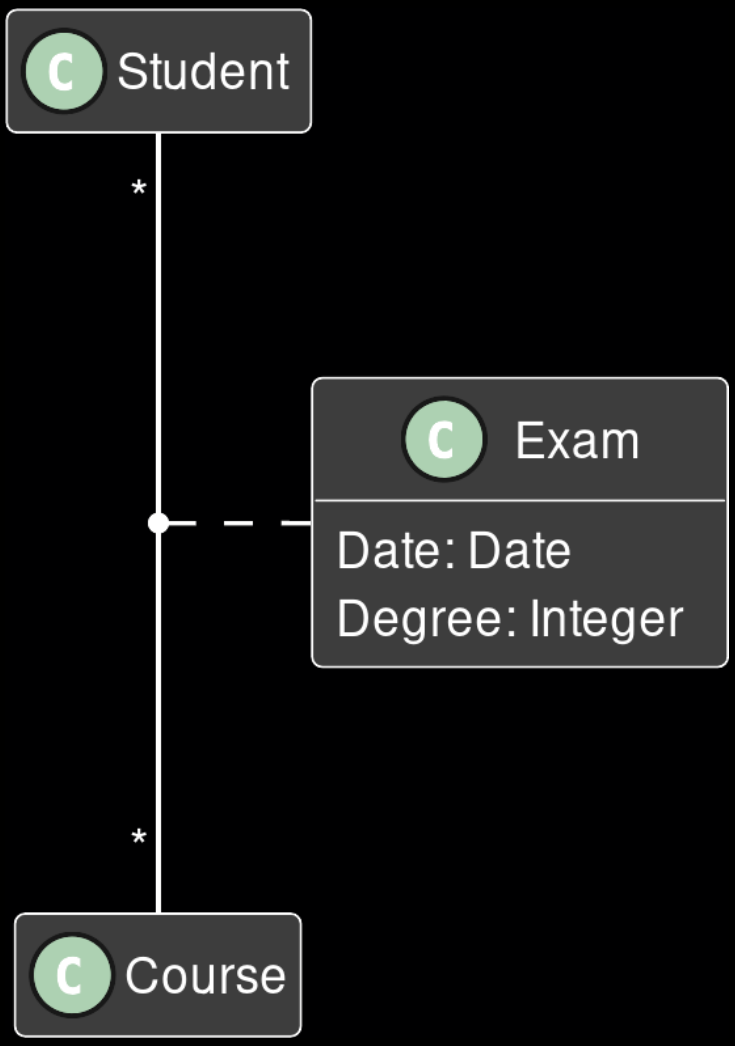


Figura 6.10: Esempio di Classe di Associazione UML: Exam.

6.7.4 Associazione N-aria (N-ary Association) e Reificazione

Un'associazione può coinvolgere più di due classi (ternaria, quaternaria, ecc.). Graficamente, si usa un rombo (come in ER) a cui sono collegate le classi. Se l'associazione n-aria ha attributi, una classe di associazione viene collegata al rombo. Le associazioni n-arie (n > 2) sono spesso complesse da gestire e interpretare. Una pratica comune è la **reificazione** (o "promozione a classe"): l'associazione n-aria viene trasformata in una nuova classe regolare, che viene poi collegata alle classi originariamente coinvolte tramite associazioni binarie.

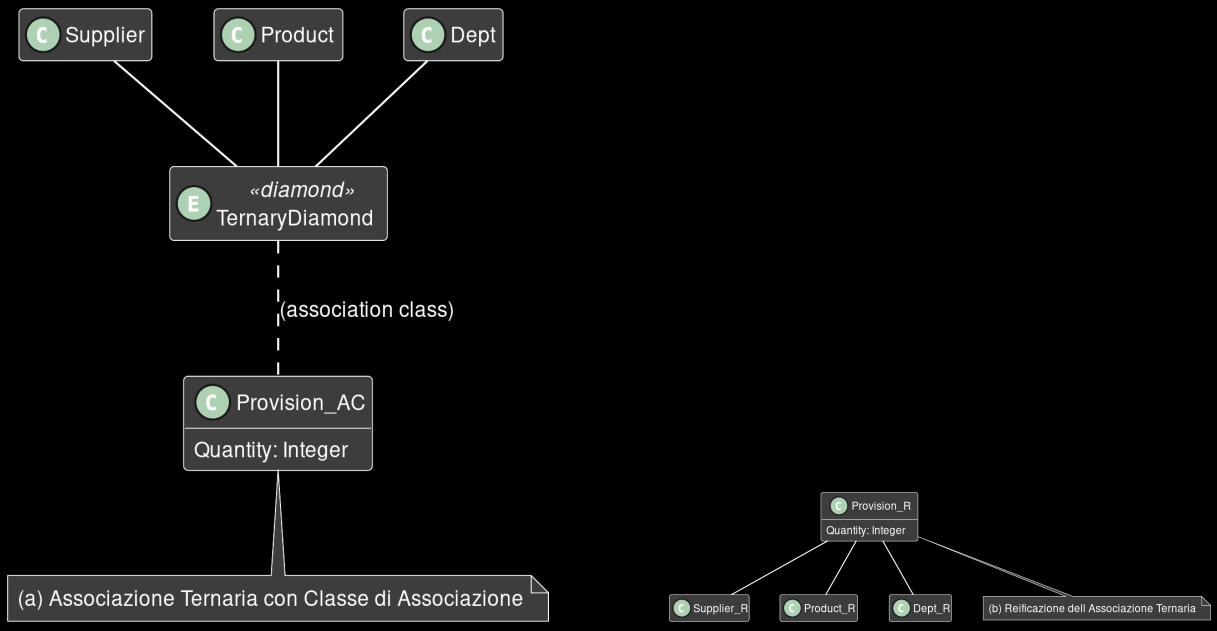


Figura 6.11: Associazione Ternaria con classe di associazione (a, sinistra) e sua Reificazione (b, destra) in UML.

6.7.5 Aggregazione e Composizione (Aggregation and Composition)

Sono tipi speciali di associazione che rappresentano relazioni "parte-di" (whole-part).

- **Aggregazione (Aggregation):** Rappresenta una relazione "ha-un" debole. Le parti possono esistere indipendentemente dal tutto. È indicata da un **rombo vuoto** dal lato del "tutto" (aggregato). *Esempio:* Un `Team` è composto da `Technician`. Un tecnico può esistere anche se il team viene sciolto, o può appartenere a più team (a seconda della molteplicità).
- **Composizione (Composition):** Rappresenta una relazione "ha-un" forte. Le parti dipendono esistenzialmente dal tutto; se il tutto viene distrutto, anche le parti lo sono. È indicata da un **rombo pieno** dal lato del "tutto" (composito). La molteplicità dal lato del composito verso la parte è solitamente `1` o `0..1` (una parte appartiene a un solo tutto). *Esempio:* Un'automobile (`Car`) è composta da un motore (`Engine`). Se l'automobile viene rottamata, anche il suo motore specifico (come parte di quell'auto) cessa di esistere in quel contesto. Una `Agency` è parte di una `Firm`.

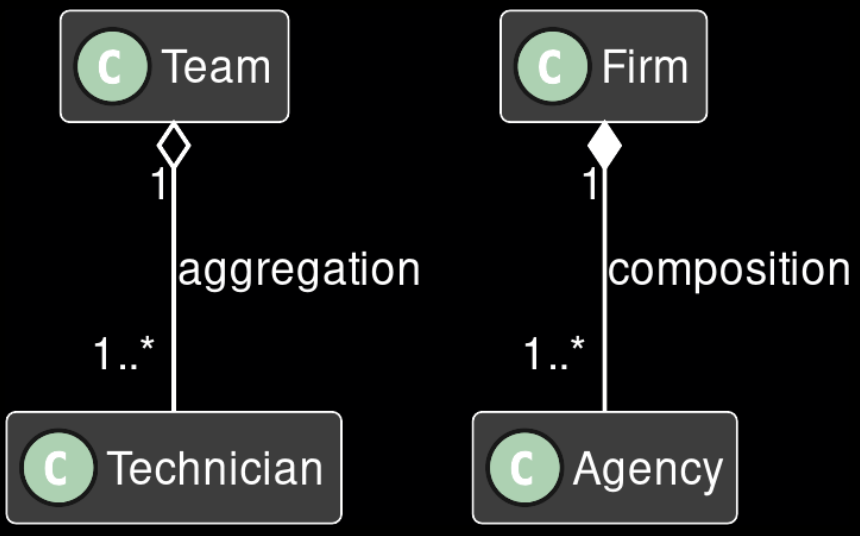


Figura 6.12: Esempio di Aggregazione (Team-Technician) e Composizione (Firm-Agency) in UML.

6.7.6 Identificatori (Identifiers)

In UML, gli attributi che compongono l'identificatore (chiave primaria) di una classe possono essere contrassegnati con la proprietà `{id}` o talvolta sottolineati (anche se `{id}` è più comune in UML2).

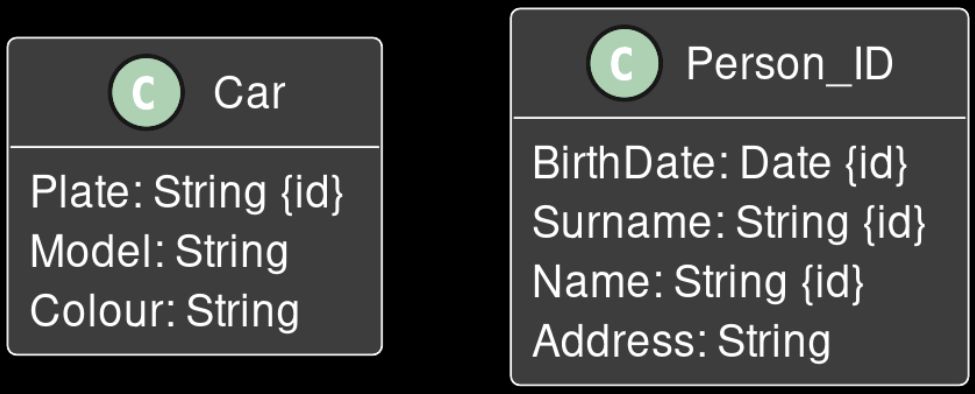


Figura 6.13: Esempio di Identificatori UML con `{id}`.

6.7.7 Identificatore Esterno (External Identifier) e Associazioni Qualificate

Un concetto simile all'identificatore esterno ER si ha quando l'identità di una classe (la "debole") dipende parzialmente da un'altra classe attraverso un'associazione. Questo può essere indicato con:

- Uno **stereotipo** sull'associazione, come `«identifying»` (come suggerito nelle slide del prof., anche se non standard UML stretto per questo specifico caso).
- Una **Associazione Qualificata**: un piccolo rettangolo (il qualificatore) è attaccato alla classe "forte", contenente un attributo della classe "debole" che, insieme all'istanza della classe forte, identifica univocamente l'istanza della classe debole.
- La molteplicità dal lato della classe "debole" verso la "forte" è spesso `1` in un'associazione identificante.

Nelle slide del Prof. Montesi (slide 100), si usa lo stereotipo `«identifiying»` (probabile typo per `«identifying»`) sull'associazione e `{id}` sugli attributi che compongono la chiave, inclusi quelli della classe "debole" e quelli ereditati implicitamente tramite l'associazione identificante.

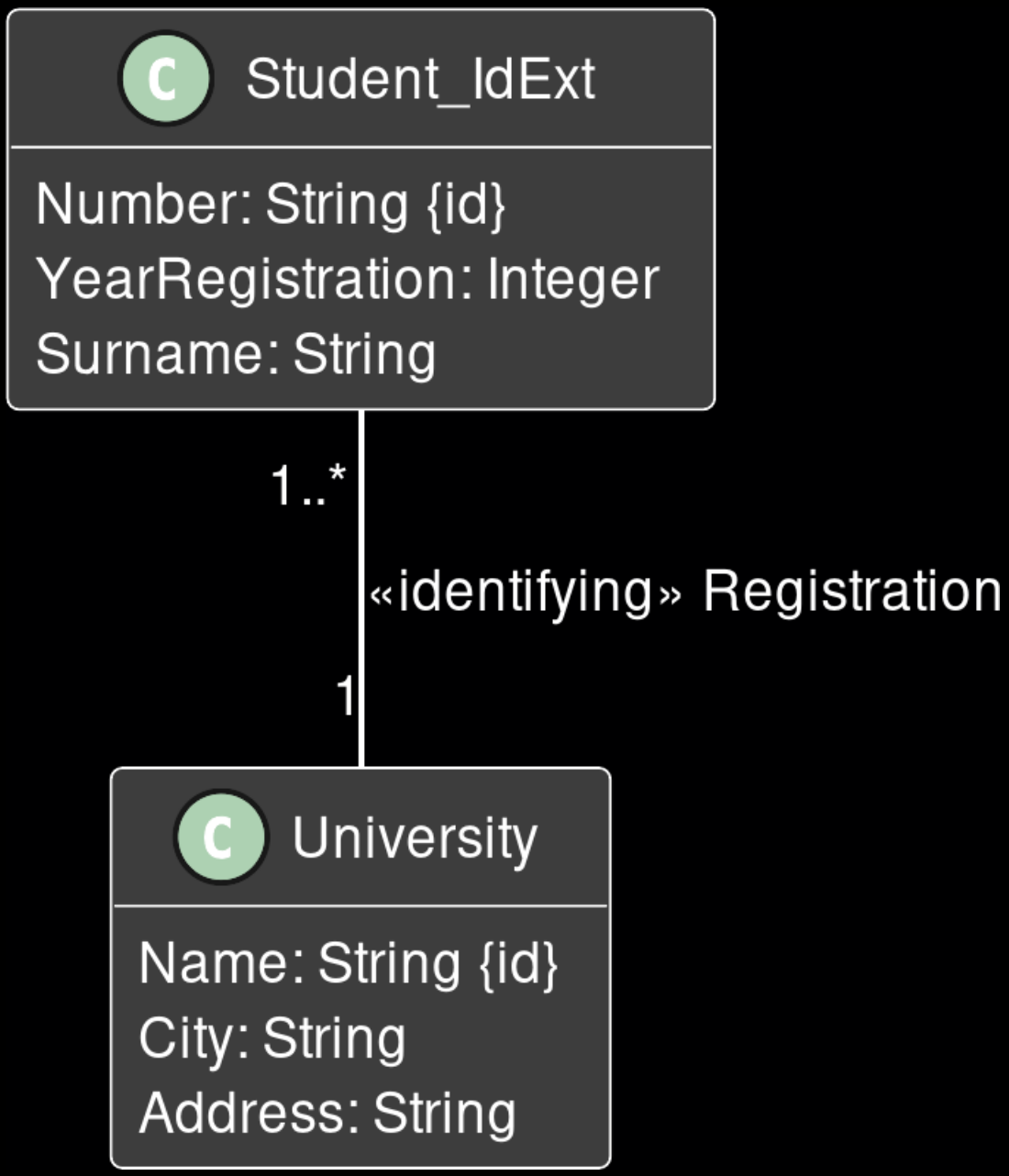


Figura 6.14: Esempio di associazione identificante con stereotipo (come da slide 100).

Nota: L'attributo `Number` di `Student` è `{id}` nel contesto dell'associazione con `University`. La vera chiave univoca di `Student` sarebbe una combinazione di `Student.Number` e l'identificatore di `University`.

6.7.8 Generalizzazione (Generalization)

Corrisponde alla generalizzazione/specializzazione del modello ER e rappresenta una relazione "è-un-tipo-di" (is-a-kind-of) tra una classe più generale (superclasse) e una classe più specifica (sottoclasse).

- **Rappresentazione Grafica:** Una linea continua con una **grande freccia triangolare vuota** che punta dalla sottoclasse alla superclasse.
- **Ereditarietà:** La sottoclasse eredita attributi, operazioni e associazioni della superclasse.
- **Vincoli:** Simili a ER, si possono specificare vincoli come:
  - `{complete, disjoint}` o `{total, disjoint}`: Ogni istanza della superclasse è esattamente una delle sottoclassi.

- `{incomplete, disjoint}` o `{partial, disjoint}`: Un'istanza della superclasse può essere una delle sottoclassi o nessuna di esse (solo la superclasse), ma non più di una.
- `{overlapping}`: Una sottoclasse può essere istanza di più sottoclassi (più raro).

Questi vincoli sono scritti vicino alla freccia di generalizzazione.

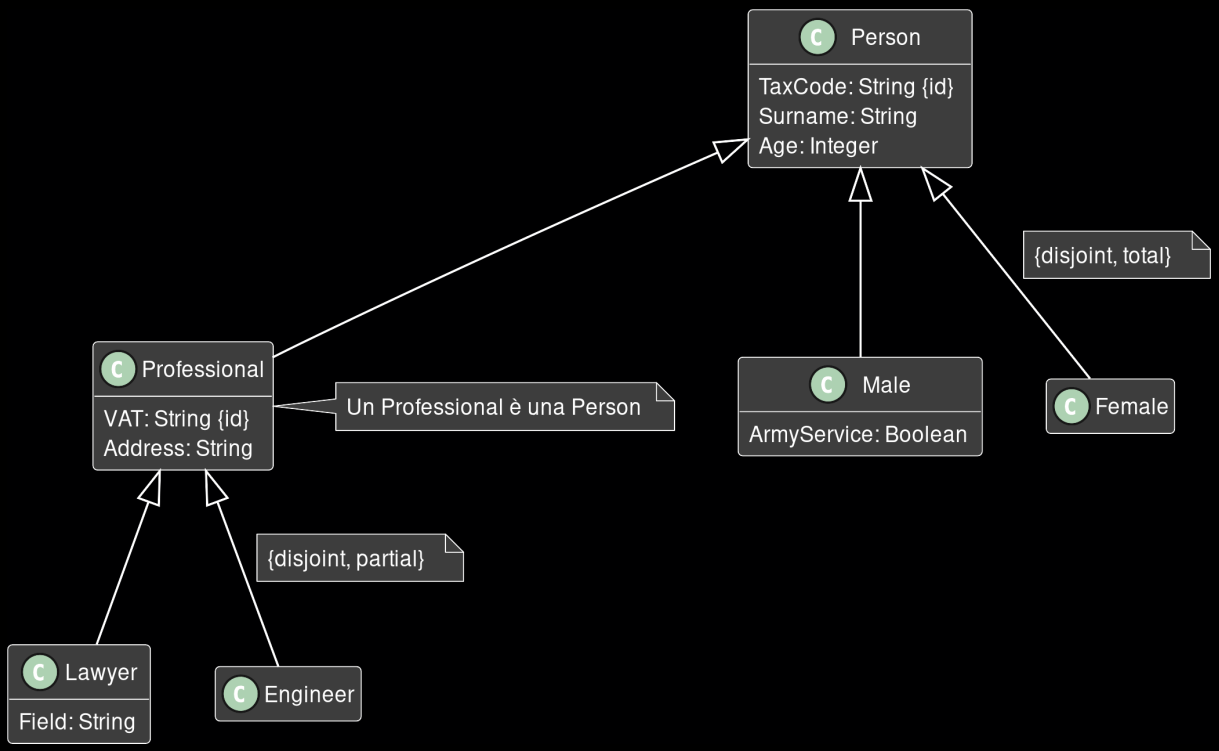


Figura 6.15: Esempio di Generalizzazione UML (basato su slide 101).

6.7.9 Esempio Complessivo: Schema Concettuale in UML

La slide 102 del Prof. Montesi mostra un diagramma ER tradotto in UML Class Diagram. Notiamo:

- Le entità diventano Classi (`Employee`, `Dept`, `Project`, `Office`).
- Le relazioni diventano Associazioni (`Management`, `Affiliation`, `Attendance`).
- `Affiliation` è una classe di associazione perché ha l'attributo `Date`.
- `Composition` tra `Dept` e `Office` è una composizione forte (rombo pieno) e identificante (`«identifying»`).
- Le cardinalità ER sono tradotte in molteplicità UML.
- Gli identificatori sono marcati con `{id}`.
- Una nota è attaccata alla classe `Project`.

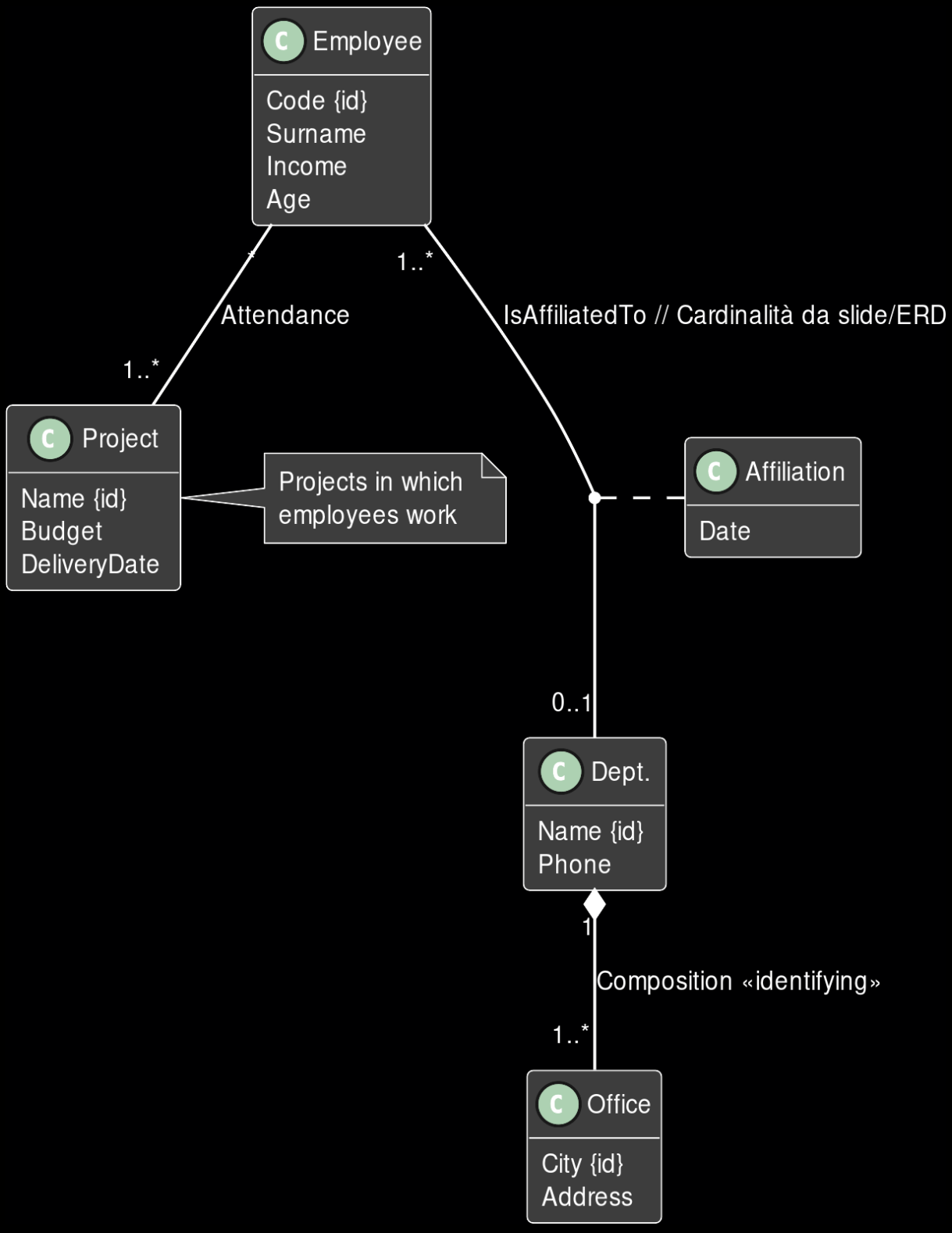


Figura 6.16: Diagramma concettuale UML basato sulla slide 102.

Conclusione: UML offre un set ricco di costrutti per la modellazione concettuale dei dati, con una notazione leggermente diversa ma concettualmente simile a ER. La scelta tra ER e UML Class Diagrams spesso dipende dalle convenzioni del team o dalla necessità di integrare il modello dati con altri modelli UML del sistema.

Capitolo 7

Progettazione Concettuale di Basi di Dati

7.1 Introduzione alla Progettazione Concettuale

L'obiettivo della progettazione concettuale è creare un modello dei dati che sia **indipendente** da qualsiasi specifico Database Management System (DBMS) o tecnologia. È la fase in cui tradiamo i requisiti del mondo reale in una struttura formale.

7.2 Il Processo di Progettazione di un Database

Il design di un database si articola in diverse fasi:

1. **Raccolta dei Requisiti del DB (**`DB requirements`**)**: Cosa deve fare il database? Quali dati deve memorizzare?
2. **Progettazione Concettuale (**`Conceptual Design`**)**:
    - È la fase di **ANALISI** ("`WHAT?`").
    - Si traduce i requisiti in un modello concettuale (es. Diagramma Entità-Relazione o E-R).
    - L'output è lo **Schema Concettuale (**`Conceptual Schema`**)**. Questo schema è una descrizione astratta della struttura del database, focalizzata sulle entità, gli attributi e le relazioni tra di esse, senza preoccuparsi di come verranno implementate.
3. **Progettazione Logica (**`Logical Design`**)**:
    - È la fase di **PROGETTAZIONE** ("`HOW?`").
    - Si traduce lo schema concettuale in un modello logico, specifico per un tipo di DBMS (es. relazionale, NoSQL).
    - L'output è lo **Schema Logico (**`Logical Schema`**)** (es. tabelle SQL con chiavi primarie/esterne, tipi di dato).
4. **Progettazione Fisica (**`Physical Design`**)**:
    - Si specificano i dettagli di implementazione fisica (es. indici, partizionamento).
    - L'output è lo **Schema Fisico (**`Physical Schema`**)**.

Noi ci concentriamo sulla Progettazione Concettuale.

7.3 Attività della Progettazione Concettuale e Modellazione dei Dati

Queste attività sono interconnesse e spesso iterative:

- **Elicitazione dei Requisiti (`Requirements' elicitation`)**: Raccogliere le informazioni.
- **Analisi dei Requisiti (`Requirements' analysis`)**: Capire e chiarire le informazioni.
- **Costruzione dello Schema Concettuale (`Building the conceptual schema`)**: Disegnare il modello E-R.
- **Costruzione del Glossario (`Building the glossary`)**: Definire i termini chiave.

7.4 Raccolta dei Requisiti

7.4.1 Fonti dei Requisiti

- **Utenti e Clienti**:
  - Interviste.
  - Documentazione specifica (*ad hoc*).
- **Documentazione Esistente**:
  - Leggi e regolamenti di settore.
  - Regolamenti interni, processi aziendali.
  - Soluzioni preesistenti.
- **Moduli (`Forms`)**: I moduli cartacei o digitali esistenti sono una miniera d'oro di informazioni sui dati.

7.4.2 Acquisizione e Analisi

- È un'attività **difficile e non standardizzata**.
- Spesso si parte da requisiti iniziali che necessitano di **raffinamento** attraverso ulteriori acquisizioni.

7.4.3 Acquisizione tramite Interviste

- **Diverse tipologie di utenti forniscono informazioni diverse**.
- I manager di alto livello hanno una visione più ampia ma meno dettagliata.
- Le interviste portano a raffinamenti successivi.

7.4.4 Interagire con gli Utenti: Consigli

- **Controlli di comprensione e coerenza frequenti**: "Quindi, se ho capito bene, ogni studente può iscriversi a più corsi, e ogni corso può avere più studenti?"
- **Esempi di casi d'uso (`Use cases`)**: Molto utili, specialmente casi generici e casi limite. "Cosa succede se uno studente si iscrive e poi si ritira? E se un corso non ha iscritti?"
- **Chiedere definizioni e classificazioni**: "Cosa intende esattamente per 'studente attivo'?"
- **Chiedere di evidenziare aspetti essenziali vs. periferici**: "È fondamentale tracciare lo storico degli indirizzi dello studente, o basta l'indirizzo attuale?"

7.5 Documentazione Descrittiva e Gestione dei Termini

7.5.1 Regole per la Documentazione Descrittiva

- **Scegliere il giusto livello di astrazione**.
- **Struttura delle frasi standard**: Semplifica l'analisi.
- **Dividere frasi troppo lunghe/complesse**.
- **Distinguere frasi sui "dati" da frasi sulle "funzioni"**:
  - Dati: "Uno studente *ha* un nome, un cognome e una matricola."
  - Funzioni: "Il sistema *deve permettere* di iscrivere uno studente a un corso."

7.5.2 Regole Generali per Termini e Concetti

- **Costruire un glossario dei termini**: Cruciale per evitare ambiguità.
  - *Esempio Pratico*: Se nel tuo team Node.js uno chiama un campo `customerId` e un altro `client_id`, il glossario chiarisce che `Customer` e `Client` sono sinonimi e si userà `customerId`.
- **Omonimi e sinonimi devono essere unificati**: Un solo termine per un concetto.
- **Chiarire esplicitamente le relazioni tra i termini**.
- **Ordinare le frasi per concetti**: Raggruppare requisiti simili.

7.6 Esempi di Requisiti

7.6.1 Esempio Database Bibliografico

- Automatizzare riferimenti bibliografici.
- ID di 7 caratteri (iniziali autori, anno, carattere disambiguazione).
- Riferimenti possono essere *monografie* (editore, data, luogo) o *articoli di rivista* (nome rivista, volume, numero, pagine, anno).
- Per entrambi: nomi degli autori.

7.6.2 Esempio Azienda di Formazione

Questo esempio sarà usato più avanti per la progettazione.

- Gestire corsi, lezioni, insegnanti.
- **Studenti ($\sim$5000)**: ID, codice fiscale, cognome, età, sesso, luogo nascita, nome datori di lavoro (passati e presenti con date), indirizzo, telefono, corsi frequentati ($\sim$200) con voto finale.
- **Workshop/Corsi**: Tracciare workshop frequentati, dove e quando si tengono le lezioni. I corsi hanno codice, titolo, edizioni (con data inizio/fine, numero partecipanti).
- **Studenti Freelance**: Area di interesse, titolo onorifico.
- **Insegnanti ($\sim$300)**: Cognome, età, luogo nascita, nome corso insegnato, set corsi insegnati (passati/futuri), storico telefonate. Possono essere dipendenti interni o collaboratori esterni.

7.7 Il Glossario

Il glossario è fondamentale. Ecco un esempio basato sull'azienda di formazione:

Termine	Descrizione	Sinonimo	Correlato a
Partecipante	Chi prende parte ai corsi	Studente	Corso, Società
Docente	L'insegnante dei corsi. Potrebbe essere un dipendente interno.	Insegnante	Corso
Corso	Corso interno. Può avere diverse edizioni.	Workshop	Docente
Azienda	Luogo di lavoro attuale (o passato) del partecipante.	Luogo	Partecipante

Esempio Pratico: Nel tuo schema Prisma o MongoDB:

- `Participant` potrebbe diventare `model Student {}` o una collection `students`.
- Il glossario ti aiuta a decidere se `Lecturer` e `Teacher` sono la stessa cosa e come chiamare l'entità/collection (`model Teacher {}`).
- `Course` e `Workshop` sono sinonimi per la stessa entità/collection.

7.8 Strutturare i Requisiti

Dopo la raccolta, i requisiti vanno organizzati in gruppi omogenei di frasi. L'esempio dell'azienda di formazione viene strutturato:

- **Frasi generali**: "L'azienda richiede un DB per corsi, lezioni, insegnanti."
- **Frasi sui partecipanti**: Dettagli sugli studenti (ID, CF, nome, età, datori lavoro, corsi frequentati, ecc.).
- **Frasi specifiche sui partecipanti**: Dettagli per freelance (area interesse) o dipendenti di organizzazioni (livello gerarchico).
- **Frasi sul datore di lavoro**: Dettagli sui datori di lavoro dei partecipanti (nome, indirizzo, telefono).
- **Frasi sui corsi**: Dettagli sui corsi (codice, titolo, edizioni, date, n° partecipanti, aule, orari).
- **Frasi sui docenti**: Dettagli sugli insegnanti (cognome, età, corsi insegnati, tipo contratto, ecc.).

Questa strutturazione aiuta a identificare le future entità e le loro proprietà.

7.9 Dai Requisiti agli Schemi Concettuali (E-R)

Come tradurre i termini identificati nei costrutti del modello Entità-Relazione (E-R)?

- **Entità (`Entity`)**:
  - Se il termine ha **proprietà rilevanti** e descrive **oggetti autonomi**.
  - *Esempio*: `Studente`, `Corso`, `Docente`.
  - *Prisma/SQL*: Diventeranno tabelle (`model Student {}`, `CREATE TABLE Student (...)`).
  - *MongoDB*: Diventeranno collections (`db.students`).
- **Attributo (`Attribute`)**:
  - Se è un termine **semplice senza ulteriori specificazioni** (proprietà di un'entità).
  - *Esempio*: `Nome` dello Studente, `Titolo` del Corso.
  - *Prisma/SQL*: Diventeranno colonne nelle tabelle (`name: String`, `title: String`).
  - *MongoDB*: Diventeranno campi nei documenti (`{ name: "Mario", title: "Database 101" }`).

- **Relazione (`Relationship`)**:
  - Quando un termine **collega altri termini** (entità).
  - *Esempio*: "Studente *si iscrive a* Corso".
  - *Prisma/SQL*: Spesso implementate con chiavi esterne e tabelle di join.

```
model Student {
id          Int      @id @default(autoincrement())
// ... altri attributi
enrollments Enrollment[]
}
model Course {
id          Int      @id @default(autoincrement())
// ... altri attributi
enrollments Enrollment[]
}
model Enrollment { // Tabella di join
studentId Int
courseId  Int
student   Student @relation(fields: [studentId], references: [id])
course    Course  @relation(fields: [courseId], references: [id])
enrollmentDate DateTime
@@id([studentId, courseId])
}
```

  - *MongoDB*: Spesso implementate con DBRefs, array di ID, o embedding (se la relazione è 1-a-pochi e i dati sono strettamente legati).
- **Generalizzazione (`Generalization / ISA Relationship`)**:
  - Quando un termine è un **caso più generale di un altro**.
  - *Esempio*: `Persona` è una generalizzazione di `Studente` e `Docente`. Sia studenti che docenti sono persone e condividono attributi comuni (nome, cognome, CF) ma hanno anche attributi specifici.
  - *Prisma/SQL*: Ci sono varie strategie:
    1. Tabella unica con un campo "tipo" (es. `Person` con `personType: "Student" | "Teacher"`).
    2. Tabelle separate per le specializzazioni che referenziano una tabella base comune.
    3. Tabelle separate che duplicano gli attributi comuni (meno ideale per la consistenza).
  - *MongoDB*: Spesso si usa un campo `type` in una singola collection `people`, oppure collections separate se le differenze sono marcate.

7.10 Design Pattern E-R Comuni

Sono "best practices" per risolvere problemi comuni di modellazione.

7.10.1 Reificazione di Attributi in Entità

- **Problema**: Un attributo di un'entità ha esso stesso delle proprietà o partecipa ad altre relazioni.
- **Esempio**: Inizialmente `Company` è un attributo (es. `companyName`) di `Employee`.
  - Se `Company` deve avere un suo indirizzo, partita IVA, o essere collegata ad altri `Employee` o a `Projects`, allora `Company` va "reificata" (resa concreta) come un'entità separata.
  - Si crea l'entità `Company` e una relazione `Job` (o `WorksFor`) tra `Employee` e `Company`.
- **Cardinalità**: Un `Employee` `(1,1)` lavora per una `Company`. Una `Company` `(1,N)` può avere molti `Employee`.
  - `(1,1)`: Esattamente uno.

- (1,N): Da uno a molti.
- (0,1): Zero o uno.
- (0,N): Da zero a molti.

- *Esempio Pratico*:
  - *Prima*: `model Employee { id Int @id; name String; companyName String; }`
  - *Dopo*:

```
model Employee {
id        Int     @id
name      String
companyId Int
company   Company @relation(fields: [companyId], references: [id])
}
model Company {
id        Int     @id
name      String
address   String? // Company ha i suoi attributi
employees Employee[]
}
```

7.10.2 Relazioni "Part-of" (Composizione e Aggregazione)

- Relazioni (1,N) che rappresentano "parte di".
- **Composizione (Composition)**: Forte dipendenza. La parte non può esistere senza il tutto.
  - Esempio: Cinema (1) è composto da Hall (N). Ogni Hall (1,1) appartiene a un solo Cinema. Se il cinema viene distrutto, le sale non esistono più.
- **Aggregazione (Aggregation)**: Debole dipendenza. La parte può esistere indipendentemente dal tutto.
  - Esempio: Team (1) è composto da Expert (N). Un Expert (0,1) può appartenere a un Team (o a nessuno). Se il team si scioglie, gli esperti esistono ancora.

7.10.3 "Instance-of"

- **Problema**: Distinguere una rappresentazione astratta/modello da una sua istanza concreta.
- **Esempi**:
  - Flight (astratto: rotta, orario generico) vs. ScheduledFlight (istanza: volo specifico di un giorno con data, aereo assegnato).
  - Tournament (astratto: nome del torneo) vs. Edition (istanza: edizione 2024 del torneo, con date specifiche).
- *Esempio Pratico*:
  - ProductTemplate (nome, descrizione generica) vs ProductInstance (SKU specifico, colore, taglia, data di produzione).
  - CourseDefinition (codice, nome, crediti) vs CourseOffering (anno accademico, semestre, docente, aula).

7.10.4 Reificazione di Relazioni Binarie

- **Problema**: Una relazione tra due entità ha essa stessa degli attributi o partecipa ad altre relazioni.
- **Esempio**: Studente - Esame - Lezione.
  - Inizialmente, `Exam` potrebbe essere una relazione tra `Student` e `Lecture`.
  - Se l'esame ha attributi come `Grade` (voto) e `Date`, allora `Exam` viene reificata come entità.
  - Si creano due relazioni binarie: `Student-takes-Exam (S-E)` e `Exam-is_for-Lecture (E-L)`.
- *Esempio Pratico (Prisma, vedi sopra per `Enrollment`)*: Se la relazione "studente si iscrive a corso" ha una data di iscrizione, un voto, ecc., la tabella di join `Enrollment` diventa un'entità reificata.

7.10.5 Reificazione di Relazioni Ricorsive

- **Problema**: Una relazione tra istanze della stessa entità ha attributi.
- **Esempio**: `Team` gioca una `Match` contro un altro `Team`.
  - Una relazione `PlaysAgainst` tra `Team` e `Team`.
  - Se la partita (`Match`) ha attributi come `Date`, `Score`, allora `Match` viene reificata come entità.
  - Si creano due relazioni: `Team_Home-plays-Match` e `Team_Visiting-plays-Match`.
- *Esempio Pratico*: Un `Employee` può essere `manager` di altri `Employee`. Se questa relazione di management ha una `startDate` o un `roleDescription`, si potrebbe reificare in un'entità `ManagementRelationship`.

7.10.6 Reificazione di Attributi di Relazioni

- **Problema**: Un attributo di una relazione molti-a-molti ha esso stesso delle proprietà.
- **Esempio**: `Player` (musicista) - `Plays` (suona) - `Orchestra`. La relazione `Plays` ha un attributo `Instrument`.
  - Se `Instrument` (es. "Violino Stradivari Modello X") deve avere attributi propri (es. `Trademark`, `Type`, anno di fabbricazione) o essere suonato da più musicisti in diverse orchestre, allora `Instrument` va reificato.
  - Si crea l'entità `Instrument` e la relazione `Plays` diventa ternaria (o si reifica `Plays` in un'entità che collega `Player`, `Orchestra`, `Instrument`).

7.10.7 Caso Specifico (Generalizzazione/ISA)

- **Problema**: Una sottocategoria di un'entità ha caratteristiche o relazioni aggiuntive.
- **Esempio**: `Manager` è un caso specifico di `Employee`.
  - Tutti i `Manager` sono `Employee`, ma solo i `Manager` gestiscono (`Manage`) dei `Project`.
  - Non tutti gli `Employee` gestiscono progetti.
- Si usa una freccia di generalizzazione (concettualmente) da `Manager` a `Employee`.

7.10.8 Storicizzazione di un Concetto

- **Problema**: Necessità di tracciare i cambiamenti di un concetto nel tempo.
- **Esempi**:
  - `Anagraphic` (dati anagrafici) può avere una versione `Historic` e una `Current`. Si usano attributi come `StartDate`, `ExpiryDate`.
  - `Software` può avere una versione `Legacy` e una `Updated`.
  - **Impiego (`Employment`)**: Si può modellare `CurrentEmployment` e `PastEmployment`.

    - *Opzione 1*: Due relazioni separate (`CurrentEmployment`, `PastEmployment`) tra `Employee` e `Company`.
    - *Opzione 2*: Reificare `Employment` come entità con `BeginDate`, `EndDate` e poi generalizzarla in `CurrentEmployment` e `PastEmployment` (o usare un attributo di stato).
- *Esempio Pratico*: Per tracciare lo storico degli indirizzi di un cliente:

```
model Customer {
id              Int       @id
// ...
addressHistory  Address[]
}
model Address {
id          Int       @id
street      String
city        String
customerId  Int
customer    Customer  @relation(fields: [customerId], references: [id])
startDate   DateTime
endDate     DateTime? // Null se è l'indirizzo corrente
}
```

7.10.9 Estensione di un Concetto (Generalizzazione/ISA)

- **Problema**: Un concetto esistente viene esteso con nuove informazioni per casi specifici.
- **Esempio**: `Project` è un concetto generale. Un `AcceptedProject` (progetto accettato) richiede informazioni aggiuntive come `Founding` (finanziamento) e `StartDate`, che non sono necessarie per progetti in attesa o rifiutati.
- Si usa una generalizzazione (concettualmente) da `AcceptedProject` a `Project`.

7.10.10 Relazioni Ternarie e Loro Reificazione

- **Relazione Ternaria**: Coinvolge tre entità.
    - Esempio: `Employee` lavora su un `Task` in un `Office`. La relazione `Work` collega queste tre.
    - Le cardinalità indicano che un impiegato può lavorare su più task in più uffici, un ufficio può ospitare più impiegati su più task, e un task può essere svolto da più impiegati in più uffici.
- **Reificazione di Relazione Ternaria (1)**:
    - La relazione ternaria `Work` viene trasformata in un'entità `Work`.
    - L'entità `Work` è collegata a `Employee`, `Office`, e `Task` tramite tre relazioni binarie (E-W, O-W, T-W).
    - Questo è utile se l'evento "Work" ha attributi propri (es. `duration`, `status`).
- **Reificazione di Relazione Ternaria (2) - Semplificata**:
    - Se ci sono vincoli specifici (es. "un task può essere eseguito da un solo operatore e in un solo ufficio"), il modello può essere semplificato.
    - Nell'esempio delle slide, `Task` diventa centrale, con una relazione `(1,1)` verso `Employee` (tramite O-S, probabilmente "Operator-for-Service") e `(1,1)` verso `Office` (tramite S-I, probabilmente "Service-at-Location").
    - *Nota*: Le etichette delle relazioni (O-S, S-I) sono un po' criptiche, ma il concetto è la semplificazione basata su vincoli.

7.11 Strategie di Progettazione dello Schema E-R

Come si affronta la creazione dello schema E-R?

7.11.1 Strategia Top-Down

1. Si parte dai concetti più generali (entità principali).
2. Si raffinano progressivamente aggiungendo dettagli:
    - Identificare attributi.
    - Identificare relazioni.
    - Scomporre entità complesse.
    - Introdurre generalizzazioni/specializzazioni.

Esempi:

- `Exam` (iniziale) → `Student` - `Exam` (relazione) - `Lecture`.
- `Employee` (iniziale) → `Employee` con attributi `Surname`, `Age`, `Wage`.
- `People` (iniziale) → Generalizzazione in `Man` e `Woman`.

7.11.2 Strategia Bottom-Up

1. Si parte dai dettagli: attributi e concetti specifici.
2. Si raggruppano per formare entità e relazioni.
3. Si integrano i vari "pezzi" di schema per formare lo schema completo.

Esempi:

- Requisito su `Employee` → Entità `Employee`.
- Concetti `Student`, `Exam`, `Lecture` → Schema `Student-Exam-Lecture`.
- Entità `Man`, `Woman` → Generalizzazione `People`.

7.11.3 Strategia Inside-Out

1. Si identifica un concetto centrale e ben compreso.
2. Si espande lo schema "verso l'esterno", aggiungendo concetti (entità, attributi, relazioni) direttamente collegati a quelli già identificati.

Esempio (dalle slide):

1. Inizio: `Employee`.
2. Aggiungo attributi a `Employee`: `Surname`, `Code`.
3. Aggiungo `Dept` (Dipartimento) e le relazioni `Supervision` (Employee supervisiona Dept) e `Belonging` (Employee appartiene a Dept, con attributo `Date`).
4. Aggiungo `Project` e la relazione `Enrollment` (Employee partecipa a Project).
5. Aggiungo `Office` e la relazione `Composition` (Dept è composto da Office, con attributo `Addr` complesso).

7.12 Regola Pratica e Metodologia

7.12.1 Regola Pratica: Usare uno Stile Misto!

1. **Crea uno "schizzo" (`sketch`)**: Identifica le entità più rilevanti.
2. **Decomponi lo schema**: Dividi il problema se complesso.
3. **Raffina (top-down), integra (bottom-up), espandi (inside-out)**.

7.12.2 Sketching dello Schema E-R

- Parti dalle entità più rilevanti (più citate o esplicitamente indicate come tali).
- Crea un primo schema E-R di base.

7.12.3 Metodologia "Best Practice"

1. **Analisi dei Requisiti**:
    - Analizza, risolvi ambiguità.
    - Crea un glossario.
    - Raggruppa requisiti simili.
2. **Caso Base (`Base case`)**:
    - Definisci uno schema "abbozzato" con i concetti più rilevanti.
3. **Caso Iterativo (`Iterative case`)** (ripeti finché non va bene):
    - Raffina i concetti base usando i requisiti.
    - Aggiungi concetti per descrivere requisiti non ancora coperti.
4. **Analisi di Qualità (`Quality analysis`)** (ripeti durante tutto il processo):
    - Controlla la qualità dello schema e modificalo.

7.13 Qualità dello Schema E-R

Misure di qualità per uno schema E-R:

- **Correttezza (`Correctness`)**: Lo schema rappresenta accuratamente i requisiti? Usa i costrutti E-R in modo appropriato?
- **Completezza (`Completeness`)**: Tutti i requisiti sono stati rappresentati nello schema? Tutti i dati necessari sono modellati?
- **Chiarezza (`Clarity`)**: Lo schema è facile da capire? È ambiguo?
- **Minimalità (`Minimality`)**: Ci sono elementi ridondanti (entità, attributi, relazioni non necessarie)? Si potrebbe rappresentare la stessa informazione in modo più semplice?

7.14 Best Practice e Integrazione di Schemi

Per sistemi complessi, si può decomporre il problema:

7.14.1 Approccio 1

1. Analisi dei Requisiti.
2. Caso Base (schema "scheletro" generale).
3. **Decomposizione**: Suddividi i requisiti complessi secondo lo schema scheletro.
4. Caso Iterativo per ogni **sotto-schema**.
5. **Integrazione**: Unisci i sotto-schemi in uno schema totale, usando lo schema scheletro come riferimento.
6. Analisi di Qualità.

7.14.2 Approccio 2

1. Analisi dei Requisiti.
2. **Decomposizione**: Identifica aree di interesse e partiziona i requisiti (o acquisiscili separatamente per area).
3. **Per ogni area**:
   - Caso Base.
   - Caso Iterativo.
4. **Integrazione**: Unisci gli schemi delle varie aree.
5. Analisi di Qualità.

7.15 Esempio Finale: Azienda di Formazione

Questo è un'applicazione pratica della metodologia all'esempio dell'azienda di formazione.

7.15.1 Affermazione Generale

"Azienda di formazione richiede DB per corsi, lezioni, insegnanti."

7.15.2 Schema Abbozzato (`Sketched Schema`)

- Entità: `Participant`, `Lecture` (Lezione/Corso), `Lecturer` (Docente).
- Relazioni: `Presence` (Participant - Lecture), `Teaching` (Lecturer - Lecture).

7.15.3 Raffinamento: Partecipanti e Datori di Lavoro

- Requisiti: ID, CF, dati anagrafici, datori di lavoro (passati/presenti), corsi frequentati. Freelance vs. Dipendenti.
- **Schema Parziale (1)** (basato sulla slide 70):
  - Entità `Participant` con attributi (`Tax`, `Code`, ...).
  - Generalizzazione: `Participant` è generalizzazione di `Employee` (con attributi `Level`, `Position`) e `Freelance` (con `Title`, `Area`).
  - Entità `Employer` (Datore di lavoro) con attributi (`Name`, ...).
  - Relazioni: `CurrEmpl` (tra `Participant` e `Employer` per impiego attuale, `1-a-N`), `PastEmpl` (tra `Participant` e `Employer` per impieghi passati, `N-a-N`, reificata).

7.15.4 Raffinamento: Corsi

- Requisiti: Corsi (∼200) con codice, titolo, edizioni (con data inizio/fine, n° partecipanti), lezioni (giorno, aula, orario).
- **Schema Parziale (2)** (basato sulla slide 72):
  - Pattern "Instance-of": `Lecture` (Corso generico) e `Edition` (Edizione specifica del corso).
    - `Lecture` (Attributi: `Title`, `Code`).
    - `Edition` (Attributi: `Start`, `End`, `#Part.` - numero partecipanti).
    - Relazione `KindOf` (`1,1`) tra `Edition` e `Lecture` (un'edizione è di un solo tipo di corso).
  - Pattern "Part-of": `Edition` è composta da `Lesson` (singola lezione).
    - `Lesson` (Attributi: `Time`, `Room`, `Day`).
    - Relazione `MadeOf` (`1,N`) tra `Edition` e `Lesson`.

7.15.5 Raffinamento: Docenti

- Requisiti: Docenti (∼300) con dati anagrafici, corsi insegnati (passati/futuri), storico telefonate. Dipendenti interni vs. Esterni.
- **Schema Parziale (3)** (basato sulla slide 74):
  - Entità `Lecturer` con attributi (`Tax`, `Surname`, `Age`, `Place of Birth`, `Phone` (multi-valore, `1,N`)).
  - Generalizzazione: `Lecturer` è generalizzazione di `Independent` (esterno) e `Home` (interno).

7.15.6 Integrazione dello Schema

Si parte dallo schema abbozzato e si integrano i raffinamenti.

- **Schema Intermedio (1)** (basato sulla slide 76): Integrazione di `Participant` e `Lecture`.
  - Relazioni `PastPresence` (`0,N`)-(`0,N`) e `CurrentPresence` (`0,1`)-(`0,N`) tra `Participant` e `Lecture`. (Questo modella la frequenza ai corsi/lezioni).
- **Schema Intermedio (2)** (basato sulla slide 77): Integrazione di `Lecturer`, `Lecture`, `Edition`.
  - Relazioni `Past` (`0,1`)-(`0,N`) e `Current` (`0,1`)-(`0,N`) tra `Edition` e `Lecturer` (per insegnamento).
  - Relazione `Duty` (`1,N`)-(`0,N`) tra `Lecturer` e `Lecture` (per indicare i corsi che un docente *può* insegnare o *ha insegnato* in generale, separato dalle specifiche edizioni).

7.15.7 Schema Finale (Solo Entità e Relazioni)

La slide 78 mostra la struttura complessiva integrando tutti i pezzi, omettendo gli attributi per chiarezza. Si vedono chiaramente:

- `Participant` generalizzato in `Employee` e `Freelance`.
- `Employee` collegato a `Employer`.
- `Lecturer` generalizzato in `Independ.` e `Home`.
- Il nucleo `Lecture` → `Edition` → `Lesson`.
- Le relazioni che collegano `Participant` a `Lecture`/`Edition` (frequenza).
- Le relazioni che collegano `Lecturer` a `Lecture`/`Edition` (insegnamento).

Questo processo iterativo di sketching, raffinamento e integrazione, guidato dai requisiti e supportato da un glossario e da pattern di progettazione, porta a uno schema concettuale robusto.

Capitolo 8

Progettazione Logica dei Database

8.1 Introduzione alla Progettazione Logica dei Database

L'obiettivo principale della **progettazione logica** è "tradurre" lo schema concettuale (spesso un diagramma E-R) in uno schema logico (ad esempio, per un database relazionale come SQL Server, MySQL, PostgreSQL) che rappresenti gli stessi dati in modo **corretto ed efficiente**.

8.1.1 Input della Progettazione Logica

- **Schema Concettuale:** Il diagramma E-R che descrive la realtà di interesse.
- **Informazioni sul Carico Applicativo (Workload):** Quali operazioni verranno eseguite più frequentemente? Quanti dati ci aspettiamo? (Fondamentale per l'efficienza).
- **Modello Logico Scelto:** Ad es. relazionale, a oggetti, a grafo. Ci concentreremo sul relazionale.

8.1.2 Output della Progettazione Logica

- **Schema Logico:** Ad esempio, un insieme di istruzioni `CREATE TABLE` per SQL.
- **Documentazione Associata:** Spiegazioni delle scelte fatte.

8.1.3 Non è una semplice traduzione!

- Alcuni aspetti dello schema concettuale potrebbero non essere rappresentabili direttamente nel modello logico scelto (es. generalizzazioni nel modello relazionale puro).
- Bisogna considerare le **prestazioni (efficienza)**.

8.2 Fasi della Trasformazione Logica

1. **Ristrutturazione dello Schema E-R (E-R Schema Restructuring):**
    - Si modifica lo schema E-R iniziale tenendo conto del carico applicativo e del modello logico.
    - **Perché?**
        - Semplificare la successiva traduzione.
        - Ottimizzare le prestazioni.
    - **Nota Bene:** Uno schema E-R ristrutturato *non è più* uno schema concettuale puro, perché inizia a includere considerazioni di implementazione.
2. **Traduzione nel Modello Logico:**
    - Si converte lo schema E-R ristrutturato nello schema del modello scelto (es. tabelle relazionali).

8.3 Analisi delle Prestazioni (Approssimata)

A livello concettuale/logico iniziale, non possiamo valutare le prestazioni con precisione assoluta, ma usiamo degli "indicatori":

- **Spazio (Space):** Numero di istanze (righe nelle tabelle) attese.
  - *Esempio Pratico:* Se hai una tabella `Utenti` e prevedi 1 milione di utenti, questo è un indicatore di spazio.
- **Tempo (Time):** Numero di istanze (entità e relazioni) "visitate" (lette/scritte) durante un'operazione.
  - *Esempio Pratico:* Per mostrare il profilo di un utente con i suoi ultimi 10 post e i commenti a quei post, quante righe da diverse tabelle (`Utenti`, `Post`, `Commenti`) devi leggere?

La **Tabella delle Dimensioni (Size Table)** stima il numero di istanze per ogni entità (E) e relazione (R). La **Tabella degli Accessi (Access Table)**, per un'operazione specifica, elenca:

- Quali entità/relazioni vengono accedute.
- Quante volte (numero accessi).
- Tipo di accesso (Lettura/Scrittura).
- Ordine di accesso.

Questo aiuta a confrontare diverse opzioni di ristrutturazione.

8.4 Attività di Ristrutturazione dello Schema E-R

Sono 4 attività principali:

8.4.1 Analisi delle Ridondanze

- Una **ridondanza** è un'informazione rilevante che può essere derivata da altre informazioni già presenti.
- Bisogna decidere se: mantenere, rimuovere o *creare nuove* ridondanze.
- **Pro della Ridondanza:**
  - Semplifica le query (meno join, calcoli al volo).
  - *Esempio Pratico:* In una tabella `Ordini`, potresti avere una colonna `TotaleOrdine`. Questo è ridondante se puoi calcolarlo sommando `Prezzo * Quantità` da una tabella `RigheOrdine` collegata. Averlo precalcolato velocizza la lettura del totale ordine.
- **Contro della Ridondanza:**
  - Gli aggiornamenti richiedono più tempo (devi aggiornare il dato in più posti e mantenerlo consistente).
  - Aumenta lo spazio di archiviazione.
  - *Esempio Pratico (continuazione):* Se modifichi una riga in `RigheOrdine`, devi ricalcolare e aggiornare `TotaleOrdine` nella tabella `Ordini`. Se non lo fai, i dati diventano inconsistenti.
- **Tipi di Ridondanze comuni in E-R:**
  - **Attributi derivabili:**
    - Da altri attributi nella stessa entità/relazione (es. `Fattura` con `ValoreNetto`, `IVA`, `ValoreLordo`. `ValoreLordo` è derivabile).
    - Da attributi di altre entità/relazioni (es. `Acquisto` con attributo `Totale`, derivabile da $\sum$(`Composizione.Quanti` `Prodotto.Prezzo`)).

  - **Relazioni derivabili:** Spesso cicli di relazioni (es. se hai `Studente - Frequenta - Lezione - TenutaDa - Docente`, una relazione diretta `Studente - InsegnatoDa - Docente` potrebbe essere ridondante).
  - **Attributi derivabili da conteggio:** (es. `Citta` con attributo `NumeroAbitanti`, derivabile da `COUNT(*)` delle persone che risiedono in quella città).
- **Decisione sulla Ridondanza:** Si basa sull'analisi costi/benefici, considerando la frequenza delle operazioni.
  - Se un dato derivato è letto molto frequentemente e scritto/aggiornato raramente, mantenerlo ridondante può essere vantaggioso.
  - Se è aggiornato spesso, la ridondanza può diventare problematica.

8.4.2 Eliminazione delle Generalizzazioni (Gerarchie)

- Il modello relazionale puro non supporta direttamente le generalizzazioni (ereditarietà). Vanno trasformate.
- **Tre Soluzioni Possibili** (esempio: E0 genitore, E1 ed E2 figli):
  1. **Embedding dei Figli nel Genitore (Roll-up / Accorpamento verso l'alto):**
     - Si elimina E1 ed E2. L'entità E0 eredita tutti gli attributi di E1 ed E2.
     - Si aggiunge un attributo "Tipo" (o "Kind") a E0 per distinguere se un'istanza era originariamente E1 o E2.
     - Gli attributi specifici di E1 o E2 diventano `NULL`abili in E0 se un'istanza non è di quel tipo.
     - Le relazioni dei figli vengono "spostate" sul genitore.
     - *Esempio Pratico:* `Veicolo` (genitore), `Auto` (figlio), `Moto` (figlio). Diventa un'unica tabella `VEICOLI(targa, marca, modello, tipoVeicolo, numPorte_auto, cilindrata_moto, ...)`. `numPorte_auto` sarà `NULL` per le moto.
     - **Quando usarla:** Se le operazioni accedono frequentemente al genitore e ai figli contemporaneamente.
  2. **Embedding del Genitore nei Figli (Roll-down / Accorpamento verso il basso):**
     - Si elimina E0. Le entità E1 ed E2 ereditano tutti gli attributi di E0.
     - Le relazioni di E0 vengono replicate per E1 ed E2.
     - *Esempio Pratico:* Tabelle separate `AUTO(targa, marca, modello_veicolo, numPorte, ...)` e `MOTO(targa, marca, modello_veicolo, cilindrata, ...)`. `marca` e `modello_veicolo` sono duplicati.
     - **Quando usarla:** Se le operazioni accedono ai figli indipendentemente l'uno dall'altro.
  3. **Sostituzione della Generalizzazione con Relazioni (Una tabella per classe):**
     - Si mantengono E0, E1, E2 come entità separate.
     - Si creano relazioni 1-a-1 tra E0 ed E1, e tra E0 ed E2. La chiave primaria di E1 (e E2) sarà anche chiave esterna verso E0.
     - *Esempio:* Tabella `VEICOLI(targa_PK, marca, modello)`. Tabella `AUTO(targa_FK_PK, numPorte)`. Tabella `MOTO(targa_FK_PK, cilindrata)`. Per avere tutti i dati di un'auto, fai un `JOIN`.
     - **Quando usarla:** Se i figli sono acceduti indipendentemente dal padre.
- **Soluzioni Ibride:** Si possono combinare queste strategie, specialmente con gerarchie a più livelli.

8.4.3 Partizionamento/Raggruppamento di Entità e Relazioni

L'obiettivo è ridurre il numero di accessi.

- **Partizionamento Verticale di Entità:**
  - Se un'entità ha molti attributi e alcuni sono usati frequentemente insieme, mentre altri raramente, si può dividere l'entità in due (o più) entità collegate da una relazione 1-a-1.

  - *Esempio Pratico:* `Impiegato(Codice, Cognome, Indirizzo, DataNascita, Livello, Salario, Tasse)` può diventare `DatiPersonali(Codice_PK, Cognome, Indirizzo, DataNascita)` e `DatiAziendali(Codice_PK_FK, Livello, Salario, Tasse)`.

- **Ristrutturazione di Attributi Multi-Valore:**
  - Un attributo multi-valore (es. `Ufficio` con `Telefono(1,N)`) viene trasformato in una nuova entità (`Telefono`) e una relazione 1-a-N (`Possiede`).
  - *Esempio Pratico:* Se un `Prodotto` può avere più `Tag`, crei una tabella `Prodotti`, una tabella `Tag` e una tabella di join `ProdottoTag`.

- **Raggruppamento di Entità (Denormalizzazione):**
  - Se un'entità A ha una relazione 1-a-1 (o N-a-1) con un'entità B, e sono *sempre* accedute insieme, gli attributi di B possono essere "assorbiti" in A.
  - *Esempio Pratico:* `Persona(0,1) -- ViveIn -- (1,1)Appartamento`. Si possono spostare `NumAppartamento` e `IndirizzoAppartamento` nell'entità `Persona`, rendendoli `NULL`abili.

- **Partizionamento Orizzontale di Relazioni:**
  - Si dividono le istanze di una relazione in più relazioni distinte se hanno pattern di accesso diversi.
  - *Esempio Pratico:* Relazione `ParteDi` tra `Giocatore` e `Squadra` può essere divisa in `MilitaAttualmenteIn` e `HaMilitatoInPassato`.

8.4.4 Identificazione delle Chiavi Primarie

- Operazione **obbligatoria** per la traduzione in un modello relazionale.
- **Criteri di Scelta:**
  1. **Informazione Obbligatoria:** L'attributo deve essere `NOT NULL`.
  2. **Semplicità:** Preferire chiavi singole a chiavi composite.
  3. **Usata nelle Operazioni più Frequenti/Rilevanti.**
- **Nuovi Attributi (Surrogate Keys):** Se nessuna combinazione di attributi esistenti è una buona chiave primaria, si introducono nuovi attributi "artificiali".
  - *Esempio Pratico:* `id INT AUTO_INCREMENT PRIMARY KEY` in SQL, `ObjectId()` in MongoDB.

8.5 Traduzione nel Modello Relazionale (Regole Generali)

- **Entità:** Diventano tabelle. Gli attributi dell'entità diventano colonne. La chiave primaria identificata diventa la `PRIMARY KEY`.
- **Relazioni:**
  - **Molti-a-Molti (M:N):**
    - Diventano una **nuova tabella** (tabella di associazione).
    - Colonne: chiavi primarie delle entità coinvolte (come `FOREIGN KEY`, formano la `PRIMARY KEY` della nuova tabella) + attributi propri della relazione.
    - *Esempio:* `IMPIEGATO(0,N) -- ISCRIZIONE(DataInizio) -- (1,N)PROGETTO`

```
IMPIEGATO(Numero_PK, Cognome, Salario)
PROGETTO(Codice_PK, Nome, Budget)
ISCRIZIONE(NumeroImpiegato_FK_PK, CodiceProgetto_FK_PK, DataInizio)
```

    - **Vincoli di Integrità Referenziale:** Vanno definiti.
    - **Nomi Espressivi per FK:** Meglio `IDImpiegato`, `IDProgetto`.

    - **Cardinalità Minima:** La traduzione base M:N *non* cattura la cardinalità minima. Richiede logica applicativa o `TRIGGER/CHECK` complessi.
- **Relazioni Ricorsive (M:N sulla stessa entità):**
    - Simile a M:N, si crea una nuova tabella con due chiavi esterne che puntano alla stessa tabella originale.
    - *Esempio:* `PRODOTTO -- CompostoDa(NumeroPezzi) -- PRODOTTO`

```
PRODOTTO(Codice_PK, Nome, Costo)
COMPOSTODA(CodiceProdottoComposto_FK_PK, CodiceComponente_FK_PK, NumeroPezzi)
```

- **Relazioni N-arie (coinvolgono 3 o più entità):**
    - Diventano una nuova tabella con le chiavi primarie di tutte le entità coinvolte (come FK) + attributi propri.
    - *Esempio:* `FORNITORE -- FORNITURA(NumeroPezzi) -- PRODOTTO -- A REPARTO`

```
FORNITURA(ID_Fornitore_FK_PK, ID_Prodotto_FK_PK, ID_Reparto_FK_PK, NumeroPezzi)
```

- **Uno-a-Molti (1:N):**
    - La chiave primaria dell'entità sul lato "1" viene **propagata** come `FOREIGN KEY` nell'entità sul lato "N".
    - Gli attributi della relazione vengono aggiunti alla tabella sul lato "N".
    - *Esempio:* `GIOCATORE(1,1) -- CONTRATTO(DataIngaggio) -- (0,N)SQUADRA`

```
SQUADRA(Nome_PK, Citta, ColoriSociali)
GIOCATORE(CF_PK, Cognome, DataNascita, Ruolo, NomeSquadra_FK, DataIngaggio)
```

    - **Cardinalità Minima (0 sul lato N):** Se opzionale, la `FOREIGN KEY` permette `NULL`.
    - **Cardinalità Minima (1 sul lato N):** Se obbligatoria, la `FOREIGN KEY` è `NOT NULL`.
- **Entità con Identificatore Esterno (Entità Debole):**
    - L'entità debole diventa una tabella la cui chiave primaria è composta dalla PK dell'entità forte + identificatore parziale.
    - *Esempio:* `STUDENTE(Matricola)` identificato da `UNIVERSITA(Nome)`.

```
UNIVERSITA(Nome_PK, Citta, Indirizzo)
STUDENTE(NomeUniversita_FK_PKpart, Matricola_PKpart, Cognome, AnnoCorso)
```

    La PK di `STUDENTE` è `(NomeUniversita, Matricola)`.
- **Uno-a-Uno (1:1):**
    - Si sceglie una delle due tabelle e si aggiunge la PK dell'altra come `FOREIGN KEY` e `UNIQUE KEY`. Gli attributi della relazione vanno nella tabella scelta.
    - *Esempio:* `CAPO(1,1) -- SUPERVISIONE(DataInizio) -- (1,1)DIPARTIMENTO`

```
-- Opzione 1:
DIPARTIMENTO(Nome_PK, Ufficio, Telefono)
CAPO(Codice_PK, Cognome, Salario, NomeDipartimento_FK_UNIQUE,
↪  DataInizioSupervisione)

-- Opzione 2:
CAPO(Codice_PK, Cognome, Salario)
DIPARTIMENTO(Nome_PK, Ufficio, Telefono, CodiceCapo_FK_UNIQUE,
↪  DataInizioSupervisione)
```

* **Cardinalità (0,1) - Opzionale:**
  Se una partecipazione è opzionale, la `FOREIGN KEY` è `NULL`abile (ma sempre `UNIQUE` se presente).
  Se entrambe sono `(0,1)`: la soluzione più pulita è una tabella separata per la relazione `SUPERVISIONE(CodiceCapo_FK_PK, NomeDipartimento_FK_PK, DataInizio)`.

8.6 Attenzione Finale

Piccole differenze nelle cardinalità e nelle scelte degli identificatori nello schema E-R possono portare a significati e schemi logici molto diversi. È fondamentale essere precisi.

8.7 Strumenti (Tools)

Esistono software CASE (Computer-Aided Software Engineering) che supportano la modellazione, come:

- ERwin/ERX
- IBM Rational Rose

Questi strumenti aiutano a disegnare diagrammi E-R e a generare lo schema DDL.

Capitolo 9

Normalizzazione dei Database

9.1 Normalizzazione nel Contesto dei Database

La **normalizzazione** è un processo fondamentale nella progettazione di database relazionali. Il suo scopo principale è organizzare i dati in modo da:

1. **Ridurre la ridondanza:** Evitare di ripetere le stesse informazioni in più punti.
2. **Eliminare le anomalie:** Prevenire problemi che possono sorgere durante l'inserimento, l'aggiornamento o la cancellazione dei dati.
3. **Garantire la qualità e l'integrità dei dati:** Assicurare che i dati siano coerenti e affidabili.

Le **Forme Normali (FN)** sono un insieme di regole che definiscono quanto "ben formata" è una tabella (relazione). Se una relazione non è in una forma normale adeguata, può presentare:

- **Ridondanze:** Dati duplicati inutilmente.
- **Comportamenti indesiderati durante gli aggiornamenti:** Ad esempio, la necessità di modificare lo stesso dato in più righe, con il rischio di dimenticarne qualcuna e creare inconsistenza.

La normalizzazione è una **tecnica di verifica** del design del database, non una metodologia di progettazione da zero. Prima progetti lo schema (magari con un modello E-R), poi lo verifichi e lo affini con la normalizzazione.

9.1.1 Esempio di Tabella con Anomalie

Consideriamo una tabella che traccia impiegati, progetti a cui lavorano, i loro stipendi, il budget dei progetti e il loro ruolo nel progetto:

Employee	Wage	Project	Budget	Role
Jones	20	Mars	2	Technician
Smith	35	Jupiter	15	Designer
Smith	35	Venus	15	Designer
Williams	55	Venus	15	Chief
Williams	55	Jupiter	15	Consultant
Williams	55	Mars	2	Consultant
Brown	48	Mars	2	Chief
Brown	48	Venus	15	Designer
White	48	Venus	15	Designer
White	48	Jupiter	15	Director

Questa tabella presenta diversi problemi (anomalie):

1. **Ridondanza:**
    - Lo stipendio (`Wage`) di un impiegato (es. Smith, 35) è ripetuto per ogni progetto a cui lavora.

- Il budget (`Budget`) di un progetto (es. Jupiter, 15) è ripetuto per ogni impiegato che ci lavora.

2. **Anomalia di Aggiornamento (Update Anomaly):**
   - Se lo stipendio di Smith cambia, dobbiamo aggiornarlo in *tutte* le righe in cui Smith compare. Se ne dimentichiamo una, il database diventa inconsistente.
3. **Anomalia di Cancellazione (Deletion Anomaly):**
   - Se Jones smette di lavorare al progetto Mars (e Mars era il suo unico progetto), cancellando quella riga potremmo perdere l'informazione che Jones ha uno stipendio di 20 (se non ci sono altre tabelle che lo tracciano).
   - Similmente, se il progetto Mars viene cancellato e Jones e Brown lavoravano solo a Mars, perderemmo le informazioni su Jones e Brown.
4. **Anomalia di Inserimento (Insertion Anomaly):**
   - Non possiamo inserire un nuovo impiegato con il suo stipendio se non è ancora assegnato a un progetto.
   - Non possiamo inserire un nuovo progetto con il suo budget se nessun impiegato ci sta ancora lavorando.

9.1.2 Perché questa situazione è indesiderabile?

Perché stiamo mescolando diversi "concetti" o "pezzi di informazione" nella stessa tabella:

- Informazioni sugli impiegati e i loro stipendi.
- Informazioni sui progetti e i loro budget.
- Informazioni sul ruolo di un impiegato *all'interno di uno specifico progetto*.

9.2 Dipendenze Funzionali (Functional Dependencies - FD)

Per studiare e risolvere queste anomalie in modo sistematico, introduciamo il concetto di **Dipendenza Funzionale (FD)**. Una FD è un vincolo di integrità che descrive una relazione tra attributi all'interno di una tabella.

9.2.1 Definizione Formale

Data una relazione r con uno schema $R(X)$ (dove X è l'insieme di tutti gli attributi), e dati due sottoinsiemi non vuoti di attributi Y e Z (contenuti in X), esiste una dipendenza funzionale $Y \to Z$ (si legge "Y determina funzionalmente Z" o "Z dipende funzionalmente da Y") se e solo se: *Per ogni coppia di tuple (righe)* t_1 *e* t_2 *in* r*, se i valori degli attributi in* Y *sono uguali in* t_1 *e* t_2 *(cioè* $t_1[Y] = t_2[Y]$*), allora anche i valori degli attributi in* Z *devono essere uguali (cioè* $t_1[Z] = t_2[Z]$*).*

In parole povere: se conosci il valore di Y, puoi determinare *univocamente* il valore di Z.

9.2.2 Spiegazione in termini più semplici

"Determinare funzionalmente" significa semplicemente che se conosci il valore di un attributo, puoi conoscere con certezza il valore di un altro attributo.

Esempi dalla tabella precedente

- Employee $\to$ Wage
- Project $\to$ Budget
- $\{$Employee, Project$\}$ $\to$ Role

Spiegazione

- Employee $\rightarrow$ Wage significa: "Se sai chi è l'impiegato, sai sicuramente quanto guadagna". Nella nostra tabella, ogni volta che appare "Smith", il salario è sempre "35", ogni volta che appare "Jones" il salario è sempre "20". Il nome dell'impiegato *determina* univocamente il suo stipendio.
- Project $\rightarrow$ Budget significa: "Se sai qual è il progetto, sai sicuramente qual è il suo budget". Ogni volta che vedi "Mars" come progetto, il budget è sempre "2", ogni volta che vedi "Jupiter", il budget è sempre "15".

9.2.3 FD Triviali e Non Triviali

- Una FD $Y \rightarrow A$ è **triviale** se $A \subseteq Y$ (es. $\{$Employee, Project$\}$ $\rightarrow$ Project). Sono sempre vere e poco utili.
- Una FD $Y \rightarrow A$ è **non triviale** se $A \not\subseteq Y$. Sono queste che ci interessano per la normalizzazione.

Spiegazione semplice con esempi pratici

Dipendenza Funzionale Triviale - In termini semplici, è come dire "se conosci qualcosa, allora conosci anche una parte di quel qualcosa".

- **Esempio pratico:** In una tabella SQL `Clienti(ID, Nome, Cognome, Email)`, la dipendenza funzionale `{ID, Nome, Cognome} → Nome` è triviale perché ovviamente se conosci l'insieme {ID, Nome, Cognome}, allora conosci anche il Nome (che è già incluso nell'insieme).
- **In SQL:** Se scrivi `SELECT Nome FROM Clienti WHERE ID = 123 AND Nome = 'Mario' AND Cognome = 'Rossi'`, è ovvio che otterrai 'Mario' come risultato, perché è nelle condizioni stesse della query.

Dipendenza Funzionale Non Triviale - Significa che conoscendo alcuni attributi, puoi determinare altri attributi che *non* sono già contenuti nei primi.

- **Esempio pratico:** Nella tabella `Clienti(ID, Nome, Cognome, Email)`, la dipendenza `ID → Nome` è non triviale perché il Nome non è parte dell'ID e non è ovvio che l'ID determini il Nome.
- **In SQL:** Se scrivi `SELECT Nome FROM Clienti WHERE ID = 123`, otterrai un nome specifico perché esiste una dipendenza funzionale dall'ID al Nome (supponendo che ID sia una chiave primaria).

Perché le FD triviali non sono utili per la normalizzazione? Perché non rivelano nulla di nuovo sulla struttura dei dati. Sono sempre vere per definizione e non causano anomalie. Le dipendenze non triviali, invece, possono causare anomalie se non trattate correttamente.

9.2.4 Come le FD causano anomalie

Le anomalie sorgono principalmente quando abbiamo FD $X \rightarrow Y$ dove X **non è una superchiave** (o chiave candidata) della tabella.

- Employee $\rightarrow$ Wage: `Employee` da solo non è la chiave. Causa ridondanza.
- Project $\rightarrow$ Budget: `Project` da solo non è la chiave. Causa ridondanza.
- $\{$Employee, Project$\}$ $\rightarrow$ Role: $\{$Employee, Project$\}$ è (probabilmente) la chiave primaria. Questa FD **non causa anomalie**.

Le anomalie sono quindi causate dalla presenza di informazioni eterogenee.

9.3 Forma Normale di Boyce-Codd (BCNF)

La BCNF è una delle forme normali più stringenti e desiderabili.

9.3.1 Definizione

Una relazione r è in **BCNF** se, per ogni dipendenza funzionale non triviale $X \to Y$ definita su r:

- X è una **superchiave** di r.

Nella nostra tabella di esempio iniziale, non è in BCNF a causa di Employee $\to$ Wage e Project $\to$ Budget.

Spiegazione della violazione BCNF

Ricordiamo la definizione di BCNF: per ogni dipendenza funzionale non triviale $X \to Y$, X deve essere una superchiave della relazione.

Nel nostro esempio:

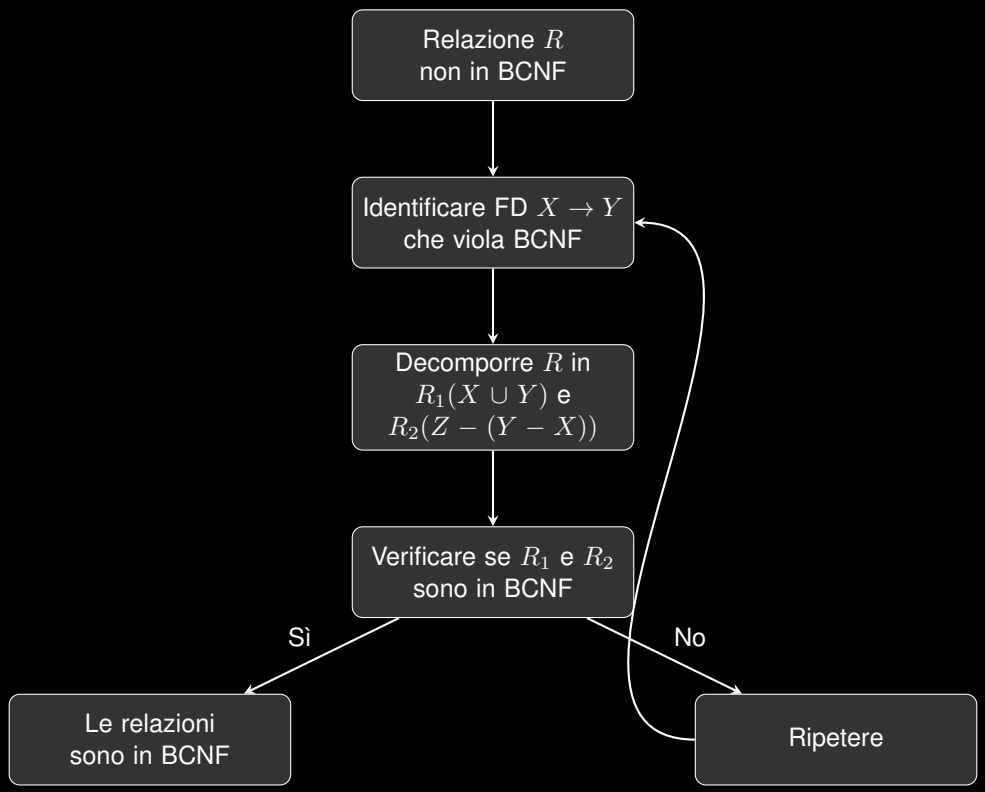
- Employee $\to$ Wage: L'attributo `Employee` determina funzionalmente `Wage`. Ma `Employee` da solo non è una superchiave della tabella, perché non può determinare univocamente tutti gli altri attributi (come `Project`, `Budget`, `Role`). La chiave primaria della tabella è {Employee, Project}.
- Project $\to$ Budget: Similmente, `Project` determina `Budget`, ma `Project` da solo non è una superchiave della tabella.

Queste violazioni causano le anomalie di inserimento, cancellazione e aggiornamento che abbiamo descritto in precedenza.

Perché questo viola BCNF? Perché BCNF richiede che quando un attributo (o gruppo di attributi) determina funzionalmente un altro attributo, il primo deve essere una "superchiave". In termini semplici, una superchiave è un attributo (o gruppo di attributi) che può identificare univocamente ogni riga nella tabella.

9.3.2 Cosa fare se una relazione non è in BCNF?

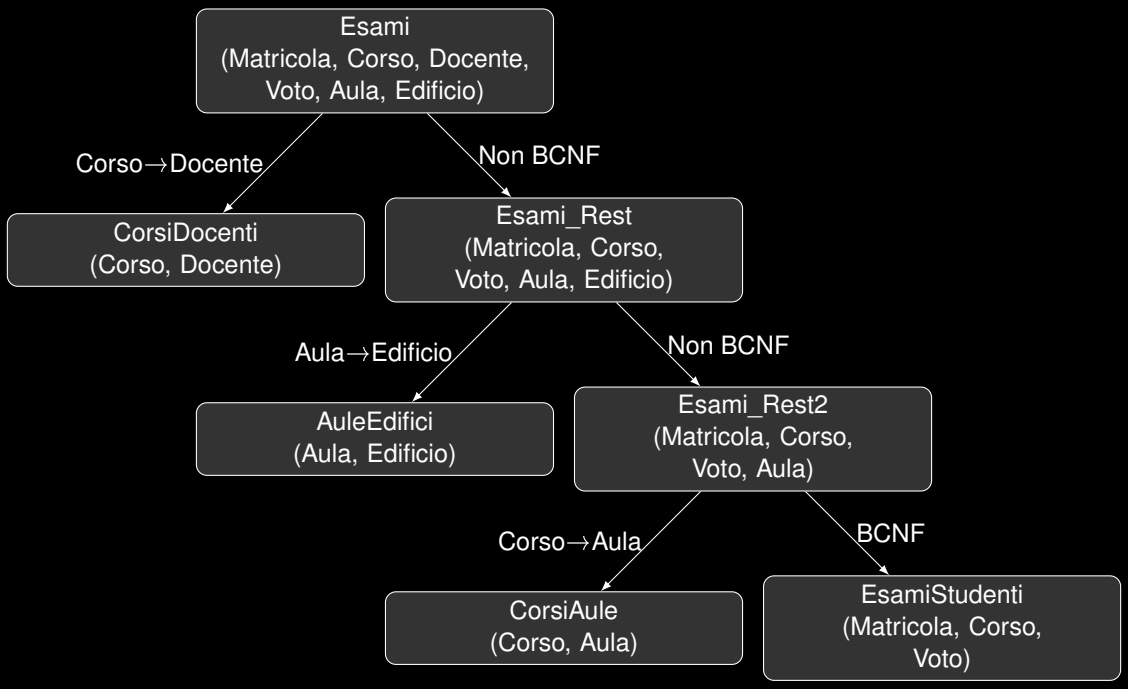
Si **decompone** la relazione in più relazioni più piccole, ognuna delle quali sia in BCNF.



Esempio pratico di decomposizione

```
Esami(Matricola, Corso, Docente, Voto, Aula, Edificio)
↓ FDs che violano BCNF:
* Corso → Docente     (Corso non è superchiave)
* Aula → Edificio     (Aula non è superchiave)
```

Decomposizione in BCNF:



Esempio con dipendenze più complesse

```
OrdiniFornitori(IDOrdine, CodiceArticolo, QuantitàOrdinata,
CodiceFornitore, RagioneSocialeFornitore, CittàFornitore)
FDs:
• IDOrdine → CodiceFornitore
• {IDOrdine, CodiceArticolo} → QuantitàOrdinata     (chiave)
• CodiceFornitore → {RagioneSocialeFornitore, CittàFornitore}
```

Ordini	Fornitori	DettagliOrdine
<u>IDOrdine</u> CodiceFornitore	<u>CodiceFornitore</u> RagioneSocialeFornitore CittàFornitore	<u>IDOrdine, CodiceArticolo</u> QuantitàOrdinata

9.3.3 Esempio di Decomposizione (per la tabella iniziale)

1. **ImpiegatiStipendi**(<u>Employee</u>, Wage)
2. **ProgettiBudget**(<u>Project</u>, Budget)
3. **ImpiegatiRuoliProgetto**(<u>Employee, Project</u>, Role)

Questa decomposizione elimina le anomalie.

9.3.4 Qualità della Decomposizione

Quando decomponiamo una tabella, dobbiamo assicurarci due proprietà fondamentali:

Lossless Join Property (Proprietà di Join Senza Perdita)

Dobbiamo essere in grado di ricreare la tabella originale facendo il JOIN delle tabelle decomposte. **Condizione:** Una decomposizione di $r(X)$ in $r_1(X_1)$ e $r_2(X_2)$ è senza perdita se l'intersezione degli attributi $X_0 = X_1 \cap X_2$ forma una chiave per almeno una delle relazioni decomposte ($X_0 \to X_1$ oppure $X_0 \to X_2$).

Esempio di Decomposizione CON PERDITA: Supponiamo R(Employee, Project, Office) con FD: Employee $\to$ Office e Project $\to$ Office. Decomposizione in:

- R_1(Employee, Office)
- R_2(Project, Office)

Tabella originale:

Employee	Project	Office
Smith	Alpha	A101
Jones	Beta	B202
Brown	Alpha	C303

Tabelle decomposte:

Employee	Office
Smith	A101
Jones	B202
Brown	C303

Project	Office
Alpha	A101
Beta	B202
Alpha	C303

Risultato del JOIN (genera tuple spurie):

Employee	Project	Office	Tuple Spurie
Smith	Alpha	A101	
Jones	Beta	B202	
Brown	Alpha	C303	
Smith	Alpha	C303	✓
Brown	Alpha	A101	✓

Attributo comune: `Office`. `Office` non è chiave né per R_1 né per R_2. Il join può generare tuple spurie.

Esempio di Decomposizione SENZA PERDITA: Tabella R(Employee, Project, Office) con FD: Employee $\to$ Office e chiave primaria {Employee, Project}. Decomposizione in:

- R_1(Employee, Office)
- R_2(Employee, Project)

Tabella originale:

Employee	Project	Office
Smith	Alpha	A101
Smith	Beta	A101
Jones	Gamma	B202

Tabelle decomposte:

Employee	Office
Smith	A101
Jones	B202

Employee	Project
Smith	Alpha
Smith	Beta
Jones	Gamma

Risultato del JOIN (senza tuple spurie):

Employee	Project	Office
Smith	Alpha	A101
Smith	Beta	A101
Jones	Gamma	B202

Attributo comune: `Employee`. `Employee` è chiave per R_1. Lossless.

Dependency Preservation (Conservazione delle Dipendenze)

Tutte le dipendenze funzionali originali devono poter essere verificate esaminando una singola tabella nello schema decomposto.

In parole semplici: La proprietà di conservazione delle dipendenze garantisce che, dopo aver decomposto una tabella in più tabelle, ogni regola (dipendenza funzionale) della tabella originale possa essere verificata esaminando *una sola* delle tabelle risultanti, senza dover eseguire join.

Se F è l'insieme delle dipendenze funzionali su R, e R viene decomposto in $R_1, R_2, \ldots, R_n$, allora:

- Per ogni dipendenza $X \to Y$ in F, deve esistere almeno un R_i tale che $X \cup Y \subseteq R_i$
- Se nessun R_i contiene tutti gli attributi di una dipendenza, allora quella dipendenza non può essere verificata senza combinare più tabelle

Perché è importante? Senza la conservazione delle dipendenze:

- Diventa difficile mantenere l'integrità dei dati
- Le operazioni di controllo richiedono join costosi
- L'efficienza del database ne risente significativamente

Esempio di Decomposizione che **NON** preserva le dipendenze:

$R(\text{Employee}, \text{Project}, \text{Office})$ con dipendenze:

- Employee $\to$ Office
- Project $\to$ Office

Tabella originale R:

Employee	Project	Office
Jones	Mars	Rome
Smith	Jupiter	Milan
Smith	Venus	Milan
White	Saturn	Milan
White	Venus	Milan
White	Mars	Milan

Decomposizione in $R_1(\text{Employee, Office})$ e $R_2(\text{Employee, Project})$:

Employee	Office
Jones	Rome
Smith	Milan
White	Milan

Employee	Project
Jones	Mars
Smith	Jupiter
Smith	Venus
White	Saturn
White	Venus
White	Mars

La FD Project $\to$ Office **non è preservata** perché:

- In R_1 manca l'attributo Project, quindi non può verificare Project $\to$ Office
- In R_2 manca l'attributo Office, quindi non può verificare Project $\to$ Office
- Non esiste nessuna singola tabella in cui possiamo verificare questa dipendenza

Una decomposizione alternativa che preserva le dipendenze:

Per preservare tutte le dipendenze funzionali, possiamo decomporre R in tre tabelle:

- $R_1(\text{Employee, Office})$ - preserva Employee $\to$ Office
- $R_2(\text{Project, Office})$ - preserva Project $\to$ Office
- $R_3(\text{Employee, Project})$ - mantiene la relazione tra Employee e Project

Decomposizione che preserva le dipendenze:

Employee	Office
Jones	Rome
Smith	Milan
White	Milan

Project	Office
Mars	Rome
Jupiter	Milan
Venus	Milan
Saturn	Milan

Employee	Project
Jones	Mars
Smith	Jupiter
Smith	Venus
White	Saturn
White	Venus
White	Mars

Questa decomposizione preserva tutte le dipendenze, (perché ogni dipendenza funzionale $X \to Y$ è contenuta interamente in almeno una delle tabelle risultanti: Employee $\to$ Office è in R_1 e Project $\to$ Office è in R_2) ma purtroppo **non garantisce** la proprietà di join senza perdita (lossless join). Infatti, quando riuniamo le tre tabelle, potremmo generare tuple spurie (ad esempio, dal join otterremmo che White lavora a Mars con Office = Rome, mentre nella tabella originale White lavora a Mars ma con Office = Milan; oppure potremmo ottenere Smith lavora a Mars, che non esiste nella relazione originale).
Il trade-off tra "Dependency Preservation" e "Lossless Join" è uno dei motivi per cui è stata definita la Terza Forma Normale (3NF).

9.4 Recap: Quello che abbiamo imparato finora

Piccolo riassunto di quanto visto finora:

- **Normalizzazione:** È un processo per organizzare i dati in un database in modo da evitare ridondanze e anomalie. Serve a verificare e migliorare uno schema già progettato, non a crearlo da zero.
- **Anomalie:** Sono problemi che possono verificarsi quando lo schema del database non è ben progettato:
  - **Anomalia di inserimento:** Non posso inserire certi dati se non ho altri dati correlati.
  - **Anomalia di cancellazione:** Cancellando alcuni dati perdo accidentalmente altre informazioni importanti.
  - **Anomalia di aggiornamento:** Devo aggiornare lo stesso dato in più punti, rischiando inconsistenze.
- **Dipendenze Funzionali (FD):** Sono vincoli che esprimono come un attributo (o set di attributi) determina univocamente un altro attributo. Esempio: `Codice_Fiscale → Data_Nascita, Comune_Nascita` significa che conoscendo il CF posso determinare con certezza la data di nascita e il comune di nascita.
- **Forma Normale di Boyce-Codd (BCNF):** Una tabella è in BCNF se per ogni dipendenza funzionale X $\to$ Y, X deve essere una superchiave (cioè deve poter identificare univocamente ogni riga della tabella).
- **Decomposizione:** Quando una tabella non è in BCNF, la dividiamo in tabelle più piccole che siano in BCNF.
- **Proprietà della decomposizione:**
  - **Lossless Join:** La decomposizione deve permettere di ricostruire la tabella originale senza generare tuple spurie. Garantita se l'intersezione degli attributi forma una chiave per almeno una delle tabelle.
    - **Esempio:** Se decompongo `Studenti(Matricola, Nome, CorsoDiLaurea)` in `Anagrafica(Matricola, Nome)` e `Iscrizione(Matricola, CorsoDiLaurea)`, l'attributo comune `Matricola` è chiave per entrambe, quindi il join sarà senza perdita.
  - **Dependency Preservation:** Le dipendenze funzionali originali devono essere verificabili nelle tabelle decomposte senza fare join.
    - **Esempio:** Se ho la FD `Matricola → CorsoDiLaurea` nella tabella originale, dopo la decomposizione devo poterla verificare in una singola tabella, ovvero `Iscrizione`.
- **Il problema:** Non sempre è possibile ottenere una decomposizione che sia sia lossless che preservi tutte le dipendenze e sia in BCNF.

Questo è il motivo per cui ora introduciamo la Terza Forma Normale (3NF), che è un po' meno restrittiva della BCNF ma garantisce sempre una decomposizione che preserva le dipendenze e ha la proprietà di lossless join.

9.5 Terza Forma Normale (3NF)

La 3NF è una forma normale leggermente meno stringente della BCNF che permette sempre una decomposizione lossless e che preserva le dipendenze.

9.5.1 Definizione

Una relazione r è in **3NF** se, per ogni dipendenza funzionale non triviale $X \to Y$ definita su r, almeno una delle seguenti condizioni è vera:

1. X è una **superchiave** di r (condizione BCNF). **OPPURE**
2. Ogni attributo in Y è parte di **almeno una chiave candidata** di r (cioè, ogni attributo in Y è un "attributo primo").

9.5.2 BCNF vs 3NF

- BCNF è più restrittiva. Ogni relazione in BCNF è anche in 3NF.
- Una relazione in 3NF potrebbe non essere in BCNF.
- Si può sempre decomporre una relazione in 3NF in modo lossless e preservando le dipendenze.
- Se una relazione ha una sola chiave candidata, allora 3NF e BCNF sono equivalenti.

9.5.3 Esempio 3NF (ma non BCNF)

Consideriamo una tabella $R(\text{Chief, Project, Office})$. Supponiamo di avere le seguenti Dipendenze Funzionali (FDs):

1. $\{\text{Project, Office}\} \to \text{Chief}$ (questa è una chiave candidata)
2. $\text{Chief} \to \text{Office}$

Un'istanza di esempio della tabella R potrebbe essere:

Chief	Project	Office
Rossi	Alpha	Stanza101
Rossi	Beta	Stanza101
Verdi	Gamma	Stanza202
Bianchi	Alpha	Stanza303

Da questa tabella, osserviamo:

- Per la FD $\{\text{Project, Office}\} \to \text{Chief}$:
  - $(\text{Alpha, Stanza101}) \to \text{Rossi}$
  - $(\text{Beta, Stanza101}) \to \text{Rossi}$
  - $(\text{Gamma, Stanza202}) \to \text{Verdi}$
  - $(\text{Alpha, Stanza303}) \to \text{Bianchi}$

  La coppia $\{\text{Project, Office}\}$ identifica univocamente `Chief`, quindi è una chiave candidata.
- Per la FD $\text{Chief} \to \text{Office}$:
  - $\text{Rossi} \to \text{Stanza101}$

- Verdi → Stanza202
- Bianchi → Stanza303

L'attributo `Chief` determina univocamente `Office`.

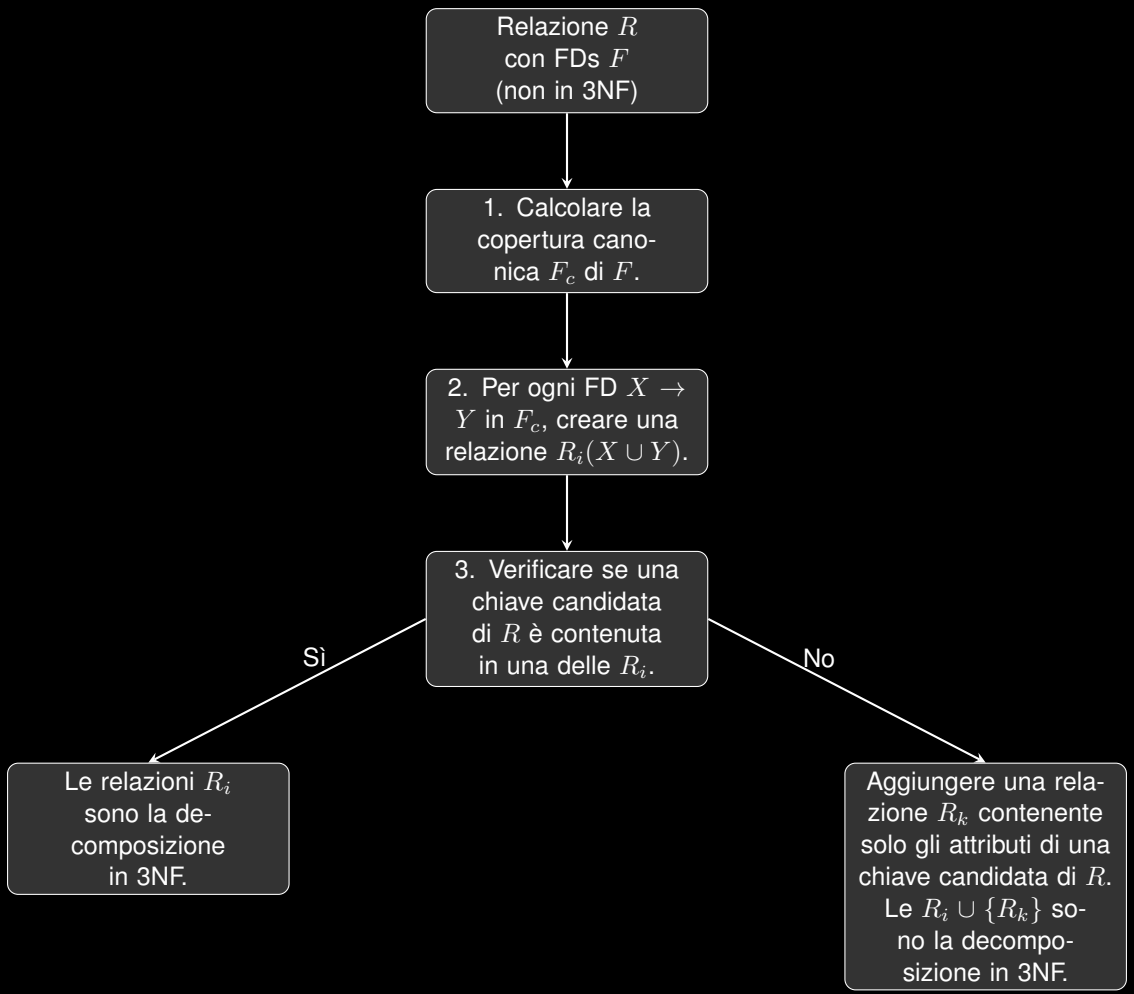
Analizziamo ora la FD Chief → Office rispetto alle forme normali:

- **BCNF:** La FD Chief → Office viola la BCNF perché Chief non è una superchiave. (Ad esempio, `Chief` da solo non determina `Project`: Rossi è associato sia al progetto Alpha che Beta). **Quindi, R NON è in BCNF.**
- **3NF:** Per la FD Chief → Office:
  1. Chief non è una superchiave. (Condizione 1 non soddisfatta)
  2. MA, ogni attributo in Y (che è $\{\text{Office}\}$ in questo caso) è parte di almeno una chiave candidata di R. L'attributo Office è infatti parte della chiave candidata $\{\text{Project, Office}\}$. (Condizione 2 soddisfatta)

  Poiché la seconda condizione è soddisfatta, **la relazione R È in 3NF.**

9.5.4 Algoritmo di Decomposizione in 3NF (Idea Generale)

1. Trova un insieme minimo di dipendenze funzionali (copertura canonica).
2. Per ogni FD $X \to Y$ in questa copertura, crea una tabella con attributi $X \cup Y$.
3. Se nessuna delle tabelle create contiene una chiave candidata della relazione originale, aggiungi un'ulteriore tabella contenente solo gli attributi di una chiave candidata originale.



9.5.5 Approccio Pratico Consigliato

1. Decomponi la relazione per raggiungere la 3NF.
2. Verifica se le tabelle risultanti sono anche in BCNF.
3. Se una tabella è in 3NF ma non in BCNF, valuta il trade-off.

9.5.6 Teoria delle Dipendenze e Implicazioni

Dato un insieme di dipendenze funzionali F, possiamo derivare altre dipendenze funzionali. Diciamo che F implica f se ogni relazione che soddisfa F soddisfa anche f. L'insieme di tutte le dipendenze implicite da F è chiamato **chiusura di F (F^+)**.

Assiomi di Armstrong

1. **Riflessività:** Se $Y \subseteq X$, allora $X \to Y$.
2. **Aumento:** Se $X \to Y$, allora $XZ \to YZ$.
3. **Transitività:** Se $X \to Y$ e $Y \to Z$, allora $X \to Z$.
    - Esempio: MatricolaStudente $\to$ CodiceCorsoLaurea e CodiceCorsoLaurea $\to$ NomeCorsoLaurea. Allora, MatricolaStudente $\to$ NomeCorsoLaurea.

Chiusura di un insieme di attributi X^+

L'insieme di tutti gli attributi che sono funzionalmente determinati da X, dato un insieme di FDs F.

Copertura Minima (o Canonica)

Un insieme "minimale" di FDs equivalente a F, senza ridondanze.

9.6 Normalizzazione nel Design Concettuale (Modello E-R)

La teoria della normalizzazione può essere usata anche per verificare la qualità di un modello Entità-Relazione.

9.6.1 Esempio: Normalizzazione su Entità

Considera un'entità `Prodotto` con attributi: `Codice` (PK), `NomeProdotto`, `Prezzo`, `PartitaIVAFornitore`, `NomeFornitore`, `IndirizzoFornitore`.

Identifichiamo una FD: PartitaIVAFornitore $\to$ NomeFornitore, IndirizzoFornitore. Qui, `PartitaIVAFornitore` non è la chiave di `Prodotto`. Questo viola le forme normali.

Decomposizione dell'Entità:

- Entità `Prodotto`(`Codice`, `NomeProdotto`, `Prezzo`)
- Entità `Fornitore`(`PartitaIVAFornitore`, `NomeFornitore`, `IndirizzoFornitore`)
- Relazione `Fornisce` tra `Fornitore` e `Prodotto`.

Nello schema proposto dalla slide 54:

- `Product(Code, Name, Price)`
- `Supplier(VATNum, Name, Address)`
- `Supply` (relationship)

9.6.2 Esempio: Normalizzazione su Relazioni (Relationship)

Considera una relazione `Tesi` che collega `Studente`, `Professore`, `DipartimentoProf`, `CorsoLaureaStudente`. Assumiamo che (MatricolaStudente, IDProfessore) sia la chiave. FDs:

- MatricolaStudente $\rightarrow$ CorsoLaureaStudente
- IDProfessore $\rightarrow$ DipartimentoProf

Nella tabella `Tesi`(MatricolaStudente, IDProfessore, CorsoLaureaStudente, DipartimentoProf):

- IDProfessore $\rightarrow$ DipartimentoProf: `IDProfessore` è parte della chiave, non la chiave intera. `DipartimentoProf` non è un attributo primo. Viola 3NF/BCNF.

Decomposizione della Relazione (Conceptual Level):

1. Creare un'entità `Professore` con attributo `Dipartimento` (slide 50: `Professor -(1,1)- Work -(0,N)- Dept`).
2. Creare un'entità `Studente` con attributo `CorsoLaurea` (slide 52: `Student -(1,1)- Enroll -(0,N)- Degree`).
3. La relazione `Tesi` ora collegherebbe solo `Studente` e `Professore` (slide 52: `Professor -(0,N)- Thesis -(0,1)- Student`).

Questo processo porta a un modello concettuale più robusto.

Capitolo 10

Database Attivi

10.1 Dai Database Passivi ai Database Attivi

L'idea di base dei database attivi è quella di rendere il database stesso più "intelligente" e "reattivo", capace cioè di eseguire automaticamente delle azioni in risposta a determinati eventi, senza che sia l'applicazione a dover gestire tutta questa logica.

10.1.1 Database Passivi (Tradizionali)

- Eseguono solo le operazioni esplicitamente richieste dall'utente o dall'applicazione.
- Un primo, rudimentale esempio di "reattività" nei database passivi sono le **strategie di reazione ai vincoli di integrità referenziale**.
  - Esempio SQL: `ON DELETE CASCADE`, `ON UPDATE SET NULL`, `ON DELETE SET DEFAULT`, `ON DELETE NO ACTION`.
  - Qui il database "reagisce" a un `DELETE` o `UPDATE` su una tabella primaria, eseguendo un'azione sulla tabella correlata.
- L'idea è di estendere questa capacità introducendo costrutti linguistici specifici (chiamati **regole attive**) per gestire una parte del comportamento procedurale che altrimenti sarebbe nell'applicazione.
- **Vantaggio:** Se questo comportamento è a livello di database, è "condiviso" tra tutte le applicazioni che accedono a quei dati, garantendo consistenza e promuovendo l'indipendenza dei dati.

10.1.2 Database Attivi

- Hanno un componente dedicato per gestire **regole attive** basate sul paradigma **ECA (Event-Condition-Action)**.
  - **Evento (Event):** Un cambiamento nel database (es. `INSERT`, `UPDATE`, `DELETE` su una tabella specifica).
  - **Condizione (Condition):** Una verifica (un predicato SQL) che deve essere vera affinché l'azione scatti. Se la condizione è omessa, si assume sempre vera.
  - **Azione (Action):** Una o più istruzioni SQL (o codice in un linguaggio procedurale specifico del DBMS, come PL/SQL per Oracle) da eseguire.
- Questi database hanno un **comportamento reattivo**: non si limitano a eseguire le transazioni dell'utente, ma *reagiscono* agli eventi eseguendo anche le regole definite.
- Nei DBMS commerciali (standard SQL3 e successivi), le regole attive sono implementate principalmente tramite i **trigger**.

10.2 Evoluzione dell'Architettura e Ruolo dei Database Attivi

Le slide mostrano un'evoluzione nel tempo di come la logica applicativa e la gestione dei dati sono state organizzate:

- **Anni '70 (No DBMS):** Le applicazioni accedevano direttamente ai file tramite il Sistema Operativo.
- **Anni '80 (Primi DBMS):** Le applicazioni interagivano con un DBMS, che gestiva "tabelle di dati".
- **Anni '90 (Comportamento Procedurale):** Esigenza di spostare parte del **comportamento procedurale condiviso** all'interno del DBMS.
  - **Stored Procedures:** Introdotte per condividere logica comune. Problemi: non standardizzate, impedance mismatch.
  - **Trigger (Database Attivi):** Introdotte regole specifiche (i **trigger**) per modellare il comportamento procedurale condiviso, gestito direttamente dal DBMS.
- **Anni 2000 (Applicazioni Web):** Architettura client-server a più livelli (Client JS, Web App Server Java/Node, Server con Active DBMS).
- **Anni 2010 (Mobile Apps):** Simile, con client mobile.

Concetto chiave dell'evoluzione: Tendenza a spostare la logica strettamente legata ai dati e condivisa all'interno del database stesso.

10.3 Trigger: Il Cuore dei Database Attivi

Un trigger è una procedura memorizzata nel database che viene eseguita automaticamente quando si verifica un determinato evento su una tabella specifica.

10.3.1 Definizione

- Definiti con istruzioni DDL (Data Definition Language), es. `CREATE TRIGGER`.
- Seguono il paradigma **ECA**:
  - **Evento:** Un'operazione di modifica dei dati (`INSERT`, `DELETE`, `UPDATE`).
  - **Condizione:** Un predicato SQL opzionale (clausola `WHEN`).
  - **Azione:** Una sequenza di istruzioni SQL o un blocco di codice procedurale.
- **Flusso intuitivo:** Attivazione (evento) $\rightarrow$ Verifica (condizione) $\rightarrow$ Esecuzione (azione).
- Ogni trigger è associato a una **tabella target**.

10.3.2 Granularità dei Trigger

- **Row-level (per tupla/riga):** Il trigger si attiva e la sua azione viene eseguita *per ogni singola riga* affetta dall'istruzione SQL.
- **Statement-level (per istruzione):** Il trigger si attiva e la sua azione viene eseguita *una sola volta per l'intera istruzione SQL*.

10.3.3 Modalità (Timing) dei Trigger

- `IMMEDIATE` **(Immediata):** L'azione del trigger viene eseguita immediatamente *prima* (`BEFORE`) o *dopo* (`AFTER`) l'evento. Modalità più comune.
- `DEFERRED` **(Differita):** L'azione del trigger viene posticipata e eseguita solo al momento del `COMMIT` della transazione.

10.3.4 Modello Computazionale e Problemi

Data una transazione utente $T^U = U_1; \ldots; U_n$. Se le regole P sono del tipo E, C $\rightarrow$ A. U_i^P è la sequenza di azioni indotte da U_i.

- **Semantica Immediata:** $T^I = U_1; U_1^P; U_2; U_2^P; \ldots; U_n; U_n^P$.
- **Semantica Differita:** $T^D = U_1; \ldots U_n; U_1^P; \ldots; U_n^P$.
- **Problemi Potenziali:**
  - **Terminazione:** L'esecuzione a cascata dei trigger deve terminare (evitare cicli infiniti).
  - **Confluenza:** Se più trigger possono essere attivati, il risultato finale è lo stesso indipendentemente dall'ordine?
  - **Equivalenza:** Diverse definizioni di regole portano allo stesso comportamento?

10.4 Sintassi dei Trigger (SQL:1999 Standard)

```
CREATE TRIGGER nomeTrigger
{ BEFORE | AFTER } -- Timing
{ INSERT | DELETE | UPDATE [OF nomeColonna [, nomeColonna]...] } -- Evento
ON nomeTabellaTarget -- Tabella target

[ REFERENCING -- Variabili per righe/tabelle vecchie e nuove
-- Per trigger STATEMENT-LEVEL:
[ OLD TABLE [AS] varTabellaVecchia ]
[ NEW TABLE [AS] varTabellaNuova ]
-- Per trigger ROW-LEVEL:
[ OLD [ROW] [AS] varTuplaVecchia ] -- Solitamente OLD
[ NEW [ROW] [AS] varTuplaNuova ]   -- Solitamente NEW
]

[ FOR EACH { ROW | STATEMENT } ] -- Granularità

[ WHEN (condizioneSQL) ] -- Condizione (opzionale)

SQLProceduralStatement; -- Azione
```

10.4.1 `BEFORE` vs `AFTER`

- `BEFORE`: Eseguito *prima* dell'operazione. Utile per validare/modificare dati in ingresso.
- `AFTER`: Eseguito *dopo* l'operazione. Utile per logging, aggiornare tabelle dipendenti.

10.4.2 Clausola `REFERENCING` (`OLD` e `NEW`)

Permette di accedere ai valori dei dati *prima* e *dopo* la modifica.

- **Per trigger `ROW-LEVEL`:**
  - `OLD`: Pseudo-riga con valori *prima* della modifica (per `UPDATE`, `DELETE`). Accesso: `OLD.nomeColonna`.
  - `NEW`: Pseudo-riga con valori *dopo* la modifica (per `INSERT`) o proposti (per `UPDATE`). Accesso: `NEW.nomeColonna`.
- **Per trigger `STATEMENT-LEVEL`:**
  - `OLD TABLE`: Tabella temporanea con righe *prima* della modifica.
  - `NEW TABLE`: Tabella temporanea con righe *dopo* la modifica.
- **Disponibilità:**
  - `INSERT`: Solo `NEW` / `NEW TABLE`.
  - `DELETE`: Solo `OLD` / `OLD TABLE`.
  - `UPDATE`: Sia `OLD` / `OLD TABLE` che `NEW` / `NEW TABLE`.

10.5 Trigger in Oracle

10.5.1 Sintassi

```
CREATE [OR REPLACE] TRIGGER nomeTrigger
{ BEFORE | AFTER }
evento1 [OR evento2 OR evento3 ...] -- Es. INSERT OR UPDATE OF col1
ON nomeTabella
[ REFERENCING OLD AS nomeVarVecchia NEW AS nomeVarNuova ] -- Default :OLD, :NEW
[ FOR EACH ROW ] -- Se omesso, è STATEMENT level
[ WHEN (condizione) ]
DECLARE
-- variabili locali PL/SQL
BEGIN
-- corpo del trigger (logica PL/SQL)
-- accesso con :OLD.colonna e :NEW.colonna per FOR EACH ROW
EXCEPTION
-- gestione errori
END;
```

10.5.2 Semantica Oracle

- Modalità Immediata (`BEFORE`, `AFTER`).
- Ordine di Esecuzione:
  1. `BEFORE STATEMENT` triggers.
  2. Per ogni riga affetta:
     (a) `BEFORE ROW` triggers.
     (b) Operazione DML + controllo vincoli.
     (c) `AFTER ROW` triggers.
  3. `AFTER STATEMENT` triggers.
- Errore: Rollback dell'intera istruzione/transazione.
- Priorità: Basata su timestamp di creazione (non garantita).
- Cascata: Massimo 32 trigger.

10.5.3 Esempio Oracle (Riordino Prodotti)

Trigger `Reorder` su tabella `Warehouse`.

- **Evento:** `AFTER UPDATE OF QtyAvbl ON Warehouse`.
- **Granularità:** `FOR EACH ROW`.
- **Condizione:** `WHEN (NEW.QtyAvbl < NEW.QtyLimit)`.
- **Azione (PL/SQL):**

```
DECLARE
X NUMBER;
BEGIN
-- Controlla se esiste già un ordine pendente per questa parte
SELECT COUNT(*) INTO X
FROM PendingOrders
WHERE Part = :NEW.Part; -- :NEW si riferisce alla riga aggiornata

IF X = 0 THEN -- Se non ci sono ordini pendenti
-- Inserisce un nuovo ordine pendente
```

```
INSERT INTO PendingOrders (Part_ID, QuantityToReorder, OrderDate)
VALUES (:NEW.Part, :NEW.QtyReord, SYSDATE);
END IF;
END;
```

10.6 Trigger in DB2

10.6.1 Sintassi

```
CREATE TRIGGER nomeTrigger
{ BEFORE | AFTER } evento -- evento è INSERT, UPDATE, DELETE
ON nomeTabella
REFERENCING { OLD AS varTuplaVecchia | NEW AS varTuplaNuova |
OLD_TABLE AS varTabellaVecchia | NEW_TABLE AS varTabellaNuova } ...
FOR EACH { ROW | STATEMENT }
[ WHEN (predicatoSQL) ]
SQLProceduralStatement; -- Può essere un blocco BEGIN ATOMIC ... END
```

10.6.2 Semantica DB2

- Modalità Immediata.
- I trigger `BEFORE` di norma *non possono modificare il database* (eccetto assegnare valori a `NEW` in `BEFORE INSERT/UPDATE ROW`), quindi non possono attivare altri trigger.
- Errore: Rollback.
- Priorità: Determinata dal sistema (timestamp).
- Cascata: Massimo 16 trigger.

10.6.3 Esempio DB2 (Controllo Riduzione Stipendio)

Trigger `checkWage` su tabella `Employee`.

- **Evento:** `AFTER UPDATE OF Wage ON Employee`.
- **Granularità:** `FOR EACH ROW`.
- **Condizione:** `WHEN (NEW.Wage < OLD.Wage * 0.97)`.
- **Azione:**

```
-- La sintassi specifica può variare leggermente in DB2 SQL PL
-- Questo è un esempio concettuale basato sulle slide
BEGIN
-- Se la riduzione è maggiore del 3%, la limita al 3%
-- L'azione qui presuppone che il trigger AFTER possa modificare la stessa riga
-- anche se più tipicamente si impedirebbe l'azione in un BEFORE trigger
-- o si farebbe l'update in modo più controllato.
-- La slide suggerisce un update, quindi lo riporto così:
UPDATE Employee
SET Wage = OLD.Wage * 0.97
WHERE EmpCode = NEW.EmpCode; -- O l'identificativo di riga corrente
END
```

Nota sull'esempio DB2: L'azione di un trigger `AFTER` che modifica la stessa riga che ha scatenato il trigger può portare a ricorsione se non gestita con attenzione. Spesso, per questo tipo di logica, si preferirebbe un trigger `BEFORE` per modificare `NEW.Wage` o per sollevare un errore se la condizione non è rispettata.

10.7 Estensioni dei Trigger (Non Sempre Disponibili)

- Eventi Temporali (periodici) o definiti dall'utente.
- Combinazioni Booleane di Eventi.
- Clausola `INSTEAD OF`: Esegue l'azione del trigger al posto dell'operazione originale (utile per viste non aggiornabili).
- Esecuzione "Detached" (transazione autonoma).
- Definizione di Priorità esplicita.
- Gruppi di Regole (attivabili/disattivabili).
- Regole su Query (`SELECT`).

10.8 Proprietà delle Regole Attive

- **Terminazione (essenziale):** L'esecuzione deve finire.
- **Confluenza:** Il risultato finale è indipendente dall'ordine di esecuzione di trigger concorrenti.
- **Determinismo delle Osservazioni:** L'utente osserva sempre lo stesso comportamento.

10.9 Applicazioni dei Trigger

10.9.1 Funzionalità Interne al DBMS

- **Gestione dei Vincoli di Integrità Complessi:** Oltre ai vincoli standard.
- **Replicazione dei Dati:** Catturare modifiche e replicarle.
- **Gestione delle Viste:**
  - **Viste Materializzate:** Propagare modifiche dalle tabelle base alla vista materializzata.
  - **Viste Virtuali (con `INSTEAD OF`):** Rendere aggiornabili viste complesse.

10.9.2 Funzionalità Applicative (Logica di Business nel DB)

- **Descrizione del Comportamento del Database:** Incapsulare logica di business direttamente nel DB per consistenza.
  - Esempi: Mantenere `last_modified_date`, inviare notifiche, audit, calcolare valori derivati, impedire operazioni basate su condizioni complesse.
- **Confronto Logica in Applicazione vs. Logica in Trigger:**
  - **Logica in Applicazione (es. Node.js con Prisma/Mongoose):**
    - *Pro:* Più facile da testare, linguaggio dell'applicazione, flessibilità.
    - *Contro:* Se il DB è accessibile esternamente, la logica può essere bypassata.
  - **Logica in Trigger DB:**
    - *Pro:* Consistenza garantita, logica vicina ai dati.
    - *Contro:* Minore visibilità/debug per lo sviluppatore applicativo, dipendenza dal linguaggio procedurale del DB, test più complessi.

10.10 Conclusione

I database attivi, attraverso i trigger, offrono un meccanismo potente per automatizzare reazioni a eventi sui dati, centralizzare la logica di business e garantire la consistenza. Tuttavia, il loro uso richiede un'attenta progettazione per evitare complessità, problemi di performance e difficoltà di manutenzione. È fondamentale bilanciare cosa implementare a livello di database tramite trigger e cosa lasciare alla logica applicativa.

Capitolo 11

Laboratorio 1: Algebra Relazionale

11.1 Introduzione all'Algebra Relazionale

L'algebra relazionale è un linguaggio procedurale formale per interrogare i database relazionali. Ogni operazione prende una o più relazioni (tabelle) come input e produce una nuova relazione come output. Questo permette di annidare le operazioni.

11.2 Operatori Fondamentali dell'Algebra Relazionale

11.2.1 Selezione (σ - Sigma)

- **Scopo:** Filtra le tuple (righe) di una relazione che soddisfano una certa condizione (predicato).
- **Notazione:** $\sigma_{\text{condizione}}(\text{Relazione})$
- **Esempio (Slide 4):** Trovare le città con più di 1.000.000 di abitanti.

$$\sigma_{\text{Population}>1000000}(\text{CITY})$$

- **Equivalente SQL (concettuale):**

```sql
SELECT *
FROM CITY
WHERE Population > 1000000;
```

- **Spiegazione:** La condizione può coinvolgere confronti tra attributi e costanti (es. Age > 30, Name $=$ 'Mario'), o tra attributi (es. Price $>$ Cost $* 1.1$). Le condizioni possono essere combinate con operatori logici AND ($\wedge$), OR ($\vee$), NOT ($\neg$).

11.2.2 Proiezione (π - Pi)

- **Scopo:** Seleziona determinati attributi (colonne) da una relazione, eliminando le tuple duplicate nella relazione risultante.
- **Notazione:** $\pi_{\text{attributo1, attributo2, ...}}(\text{Relazione})$
- **Esempio (Slide 6):** Ottenere solo i nomi delle regioni.

$$\pi_{\text{Region}}(\text{RisultatoJoinPrecedente})$$

- **Equivalente SQL (concettuale):**

```sql
SELECT DISTINCT Region
FROM RisultatoJoinPrecedente;
```

- **Spiegazione:** È importante ricordare che la proiezione elimina automaticamente i duplicati.

11.2.3 Ridenominazione (ρ - Rho)

- **Scopo:** Cambia il nome di una relazione e/o dei suoi attributi. È cruciale per:
  - Confrontare una relazione con se stessa (self-join).
  - Rendere compatibili per nome gli attributi per un natural join.
  - Evitare ambiguità quando si combinano relazioni con attributi omonimi ma con significati diversi.
- **Notazione:**
  - Ridenominare la relazione: $\rho_{\text{NuovoNomeRelazione}}(\text{RelazioneOriginale})$
  - Ridenominare attributi: $\rho_{\text{nuovoNomeAttr1} \leftarrow \text{vecchioNomeAttr1}, \dots}(\text{Relazione})$
  - Entrambi: $\rho_{\text{NuovoNomeRelazione}(\text{nuovoNomeAttr1} \leftarrow \text{vecchioNomeAttr1}, \dots)}(\text{RelazioneOriginale})$
- **Esempio (Slide 5):** Ridenominare l'attributo `City` in `BELONGING` a `Code`.

$$\rho_{\text{Code} \leftarrow \text{City}}(\text{BELONGING})$$

- **Esempio (Slide 9):** Per un self-join su `TRAIN`.

$$T1 := \rho_{\text{Code1, Switch, Miles1} \leftarrow \text{Code, End, Miles}}(\text{TRAIN})$$

$$T2 := \rho_{\text{Code2, Switch, Miles2} \leftarrow \text{Code, Start, Miles}}(\text{TRAIN})$$

- **Equivalente SQL (concettuale):**
  - Alias di colonna:

```sql
SELECT City AS Code FROM BELONGING;
```

  - Alias di tabella:

```sql
SELECT T1.Code, T2.Code
FROM TRAIN AS T1
JOIN TRAIN AS T2 ON T1.End = T2.Start;
```

11.2.4 Join (Vari Tipi)

Natural Join ($\bowtie$)

- **Scopo:** Combina tuple da due relazioni che hanno valori uguali su tutti gli attributi con lo stesso nome. Gli attributi comuni appaiono una sola volta nel risultato.
- **Notazione:** Relazione1 $\bowtie$ Relazione2
- **Esempio (Slide 5, dopo ridenominazione):**

$$\text{BELONGING\_CON\_CODE} \bowtie (\sigma_{\text{Population} > 1000000}(\text{CITY}))$$

  Il join avviene sull'attributo comune `Code`.
- **Equivalente SQL (concettuale):**

```sql
-- Dopo ridenominazione mentale o esplicita di BELONGING.City a Code
SELECT Region, B.Code, Name, Population
FROM (SELECT Region, City AS Code FROM BELONGING) AS B
NATURAL JOIN (SELECT * FROM CITY WHERE Population > 1000000) AS C;

-- Oppure, più comunemente con Theta Join:
SELECT ...
FROM BELONGING B JOIN CITY C ON B.City = C.Code
WHERE C.Population > 1000000;
```

- **Spiegazione:** Se non ci sono attributi comuni, il natural join diventa un prodotto cartesiano.

Theta Join ($\bowtie_{\text{condizione}}$)

- **Scopo:** Combina tuple da due relazioni basandosi su una condizione specificata (θ). Include l'equi-join se la condizione usa solo l'uguaglianza.
- **Notazione:** Relazione1 $\bowtie_{\text{condizione}}$ Relazione2
- **Esempio (Slide 25):** Unire `STUDENT` ed `EXAM`.

$$\text{STUDENT} \bowtie_{\text{Id=Student}} (\sigma_{\text{Mark}=30}(\text{EXAM}))$$

- **Equivalente SQL (concettuale):**

```sql
SELECT *
FROM STUDENT S JOIN (SELECT * FROM EXAM WHERE Mark=30) E
ON S.Id = E.Student;
```

Prodotto Cartesiano ($\times$)

- **Scopo:** Produce una relazione che contiene tutte le possibili combinazioni di tuple dalle due relazioni di input.
- **Notazione:** Relazione1 $\times$ Relazione2
- **Spiegazione:** Se Relazione1 ha n tuple e m attributi, e Relazione2 ha p tuple e q attributi, il risultato avrà $n \times p$ tuple e $m + q$ attributi. Di solito seguito da una selezione per implementare un join: $\sigma_{\text{condizione}}(\text{Relazione1} \times \text{Relazione2})$ è equivalente a Relazione1 $\bowtie_{\text{condizione}}$ Relazione2.

11.2.5 Operatori Insiemistici

Questi operatori richiedono che le relazioni siano *union-compatibili*, cioè abbiano lo stesso numero di attributi e i domini (tipi) degli attributi corrispondenti siano compatibili.

Unione ($\cup$)

- **Scopo:** Produce una relazione contenente tutte le tuple che appaiono in Relazione1 o in Relazione2 (o in entrambe), eliminando i duplicati.
- **Notazione:** Relazione1 $\cup$ Relazione2
- **Esempio (Slide 55):** Pizzerie frequentate solo da femmine O solo da maschi.

$$\text{PizzerieSoloFemmine} \cup \text{PizzerieSoloMaschi}$$

- **Equivalente SQL (concettuale):**

```sql
SELECT ... FROM Relazione1
UNION
SELECT ... FROM Relazione2;
```

Differenza ($-$)

- **Scopo:** Produce una relazione contenente tutte le tuple che sono in Relazione1 ma non in Relazione2.
- **Notazione:** Relazione1 $-$ Relazione2
- **Esempio (Slide 31):** Studenti che non hanno MAI preso 30.

$$\text{TuttiStudenti}(\text{Nome,Cognome,Id}) - \text{StudentiCon30}(\text{Nome,Cognome,Id})$$

- **Equivalente SQL (concettuale):**

```sql
SELECT ... FROM Relazione1
EXCEPT -- (o MINUS in Oracle)
SELECT ... FROM Relazione2;
```

Intersezione ($\cap$)

- **Scopo:** Produce una relazione contenente tutte le tuple che appaiono sia in Relazione1 sia in Relazione2.
- **Notazione:** Relazione1 $\cap$ Relazione2
- **Esempio (Slide 51):** Negozi di vino che hanno sia rosé SIA vini rossi.

$$\text{NegoziConRosé}(\text{CodiceNegozio}) \cap \text{NegoziConViniRossi}(\text{CodiceNegozio})$$

- **Equivalente SQL (concettuale):**

```sql
SELECT ... FROM Relazione1
INTERSECT
SELECT ... FROM Relazione2;
```

- **Nota:** L'intersezione può essere espressa usando la differenza: $A \cap B = A - (A - B)$.

11.3 Concetti e Pattern Comuni Illustrati negli Esercizi

11.3.1 Self-Join (Slide 8, 29, 61)

Quando una query richiede di confrontare tuple all'interno della stessa tabella. Richiede quasi sempre la ridenominazione (ρ) di almeno una "copia" della tabella per distinguere gli attributi. Esempio: Trovare treni con un interscambio, fornitori che forniscono almeno due prodotti diversi.

11.3.2 Query "Solo" / "Tutti" (Quantificazione Universale)

- **"Solo X":** Per trovare entità che sono associate *esclusivamente* a X. Pattern: $(\text{Entità associate a X}) - (\text{Entità associate a NON X})$. Esempio (Slide 35): Utenti che hanno preso in prestito *solo* libri di Fleming.

$$\text{UtentiConLibriFleming} - \text{UtentiConLibriNonFleming}$$

- **"Tutti Y che soddisfano X":** Spesso si traduce in "non esiste nessun Y che NON soddisfa X". Esempio (Slide 57): Riviste che *non hanno mai* pubblicato articoli di motociclismo.

$$\text{TutteLeRiviste} - \text{RivisteCheHannoPubblicatoMotociclismo}$$

11.3.3 Condizioni Complesse e Ordine delle Operazioni (Slide 12, 23)

La selezione $\sigma_{\text{Price}>\text{Age}*10}$ (Slide 23) può essere applicata solo *dopo* aver unito le tabelle necessarie. L'ordine è importante. "Spingere" le selezioni e le proiezioni il più presto possibile è una tecnica di ottimizzazione. L'esempio della "extended edition" (Slide 12) mostra come selezionare dopo il join o, se possibile e più efficiente, selezionare prima del join.

11.3.4 Trovare Minimi/Massimi (Slide 38-45)

L'algebra relazionale non ha operatori aggregati diretti come `MIN()/MAX()` di SQL. Per trovare il minimo (es. la prima data per uno studente), si usa un pattern che coinvolge la differenza insiemistica:

1. Si selezionano tutte le coppie <studente, data> di interesse ($R1$).
2. Si crea una copia $R2$ di $R1$ rinominando l'attributo data (es. $Date1$).
3. Si trovano le tuple (s, d) in $R1$ per le quali esiste una tupla (s, d') in $R2$ tale che $d > d'$. Queste sono le date che *non* sono minime per quello studente. Formalmente: $\pi_{\text{Student,Date}}(\sigma_{\text{Date}>\text{Date1}}(R1 \bowtie \rho_{\text{Student,Date1}\leftarrow\text{Student,Date}}(R1)))$. Le slide usano un approccio leggermente diverso (Slide 44): si considerano tutte le date possibili da $R2$ e si sottraggono quelle che sono "successive" a una data in $R1$ (dopo un join e selezione $Date < Date1$). $R_{\text{minime}} := R2 - \pi_{\text{Student,Date1}}(\sigma_{\text{Date}<\text{Date1}}(R1 \bowtie R2))$ (assumendo $R1$ ha $Date$ e $R2$ ha $Date1$, e $R1, R2$ sono create da $\pi_{\text{Student,Date}}(\sigma_{...}(\text{EXAM}))$).
4. L'idea generale è isolare le tuple che non hanno "predecessori" secondo il criterio di ordinamento.

La sequenza completa (slide 45) per la prima data è: $R1 := \pi_{\text{Student,Date}}(\sigma_{\text{Mark}=30}(\text{EXAM}))$ $R2 := \rho_{\text{Date1}\leftarrow\text{Date}}(R1)$ $R_{\text{NonPrimeDate}} := \pi_{\text{Student,Date1}}(\sigma_{\text{Date}<\text{Date1}}(R1 \bowtie R2))$ $R_{\text{PrimeDatePerStudente}} := R2 - R_{\text{NonPrimeDate}}$ Poi si rinomina $Date1$ in $Date$ per il join finale con STUDENT: $R3 := \rho_{\text{Date}\leftarrow\text{Date1}}(R_{\text{PrimeDatePerStudente}})$ Risultato: $\pi_{\text{Name,Surname,Date}}(\text{STUDENT} \bowtie_{\text{Id}=\text{Student}} R3)$

11.4 Appendix: Notazione per Esame Remoto (Slide 62-64)

Notazione testuale alternativa per gli operatori dell'algebra relazionale.

- **Rename:** `RID[(<new A1>,<old A1>),...]R`
- **Select:** `SEL[C]R`
- **Projection:** `PRO[A1, A2, ...]R`
- **Join:**
  - Natural Join: `R1 JOIN R2`
  - Theta Join: `R1 JOIN[condizione] R2`
- **Set Operations:**
  - Union: `R1 UNI R2`
  - Intersection: `R1 INT R2`
  - Difference: `R1 DIF R2`
- **Condizioni Booleane:** `NOT C`, `C1 OR C2`, `C1 AND C2`.

Esempio Combinato (dalla Slide 64):

- **Query Standard (parte della differenza):**

$$(\pi_{\text{idModello,nome}}(\sigma_{\text{categoria='Berlina'}\wedge\text{nazionalità='Francia'}}(\text{Modello} \bowtie \rho_{\text{nomeM}\leftarrow\text{nome}}\text{Marca})))$$
$$-(\pi_{\text{idModello,nome}}(\sigma_{\text{anno='2007'}}(\text{Modello} \bowtie \text{Valutazione})))$$

- **Query in Notazione Remota (per la parte della differenza):**

```
(PRO[idModello,nome](
    SEL[categoria='Berlina' AND nazionalita='Francia'](
        Modello JOIN RID[(nomeM,nome)]Marca
    )
))
DIF
(PRO[idModello,nome](
    SEL[anno='2007'](
        Modello JOIN Valutazione
    )
))
```

E l'eventuale join finale:

```
... JOIN[idModello=modello] Valutazione
```

Capitolo 12

Laboratorio 2: SQL

12.1 Approccio alla Scrittura di Query SQL

Esistono due stili principali per affrontare la scrittura di query SQL:

- **Approccio "Teorico"**:
  - Prima si pensa in termini di **algebra relazionale** (le operazioni matematiche formali sui set di dati come selezione, proiezione, join, unione, intersezione, differenza).
  - Poi si traduce l'espressione dell'algebra relazionale in una query SQL.
  - Questo approccio è utile per comprendere profondamente cosa fa la query e per ottimizzarla logicamente.
- **Approccio "Geek" (o Pratico)**:
  - Si impara SQL direttamente e, attraverso la pratica, si sviluppa un'intuizione che permette poi di comprendere o formulare espressioni di algebra relazionale.
  - Dato che conosci già SQL, probabilmente ti ritrovi più in questo. Tuttavia, avere una base di algebra relazionale può aiutare a scrivere query più complesse e corrette.

12.2 Come Risolvere gli Esercizi (Passi Preliminari - Step 0)

Prima ancora di leggere il testo dell'esercizio, è fondamentale analizzare lo schema del database:

12.2.1 Comprensione dello Schema del Database

- **Identificare Entità e Relazioni**: Quali sono le tabelle principali (entità come `STUDENTE`, `CORSO`) e come sono collegate tra loro (relazioni, spesso implementate con tabelle di raccordo come `ESAME` o `EDIZIONE`)?
- **Esempio Schema**:
  - `LECTURER(Id, Surname, Dept)`
  - `STUDENT(Id, Surname)`
  - `COURSE(Code, Name)`
  - `EDITION(Course, Year, Lecturer)` (Collega un corso e un anno a un docente)
  - `EXAM(Student, Course, Year)` (Collega uno studente a un'edizione di un corso)

12.2.2 Identificazione di Chiavi Primarie (PK) e Chiavi Esterne (FK)

- **Chiavi Primarie (PK)**:
  - Attributi (o un insieme di attributi) che identificano univocamente ogni record (riga) all'interno di una tabella.
  - Nelle slide, sono **sottolineate** (es. `LECTURER(Id)`).
- **Chiavi Esterne (FK)**:
  - Attributi (o un insieme di attributi) in una tabella che fanno riferimento alla chiave primaria di un'altra tabella.
  - Stabiliscono e applicano un legame tra i dati di due tabelle.
  - **Esempio Pratico**: In `EDITION(Course, Year, Lecturer)`:
    - `Course` è una FK che referenzia `COURSE(Code)`.
    - `Lecturer` è una FK che referenzia `LECTURER(Id)`.
  - **Esempio Pratico**: In `EXAM(Student, Course, Year)`:
    - `Student` è una FK che referenzia `STUDENT(Id)`.
    - `(Course, Year)` insieme potrebbero referenziare una PK composita in `EDITION` (se `(Course, Year)` fosse la PK di `EDITION`) oppure `Course` referenzia `COURSE(Code)` e `Year` è un attributo contestuale. Dallo schema `EDITION`, sembra che `Course` e `Year` siano parte della PK di `EDITION`, quindi `EXAM` si lega a una specifica edizione di un corso.

12.3 Come Risolvere gli Esercizi (Costruzione della Query - Step 1)

Una volta compreso lo schema e il problema, si passa alla costruzione della query:

12.3.1 Quali relazioni (tabelle) sono coinvolte?

- Identifica le tabelle necessarie per ottenere i dati richiesti e per applicare i filtri.
- Queste andranno nella clausola `FROM`.
- **Esempio**: Se devi trovare i nomi degli studenti che hanno superato l'esame del corso "Databases", ti serviranno almeno `STUDENT` (per il nome) ed `EXAM` (per l'informazione sull'esame e il collegamento allo studente), e forse `COURSE` (per filtrare per nome del corso).

12.3.2 Quali attributi (colonne) devono essere nel risultato?

- Questi andranno nella clausola `SELECT`.
- Usa `DISTINCT` se vuoi eliminare i duplicati.

12.3.3 È necessario filtrare i dati?

- **JOIN (`FROM` clause o `WHERE` clause)**:
  - `LEFT/RIGHT/OUTER JOIN` nella clausola `FROM`: Usati quando vuoi tutti i record da una tabella e solo quelli corrispondenti dall'altra (o tutti da entrambe con `FULL OUTER JOIN`).
  - *Theta-join* (join con condizioni generiche) o *equi-join* (join su uguaglianza) nella clausola `WHERE` (stile più vecchio, ma ancora visto) o con `JOIN ... ON` nella clausola `FROM` (stile moderno e preferito).

```sql
-- Stile moderno (preferito)
SELECT S.Surname
FROM STUDENT S JOIN EXAM E ON S.Id = E.Student;
```

```sql
-- Stile più vecchio
SELECT S.Surname
FROM STUDENT S, EXAM E
WHERE S.Id = E.Student;
```

- **Selezione di Valori (`WHERE` clause)**: Filtra le righe basate su condizioni specifiche (es. `Year = 2023`, `Mark >= 18`).
- **Valori Aggregati (`GROUP BY`, `HAVING` clause)**:
  - `GROUP BY`: Raggruppa righe che hanno gli stessi valori in una o più colonne, per poter applicare funzioni di aggregazione (es. `COUNT`, `SUM`, `AVG`, `MIN`, `MAX`) a ciascun gruppo.
  - `HAVING`: Filtra i gruppi creati da `GROUP BY` (simile a `WHERE`, ma `WHERE` filtra le singole righe *prima* dell'aggregazione, `HAVING` filtra i gruppi *dopo* l'aggregazione).

12.4 Ordine di Esecuzione Logica delle Clausole SQL

È importante capire l'ordine logico in cui le clausole SQL vengono processate, perché influenza cosa puoi fare in ciascuna clausola (ad esempio, non puoi usare un alias definito nel `SELECT` all'interno del `WHERE` della stessa query principale).

1. `FROM` (incluse le operazioni di `JOIN`)
2. `WHERE`
3. `GROUP BY`
4. `HAVING`
5. `SELECT`
6. `ORDER BY`
7. `LIMIT / OFFSET` (non mostrato nelle slide, ma comune)

Esempio Pratico: Se scrivi:

```sql
SELECT COUNT(*) AS TotalStudents FROM STUDENT WHERE Age > 20;
```

- `FROM STUDENT`: Prima il DB considera la tabella `STUDENT`.
- `WHERE Age > 20`: Poi filtra gli studenti con età maggiore di 20.
- `SELECT COUNT(*) AS TotalStudents`: Infine, conta le righe rimanenti e assegna l'alias.

Non potresti scrivere `WHERE TotalStudents > 5` (riferendosi all'alias) perché `WHERE` è processato prima di `SELECT`.

12.5 Considerazioni sui JOIN

12.5.1 Diverse Soluzioni per lo Stesso Problema

- `INTERSECT`: Restituisce le righe comuni a due `SELECT` (le due query devono avere lo stesso numero di colonne e tipi di dati compatibili). Utile per trovare "elementi che appartengono a entrambi gli insiemi".
- **JOIN Implicito (nella clausola `WHERE`)**:

```sql
SELECT DISTINCT L.Surname
FROM LECTURER L, STUDENT S
WHERE L.Surname = S.Surname;
```

- **JOIN Esplicito (nella clausola `FROM` con `JOIN ... ON`)**:

```sql
SELECT DISTINCT L.Surname
FROM LECTURER L JOIN STUDENT S ON L.Surname = S.Surname;
```

Quest'ultimo è generalmente **preferito** perché separa le condizioni di join (logica di collegamento tra tabelle) dalle condizioni di filtro (`WHERE`), rendendo la query più leggibile e manutenibile.

12.5.2 Non Esiste "L'Unica Soluzione Corretta"

Spesso ci sono più modi per ottenere lo stesso risultato. L'importante è che la soluzione sia corretta, efficiente e leggibile.

12.6 Implementazione dei JOIN (Teoria dell'Ottimizzatore)

I JOIN sono operazioni potenzialmente costose. Il DBMS (Database Management System) ha un **ottimizzatore** che sceglie la tecnica di JOIN più efficiente basandosi su statistiche delle tabelle (dimensione, distribuzione dei dati, indici presenti). Alcune tecniche comuni:

1. **Nested-loop Join (Join a Cicli Annidati)**:
   - **Come funziona**: Per ogni riga della tabella "esterna" (outer table), scorre tutte le righe della tabella "interna" (inner table) per trovare corrispondenze.
   - **Analogia**: Due cicli `for` annidati.

```
FOR EACH row_R IN Table_R DO
FOR EACH row_S IN Table_S DO
  IF row_R.join_attr = row_S.join_attr THEN
    output (row_R, row_S)
```

   - **Costo**: Può essere molto alto (`O(N*M)`) se non ci sono indici. L'ordine delle tabelle (quale è esterna e quale interna) e la dimensione del buffer influenzano le prestazioni.
2. **Single-loop Join (o Index Nested-loop Join)**:
   - **Come funziona**: Richiede un **indice** sull'attributo di join di una delle due tabelle (solitamente la tabella interna). Per ogni riga della tabella esterna, usa l'indice per cercare rapidamente le corrispondenze nella tabella interna, invece di scansionarla completamente.

```
FOR EACH row_R IN Table_R DO
USE_INDEX_ON_S_TO_FIND (row_R.join_attr)
```

   - **Efficienza**: Molto più efficiente del Nested-loop semplice se l'indice è selettivo.
3. **Sort-merge Join**:
   - **Come funziona**:
     (a) Entrambe le tabelle vengono **ordinate** (sort) in base agli attributi di join.
     (b) Le tabelle ordinate vengono poi "fuse" (merge) scorrendole parallelamente una sola volta per trovare le corrispondenze.

- **Quando è utile**: Efficiente per JOIN tra tabelle grandi, specialmente se i dati sono già ordinati o se l'ordinamento è richiesto da altre operazioni nella query (es. `ORDER BY`). Funziona bene anche per non-equi-join (es. <, >).

4. **Hash-based Join**:
   - **Come funziona**:
     (a) **Build Phase**: Crea una tabella hash in memoria sulla tabella più piccola (la "build table") usando gli attributi di join come chiave hash.
     (b) **Probe Phase**: Scansiona la tabella più grande (la "probe table"). Per ogni riga, calcola l'hash dell'attributo di join e cerca una corrispondenza nella tabella hash creata precedentemente.
   - **Quando è utile**: Solitamente il più efficiente per equi-join su tabelle grandi quando c'è abbastanza memoria per la tabella hash.

L'ottimizzatore sceglie tra queste (e altre) strategie basandosi su un modello di costo.

12.7 Concetti Avanzati dalle Esercitazioni

12.7.1 Ambiguità e Interpretazione

A volte la richiesta può essere ambigua (es. "corsi"). Bisogna capire a quale tabella/entità si riferisce il problema (es. `COURSE` o `EDITION`). Il contesto (es. la presenza di `Year` in `EXAM`) aiuta a disambiguare.

12.7.2 Aggregazione e Filtri su Aggregati

Quando si usa `GROUP BY`, tutti gli attributi nel `SELECT` che non sono funzioni di aggregazione devono essere presenti nel `GROUP BY`. `HAVING COUNT(*) > N` permette di filtrare i gruppi che soddisfano una certa cardinalità.

12.7.3 Subquery nella Clausola `FROM` (Tabelle Derivate o Viste Inline)

Puoi usare il risultato di una `SELECT` come se fosse una tabella nella clausola `FROM`. Questo è utile per calcoli intermedi. **Esempio (da Exercise 4)**: Trovare regioni con più abitanti (da `REGION.Population`) che residenti registrati (conteggio da `RESIDENCE`).

```
SELECT A.Name
FROM REGION AS A,
(SELECT Region, COUNT(*) AS CitizenCount -- Tabella derivata C
 FROM RESIDENCE
 GROUP BY Region) AS C
WHERE A.Name = C.Region AND A.Population > C.CitizenCount;
```

Si può anche creare una `VIEW` per rendere la query principale più pulita se la subquery è complessa o usata spesso.

12.7.4 `NATURAL JOIN`

Un tipo di `JOIN` che unisce automaticamente le tabelle basandosi su colonne che hanno lo **stesso nome e tipo di dati** in entrambe le tabelle. È comodo ma può essere **pericoloso**: se in futuro vengono aggiunte colonne con lo stesso nome ma significato diverso, il `NATURAL JOIN` potrebbe produrre risultati errati o inattesi. È spesso più sicuro usare `JOIN ... ON` specificando esplicitamente le colonne di join.

12.7.5 Subquery Correlate e Non Correlate nella Clausola `WHERE`

- **Non Correlata**: La subquery viene eseguita una sola volta e il suo risultato viene usato dalla query esterna.

```sql
SELECT Name FROM STUDENT WHERE Age > (SELECT AVG(Age) FROM STUDENT);
```

- **Correlata**: La subquery viene eseguita per ogni riga processata dalla query esterna, e dipende da valori della riga corrente della query esterna. **Esempio (Exercise 13 - better solution)**: Titoli di film con meno di 6 attori.

```sql
SELECT F.Director, F.Title
FROM FILM AS F
WHERE 6 > (SELECT COUNT(*)
FROM RECITAL AS R
WHERE F.Code = R.Film); -- F.Code è dalla query esterna
```

  Questo è potente ma può essere meno performante di altre soluzioni se non ottimizzato bene.

12.7.6 `EXISTS` e `NOT EXISTS`

Usati con subquery correlate per verificare l'esistenza (o non esistenza) di righe che soddisfano una certa condizione. `EXISTS (subquery)` restituisce `TRUE` se la subquery restituisce almeno una riga. **Esempio (Exercise 15)**: Film mai proiettati a Berlino.

```sql
SELECT F.Title
FROM FILM AS F
WHERE NOT EXISTS (SELECT *
FROM SCREENING AS S JOIN ROOM AS RM ON S.Room = RM.Code
WHERE RM.City = 'Berlin' AND F.Code = S.Film);
```

`NOT EXISTS` è spesso un modo efficiente per esprimere "anti-join" (trovare elementi in A che non hanno corrispondenza in B).

12.7.7 `IN` e `NOT IN`

`valore IN (subquery)` o `valore IN (lista_valori)`: vero se `valore` è presente nel risultato della subquery o nella lista. **Attenzione con `NOT IN` e `NULL`**: Se la subquery (o la lista) usata con `NOT IN` restituisce anche un solo valore `NULL`, l'intera condizione `NOT IN` diventerà `UNKNOWN` (o `FALSE` in alcuni DBMS), portando a risultati potenzialmente inattesi (spesso nessuna riga restituita). `NOT EXISTS` è generalmente più sicuro in questi casi.

12.7.8 Quantificatori `ALL`, `ANY`, `SOME`

Usati con operatori di comparazione (=, <, >, etc.) e una subquery.

- `valore = ALL (subquery)`: vero se `valore` è uguale a *tutti* i valori restituiti dalla subquery (o se la subquery non restituisce righe). **Esempio (Exercise 19)**: Musei a Londra che hanno *solo* opere di Tiziano.

```sql
SELECT M.Name
FROM MUSEUM AS M
WHERE M.City = 'London' AND
'Tiziano' = ALL (SELECT W.NameA
                 FROM WORK AS W
                 WHERE M.Name = W.NameM);
```

  Questo implica che se il museo ha opere, tutte devono essere di Tiziano. Se il museo non ha opere, la subquery è vuota e `ALL` restituisce `TRUE` (il museo verrebbe listato). Questa è una sfumatura importante di `ALL` con set vuoti. Per esprimere "solo opere di Tiziano e deve avere almeno un'opera di Tiziano", la logica si complica (spesso richiede un `EXISTS` aggiuntivo o una doppia negazione con `NOT EXISTS`).

- `valore = ANY (subquery)` (o `= SOME`): vero se `valore` è uguale ad *almeno uno* dei valori restituiti dalla subquery. `ANY` e `SOME` sono sinonimi. `valore = ANY (subquery)` è equivalente a `valore IN (subquery)`.

12.7.9 Esprimere Condizioni Universali ("per tutti")

"Film che hanno *sempre* guadagnato più di $500" (Exercise 17). Un modo è trovare il minimo guadagno per quel film e verificare che sia `>= 500`.

```
SELECT F.Title
FROM FILM AS F
WHERE 500 <= (SELECT MIN(S.Profits)
FROM SCREENING AS S
WHERE F.Code = S.Film);
```

Se un film non ha proiezioni, la subquery `MIN(S.Profits)` restituirà `NULL`. La comparazione `500 <= NULL` è `UNKNOWN`, quindi il film non verrà restituito (corretto). Un altro modo classico per esprimere "per tutti gli X, P(X) è vero" è usare la doppia negazione: "non esiste un X per cui P(X) è falso". Questo si traduce bene con `NOT EXISTS`.

Questi appunti dovrebbero coprire i concetti teorici chiave presentati nelle slide, con un focus sulla comprensione e applicazione pratica. Ricorda che la pratica è fondamentale per padroneggiare SQL e la progettazione di query efficienti!

Capitolo 13

Laboratorio 3 e 4: Architettura e Transazioni

13.1 Architettura di un DBMS

13.1.1 Cos'è un DataBase Management System (DBMS)?

Un DBMS è un software progettato per **creare e gestire database**. Il suo scopo principale è fornire un modo **efficiente e persistente** per creare, recuperare, aggiornare e gestire enormi quantità di dati.

- **Efficienza:** Deve rispondere rapidamente alle interrogazioni.
- **Persistenza:** I dati devono sopravvivere allo spegnimento del sistema o a crash.
- **Gestione di grandi quantità di dati:** Ottimizzato per moli di dati che non entrerebbero in memoria RAM.

Esempio pratico: Pensa a Prisma (per database SQL) o Mongoose (per MongoDB). Quando esegui un comando come:

```
// Con Prisma
const users = await prisma.user.findMany();

// Con Mongoose
const users = await User.find();
```

è il DBMS sottostante (es. PostgreSQL, MySQL, MongoDB Server) che esegue il lavoro pesante di trovare i dati sul disco, filtrarli e restituirli.

13.1.2 Implementare un DBMS Relazionale

Se volessimo costruire un DBMS relazionale da zero, dovrebbe essere in grado di:

1. **Memorizzare relazioni come tabelle:** Ad esempio, tabelle $R(A, B, C)$, $S(A, B)$, $T(A, B)$. Ogni riga è una tupla (record), ogni colonna un attributo.
2. **Eseguire query SQL:** Come:

   ```
   SELECT A, B FROM R WHERE B = 2000;
   ```

   o query più complesse con JOIN:

   ```
   SELECT R.A, T.B FROM R, S, T WHERE R.B = S.A AND S.B = T.A;
   ```

13.1.3 Funzioni Chiave di un DBMS

Un DBMS deve gestire efficientemente diverse operazioni e scenari:

- Ricerca e aggiornamento tuple.
- Ordine di esecuzione delle query SQL (ottimizzazione).
- Accessi ripetuti alle stesse locazioni su disco (caching).
- Operazioni concorrenti di lettura e scrittura (Controllo della Concorrenza).
- Controllo degli accessi (permessi).
- Disaster recovery (Ripristino dopo guasti).
- API (Application Programming Interfaces) per l'interazione con le applicazioni.

13.1.4 Architettura di un DBMS Relazionale (Componenti Principali)

L'architettura di un DBMS è complessa e modulare. I componenti principali possono essere raggruppati in tre macro-aree:

1. **Query Processor (Processore delle Query):** Responsabile della trasformazione e dell'esecuzione delle query.
    - **DDL Compiler:** Interpreta comandi DDL (Data Definition Language, es. `CREATE TABLE`, `ALTER TABLE`) e aggiorna i metadati (lo schema del database).
    - **Query Compiler:**
        - *Parsing:* Analisi sintattica della query SQL. Costruisce un albero di parsing.
        - *Preprocessing (Validazione Semantica):* Verifica che tabelle e colonne esistano, tipi di dati compatibili. Trasforma l'albero in un albero di operazioni algebriche (algebra relazionale).
        - *Optimization:* Trasforma la query originale nella sequenza di operazioni più efficiente. Decide l'ordine dei join, l'uso degli indici, ecc.
    - **Execution Engine (Motore di Esecuzione):** Esegue il piano di query scelto dall'ottimizzatore, interagendo con gli altri componenti.
2. **Resource Manager (Gestore delle Risorse):** Gestisce l'accesso ai dati su disco e in memoria.
    - **Storage Manager (Gestore della Memoria Secondaria):** Tiene traccia della posizione dei file sul disco. Recupera blocchi di dati dal disco.
    - **Buffer Manager (Gestore del Buffer):** Gestisce una porzione della memoria RAM (il buffer pool) per memorizzare temporaneamente le pagine di dati lette dal disco. Se una pagina richiesta è già nel buffer (cache hit), l'accesso è veloce. Altrimenti (cache miss), la pagina viene caricata dal disco.
    - **Index/File/Record Manager:** Conosce lo schema del database e le strutture dati (come indici B-Tree) per un accesso efficiente ai dati.
3. **Transaction Manager (Gestore delle Transazioni):** Assicura le proprietà ACID delle transazioni.
    - **Transaction Manager (componente specifico):** Supervisiona le transazioni.
    - **Logging and Recovery Manager:**
        - *Logging:* Registra tutte le modifiche al database in un file di log (Write-Ahead Logging - WAL).
        - *Recovery:* In caso di crash, usa il log per ripristinare il database a uno stato consistente.
    - **Concurrency Control Manager:** Gestisce l'accesso concorrente ai dati da parte di più transazioni, usando meccanismi come i lock. Mantiene una **Lock Table**.

Gli utenti e le applicazioni interagiscono con il DBMS, così come l'amministratore per le operazioni DDL.

13.1.5 Il Percorso di una Query (Esempio Semplificato)

Consideriamo la query:

```
SELECT A, B FROM R1, R2 WHERE C = D AND C = 'c';
```

1. **Query Compiler:**
   - Parsing e validazione.
   - Trasformazione in algebra relazionale (forma logica), ad esempio: $\pi_{A,B}(\sigma_{C=D \wedge C='c'}(R1 \bowtie R2))$
   - Ottimizzazione (piano di esecuzione fisico), ad esempio, potrebbe decidere di eseguire: $\pi_{A,B}(\sigma_{C=D}((\sigma_{C='c'}(R1)) \bowtie R2))$
     (prima si filtra $R1$, poi si fa il join con $R2$, poi si applica l'altro filtro e infine la proiezione).
2. **Execution Engine:**
   - Richiede al Resource Manager i blocchi di $R1$ necessari per calcolare $\sigma_{C='c'}(R1)$.
   - Il **Resource Manager** (tramite Storage Manager e Buffer Manager) carica i blocchi di $R1$ dal disco al buffer.
   - L'Execution Engine (o l'Index/File/Record Manager) applica il filtro $C =' c'$ sui dati di $R1$ nel buffer. Il risultato intermedio resta nel buffer.
   - Richiede i blocchi di $R2$. Vengono caricati nel buffer.
   - Esegue il join tra il risultato intermedio di $R1$ e $R2$, applica il filtro $C = D$ e la proiezione $\pi_{A,B}$, operando sui dati nel buffer.
   - Restituisce il risultato finale.

Durante tutte le operazioni di modifica, il Transaction Manager è coinvolto per il logging e il controllo della concorrenza.

13.2 Transazioni e Controllo della Concorrenza

13.2.1 Transazioni

Le operazioni su un database raramente coinvolgono una singola istruzione SQL. Spesso, una singola "azione logica" richiede una serie di query. *Esempio:* Un trasferimento bancario da conto A123 a B456 di 100€.

```
START TRANSACTION;
UPDATE Conto SET Saldo = Saldo - 100 WHERE NumeroConto = 'A123';
UPDATE Conto SET Saldo = Saldo + 100 WHERE NumeroConto = 'B456';
COMMIT; -- oppure ROLLBACK in caso di errore
```

Queste due operazioni devono avvenire insieme. O entrambe hanno successo, o nessuna delle due. Questo gruppo di operazioni è una **transazione**. Una transazione è l'esecuzione di un programma utente (una o più operazioni di lettura/scrittura) vista dal DBMS come un'unità atomica.

Problemi principali da risolvere:

1. Esecuzione Concorrente di Transazioni.
2. Crash Recovery (Ripristino dopo guasti).

13.2.2 Motivazioni per la Concorrenza

- Gli accessi al disco sono lenti; la CPU non dovrebbe attendere passivamente.
- Per migliorare le prestazioni, più transazioni vengono eseguite in modo **concorrente** (i loro passi si mescolano, *interleaving*), mantenendo la CPU occupata.
- **Problema:** L'utente non deve percepire questa "mescolanza". Ogni transazione deve apparire come se fosse eseguita da sola, in isolamento.

13.2.3 Motivazioni per il Crash Recovery

- Una transazione potrebbe non terminare come previsto a causa di un evento imprevisto (errore software, crash hardware).
- **Problema:** Il DBMS deve assicurare che le altre transazioni non siano affette dall'esecuzione incorretta e che il database rimanga in uno **stato consistente**.

13.2.4 Proprietà ACID delle Transazioni

Per garantire un'esecuzione sicura e concorrente, ogni transazione deve avere le seguenti proprietà:

- **A - Atomicity (Atomicità):** Una transazione è un'unità atomica. O tutte le sue operazioni vengono eseguite, o nessuna. Non ci deve essere uno stato intermedio parziale persistente. In caso di fallimento, le modifiche parziali vengono annullate (rollback).
- **C - Consistency (Consistenza):** Una transazione porta il database da uno stato consistente a un altro stato consistente. Le *regole di integrità* del database (es. chiavi primarie, foreign key, vincoli `CHECK`) devono essere soddisfatte alla fine della transazione.
- **I - Isolation (Isolamento):** Ogni transazione deve apparire come se fosse eseguita in isolamento, senza interferenze da altre transazioni concorrenti. Il risultato finale dell'esecuzione concorrente deve essere equivalente a quello che si otterrebbe eseguendole una dopo l'altra in una qualche sequenza (serializzazione).
- **D - Durability (Durabilità):** Una volta che una transazione è stata "commessa" (`COMMIT`), le sue modifiche sono permanenti e devono sopravvivere a fallimenti successivi.

13.2.5 Schedule (Pianificazione)

- Una transazione è una sequenza di operazioni di lettura (read) e scrittura (write) su oggetti del database.
- Ogni transazione specifica la sua azione finale: `commit` (successo) o `abort` (fallimento, con annullamento delle azioni).
- Uno **schedule** è una sequenza di azioni (read, write, commit, abort) prese da un insieme di transazioni, preservando l'ordine delle azioni *all'interno* di ciascuna transazione.
- **Schedule Seriale:** Le transazioni sono eseguite una dopo l'altra, senza interleaving.

13.2.6 Schedule Serializzabile

Uno schedule è **serializzabile** se il suo effetto su un'istanza consistente del database è **identico** a quello di un qualche schedule seriale completo. Questo è l'obiettivo del controllo di concorrenza: permettere l'interleaving per le performance, garantendo la correttezza.

13.2.7 Anomalie dell'Esecuzione Interleaved (Conflitti)

L'interleaving non controllato può portare a stati inconsistenti. Tipi di conflitti (secondo le slide):

- **Write-Read Conflict (Dirty Read):** T_2 legge un oggetto scritto da T_1, ma T_1 non ha ancora fatto `commit`. Se T_1 fa `abort`, T_2 ha letto dati "sporchi".
- **Read-Write Conflict (Unrepeatable Read):** T_2 scrive un oggetto letto da T_1, prima che T_1 faccia `commit`. Se T_1 legge di nuovo lo stesso oggetto, ottiene un valore diverso.
- **Write-Write Conflict (Lost Update):** T_2 scrive un oggetto anch'esso scritto da T_1. Una delle due scritture potrebbe andare persa.
- **Phantom Anomalies (Anomalie Fantasma):** Una transazione esegue una query che restituisce un insieme di righe. Un'altra transazione inserisce o cancella righe che soddisfano i criteri della query. Se la prima transazione riesegue la query, ottiene un insieme diverso di righe ("fantasmi" appaiono o scompaiono).

13.2.8 Livelli di Isolamento delle Transazioni

SQL definisce diversi livelli di isolamento, che offrono un compromesso tra correttezza e performance.

- **READ UNCOMMITTED:** Permette Dirty Reads, Unrepeatable Reads, Phantoms. Massime performance, minima consistenza.
- **READ COMMITTED:** Previene Dirty Reads. Permette Unrepeatable Reads, Phantoms. La transazione legge solo dati committati.
- **REPEATABLE READS:** Previene Dirty Reads, Unrepeatable Reads. Permette Phantoms. Garantisce che rileggendo un dato si ottenga lo stesso valore, ma nuove righe ("fantasmi") possono apparire.
- **SERIALIZABLE:** Previene Dirty Reads, Unrepeatable Reads, Phantoms. Massima consistenza.

Tabella Riepilogativa:

Livello	Dirty read	Lost update	Unrepeatable read	Phantom
READ UNCOMMITTED	può verificarsi	può verificarsi	può verificarsi	può verificarsi
READ COMMITTED	non si verifica	può verificarsi	può verificarsi	può verificarsi
REPEATABLE READS	non si verifica	non si verifica	non si verifica	può verificarsi
SERIALIZABLE	non si verifica	non si verifica	non si verifica	non si verifica

13.2.9 Approcci al Controllo della Concorrenza

- **Restrittivo (Pessimistico):** Previene i conflitti prima che accadano, tipicamente usando **lock**.
  - **Strict Two-Phase Locking (Strict 2PL):**
    1. Fase di crescita: La transazione acquisisce lock (condivisi S per lettura, esclusivi X per scrittura).
    2. Fase di decrescita: La transazione rilascia i lock. In 2PL "stretto", tutti i lock vengono rilasciati solo al `commit` o `abort`. Una volta rilasciato un lock, non se ne possono acquisire altri.

    Garantisce la serializzabilità. Il Lock Manager gestisce i lock in una Lock Table.
- **Ottimistico:** Assume conflitti rari. Le transazioni procedono senza lock. Prima del `commit`, una fase di **validazione** verifica conflitti. Se presenti, la transazione abortisce e riparte. Fasi: Read (in area privata), Validation, Write (se validazione OK).
- **Timestamping:** Ad ogni transazione è assegnato un timestamp. Il DBMS usa i timestamp per imporre un ordine seriale. Se un'operazione viola l'ordine, la transazione abortisce.

13.2.10 Deadlock (Stallo)

Con il locking pessimistico, può verificarsi un deadlock: T_1 ha lock su A e aspetta B, T_2 ha lock su B e aspetta A.

- **Prevenzione Deadlock:** Assegnare priorità (es. timestamp) e usare politiche come Wait-Die o Wound-Wait.
- **Rilevamento e Risoluzione Deadlock:** Permettere i deadlock, rilevarli (es. con un waits-for graph) e risolverli abortendo una transazione ("vittima"). Oppure, usare un timeout.

13.2.11 Crash Recovery

Il Logging and Recovery Manager deve assicurare Atomicità e Durabilità. **ARIES (Advanced Recovery and Integrated Extraction System):** Algoritmo di recovery comune.

- **Principi ARIES:**
  1. **Write-Ahead Logging (WAL):** Modifiche scritte nel log *prima* della scrittura della pagina dati su disco. Record di log con info per Undo/Redo.
  2. **Repeating History During Redo:** Al riavvio dopo crash, ARIES riapplica modifiche dal log (da ultimo checkpoint) per riportare DB allo stato del crash (incluse transazioni non committate).
  3. **Logging Changes During Undo:** L'annullamento di un'operazione è loggato con un **Compensation Log Record (CLR)**. I CLR non vengono mai annullati.
- **Log File:** Traccia tutte le azioni. Record con **Log Sequence Number (LSN)** univoco. Record di Log: `<LSN, TransactionID, PageID, RedoInfo, UndoInfo, PreviousLSN_for_this_TX>`.
- **Strutture Dati per Logging (in memoria):**
  - **Dirty Page Table (DPT):** Pagine modificate in buffer ma non su disco.
  - **Transaction Table:** Transazioni attive, stato, ultimo LSN.
- **Checkpoint (in ARIES è "Fuzzy"):** Snapshot dello stato per ridurre tempo di recovery.
  1. Scrive record `begin_checkpoint`.
  2. Scrive record `end_checkpoint` con DPT e Transaction Table.
  3. Scrive LSN del `begin_checkpoint` in un Master Record su disco.

  Non interrompe le operazioni normali e non forza la scrittura di tutte le dirty pages.
- **Fasi di Recovery ARIES dopo un Crash:**
  1. **Analysis Phase:** Scansiona log da ultimo checkpoint. Ricostruisce DPT e Transaction Table al momento del crash. Identifica transazioni "perdenti" (attive al crash).
  2. **Redo Phase:** Scansiona log in avanti. Riapplica modifiche (Redo) per le pagine sporche se necessario, per portare il DB allo stato esatto del crash.
  3. **Undo Phase:** Scansiona log all'indietro. Annulla operazioni delle transazioni "perdenti", scrivendo CLR per ogni operazione annullata.

Capitolo 14

Laboratorio 5: Progettazione

Guarda le slide del prof.

La parte di progettazione è puramente grafica (diagrammi ER), quindi io non sto qua ad impazzire col LaTeX e tu fai prima a guardare le slide.

Capitolo 15

Laboratorio 6: Indici e B+ Alberi

15.1 Indici: Concetti Fondamentali per gli Esercizi

Un indice è una struttura dati che migliora la velocità di recupero dei dati. Immaginalo come l'indice analitico di un libro.

15.1.1 Chiave di Ricerca (Search Key)

L'attributo (o insieme di attributi) su cui si basa l'indice.

- **Non necessariamente una chiave primaria.** Quindi, può contenere duplicati.
- Se non specificato diversamente, useremo solo "chiave".

15.1.2 Label

Ciò a cui punta la chiave dell'indice. Può essere:

1. **I record di dati stessi:** L'indice *è* i dati, ordinati secondo la chiave.
2. **L'identificatore del record (RID):** Un puntatore alla posizione fisica del record.
3. **Una lista di RID:** Se una chiave ha più record associati (duplicati).

15.1.3 Esempio Pratico dei Tipi di Label

Consideriamo una tabella `Prodotti` di un e-commerce con i seguenti dati:

ID	Nome	Categoria	Prezzo
1	iPhone 15	Elettronica	999
2	MacBook Pro	Elettronica	1999
3	AirPods	Elettronica	199
4	Felpa Nike	Abbigliamento	89
5	Scarpe Adidas	Abbigliamento	119

Tabella 15.1: Tabella Prodotti di esempio

Vediamo come funzionerebbero i tre tipi di indici quando si esegue una query come:

```
SELECT * FROM Prodotti WHERE Categoria = 'Elettronica';
```

1. **Tipo 1 (Record di dati)**:
   In questo caso, l'indice è la tabella stessa, ma ordinata per la colonna Categoria:

Categoria	ID	Nome	Prezzo
Abbigliamento	4	Felpa Nike	89
Abbigliamento	5	Scarpe Adidas	119
Elettronica	1	iPhone 15	999
Elettronica	2	MacBook Pro	1999
Elettronica	3	AirPods	199

Tabella 15.2: Indice sulla Categoria (Tipo 1)

Quando cerchi "Elettronica":

(a) Il database trova rapidamente la prima voce "Elettronica" (perché l'indice è ordinato)

(b) Legge i record consecutivi finché la categoria è "Elettronica"

(c) Restituisce direttamente questi dati (poiché l'indice contiene già tutti i dati)

Questo tipo è essenzialmente una "tabella ordinata" e occupa molto spazio perché duplica i dati.

2. **Tipo 2 (RID - Singolo puntatore)**:
In questo caso, l'indice contiene coppie [chiave, puntatore] per ogni record:

Categoria	Puntatore
Abbigliamento	RID(blocco1,slot4) $\rightarrow$ punta a Felpa Nike (ID 4)
Abbigliamento	RID(blocco2,slot1) $\rightarrow$ punta a Scarpe Adidas (ID 5)
Elettronica	RID(blocco1,slot1) $\rightarrow$ punta a iPhone 15 (ID 1)
Elettronica	RID(blocco1,slot2) $\rightarrow$ punta a MacBook Pro (ID 2)
Elettronica	RID(blocco1,slot3) $\rightarrow$ punta a AirPods (ID 3)

Tabella 15.3: Indice sulla Categoria (Tipo 2)

Quando cerchi "Elettronica":

(a) Il database scansiona l'indice cercando tutte le voci con "Elettronica"

(b) Per ogni voce trovata, segue il puntatore RID per accedere al record completo

(c) Il processo si ferma quando trova una categoria diversa o raggiunge la fine dell'indice

Devi scorrere l'indice per trovare tutte le occorrenze della categoria "Elettronica", ma l'indice è molto più piccolo della tabella originale.

3. **Tipo 3 (Lista di RID)**:
In questo caso, l'indice raggruppa tutti i puntatori per valore unico di chiave:

Categoria	Lista di puntatori
Abbigliamento	[RID(blocco1,slot4), RID(blocco2,slot1)] $\rightarrow$ punta ai prodotti ID 4 e 5
Elettronica	[RID(blocco1,slot1), RID(blocco1,slot2), RID(blocco1,slot3)] $\rightarrow$ punta ai prodotti ID 1, 2 e 3

Tabella 15.4: Indice sulla Categoria (Tipo 3)

Quando cerchi "Elettronica":

(a) Il database cerca nell'indice la voce "Elettronica" (una sola ricerca!)

(b) Trova immediatamente l'intera lista di puntatori a tutti i prodotti elettronici

(c) Segue ogni puntatore per recuperare i record completi

Questo formato è più compatto rispetto al Tipo 2 quando ci sono molti valori duplicati nella colonna di indice. Nell'esempio, invece di avere 5 voci come nel Tipo 2, abbiamo solo 2 voci (una per ogni categoria unica).

Confronto pratico delle prestazioni:
Immagina di avere 1 milione di prodotti e 10 categorie diverse:

- **Tipo 1:** Molto veloce nelle letture perché è già ordinato, ma occupa più spazio e le modifiche sono lente (devi riordinare).
- **Tipo 2:** L'indice è più piccolo (solo chiave + puntatore), ma per trovare tutti i prodotti di una categoria devi comunque scorrere l'indice sequenzialmente. Avresti 1 milione di voci nell'indice.
- **Tipo 3:** L'indice è ancora più compatto (solo 10 voci, una per categoria), e trovare tutti i prodotti di una categoria è efficientissimo - una sola ricerca nell'indice per ottenere tutti i puntatori ai prodotti.

In pratica, i database moderni usano principalmente strutture simili al Tipo 2 e Tipo 3, spesso combinate con strutture ad albero (B-tree, B+ tree) per ottimizzare ulteriormente le ricerche.

15.1.4 Osservazioni Utili per gli Esercizi

- **Un solo indice di Tipo 1:** Puoi avere i dati ordinati fisicamente in un solo modo.
- **Dimensione Indice Tipo 1:** L'indice è grande quanto i dati stessi.
- **Duplicati nelle chiavi:** Una chiave di ricerca (es. "10") può apparire più volte.
- **Tipo 3 è compatto (per chiavi con molti duplicati):** Ma le "label" (liste di RID) hanno dimensione variabile.

15.1.5 Esercizio a Risposta Multipla

Domanda: Dato un singolo file, quale affermazione è vera?

- A. Al massimo un indice "data record" (Tipo 1) può essere usato alla volta.
- B. Solo un indice "record identifier" (Tipo 2) può essere usato alla volta.
- C. L'indice "record identifier" è più compatto dell'indice "record identifier list" (Tipo 3).
- D. L'indice "data record" (Tipo 1) è più utile quando la dimensione dei dati è maggiore.

Strategia e Soluzione: Analizziamo ogni opzione:

- **A. VERA.** I dati possono essere fisicamente ordinati in un solo modo.
- **B. FALSA.** Puoi avere più indici basati su RID su diverse colonne.
- **C. FALSA (generalmente).** Per chiavi con molti duplicati, Tipo 3 è più compatto.
- **D. FALSA (non necessariamente).** L'utilità dipende dall'accesso, la dimensione può essere uno svantaggio.

Risposta corretta: A

15.2 Creazione di Indici in SQL

La sintassi comune per gli indici, utile per esercizi teorici:

```
CREATE [UNIQUE] INDEX IndexName
ON TableName (AttributesList);

-- Esempio:
CREATE INDEX EmpIdx
ON Employee (Surname, Name);

-- Per rimuovere un indice:
DROP INDEX IndexName;
DROP INDEX EmpIdx;
```

- `UNIQUE`: Assicura che i valori della chiave nell'indice siano unici.
- La sintassi `CREATE INDEX` non è pienamente standard SQL, ma è la più comune.

15.3 Classificazione degli Indici (Importante per esercizi teorici)

Queste classificazioni sono cruciali per capire le proprietà e le performance.

1. **Primario vs. Secondario:**
    - **Primario:** L'indice è sulla chiave primaria E i dati nel file sono ordinati secondo quella stessa chiave.
    - **Secondario:** Tutti gli altri indici.
2. **Denso vs. Sparso:**
    - **Denso:** C'è una voce nell'indice per *ogni valore di chiave di ricerca* presente nel file di dati.
    - **Sparso:** Ci sono voci nell'indice solo per *alcuni* dei valori di chiave di ricerca. Richiede che il file di dati sia ordinato secondo la chiave dell'indice.
3. **Clusterizzato vs. Non Clusterizzato (Clustered vs. Unclustered):**
    - **Clusterizzato:** L'ordine dei record nel file di dati rispecchia l'ordine delle voci nell'indice. Si può avere al massimo un indice clusterizzato per tabella.
    - **Non Clusterizzato:** L'ordine dei record nel file di dati *non* corrisponde all'ordine delle voci nell'indice.

15.3.1 Esempi Pratici dei Tipi di Indici

Consideriamo la seguente tabella `Studenti`:

ID	Nome	Età	Facoltà
101	Marco Rossi	22	Informatica
102	Laura Bianchi	20	Economia
103	Paolo Verdi	24	Informatica
105	Giulia Neri	21	Ingegneria
107	Luca Ferrari	23	Economia
110	Anna Romano	19	Informatica

Tabella 15.5: Tabella Studenti di esempio

Illustriamo come sarebbero strutturati i diversi tipi di indici su questa tabella:

1. Indice Primario, Denso e Clusterizzato (su ID)

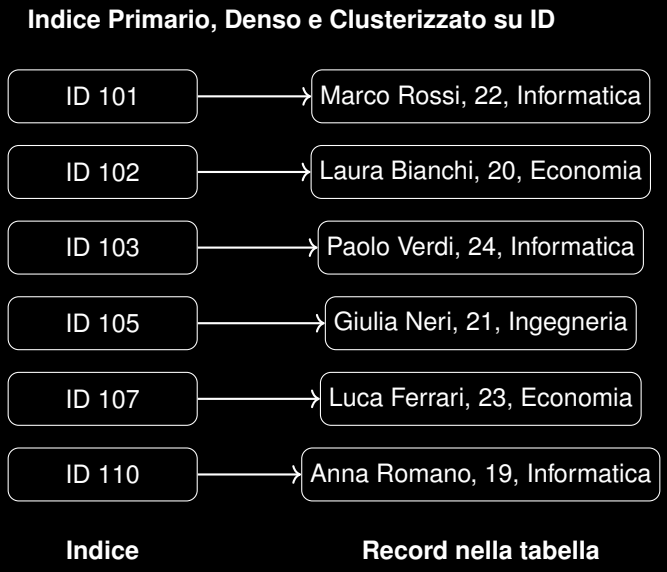


Figura 15.1: Indice Primario, Denso e Clusterizzato su ID

Caratteristiche:

- **Primario:** L'indice è sulla chiave primaria (ID) e i dati sono fisicamente ordinati per ID.
- **Denso:** C'è una voce nell'indice per ogni record nel file dati.
- **Clusterizzato:** L'ordine fisico dei dati (crescente per ID) corrisponde all'ordine dell'indice.
- **Vantaggi:** Ricerca efficiente per ID e per intervalli di ID (es. `ID BETWEEN 101 AND 105`).

2. Indice Sparso e Clusterizzato (su ID)

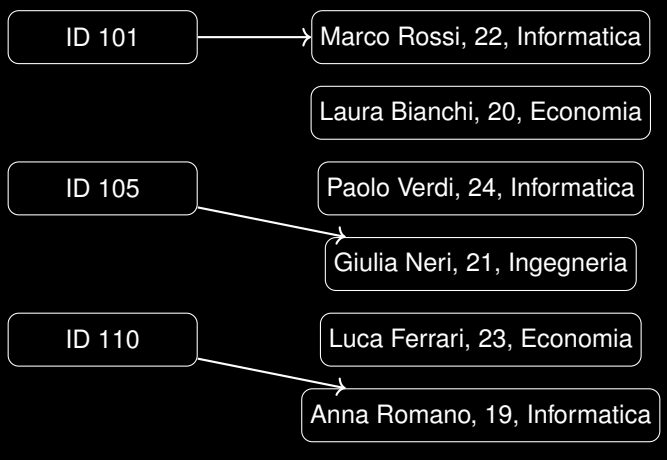


Figura 15.2: Indice Sparso e Clusterizzato su ID

Caratteristiche:

- **Sparso:** Solo alcune chiavi (ogni 2-3 record in questo esempio) hanno una voce nell'indice.

- **Clusterizzato:** L'ordine fisico dei dati è ancora per ID crescente.
- **Funzionamento:** Per trovare `ID=103`, l'indice trova che `101 <= 103 < 105` e inizia la ricerca dal record con ID 101, scorrendo sequenzialmente.
- **Vantaggio:** Occupa meno spazio dell'indice denso, efficace quando i dati sono ordinati.
- **Svantaggio:** Meno efficiente nelle ricerche puntuali rispetto all'indice denso.

3. Indice Secondario, Denso e Non Clusterizzato (su Facoltà)

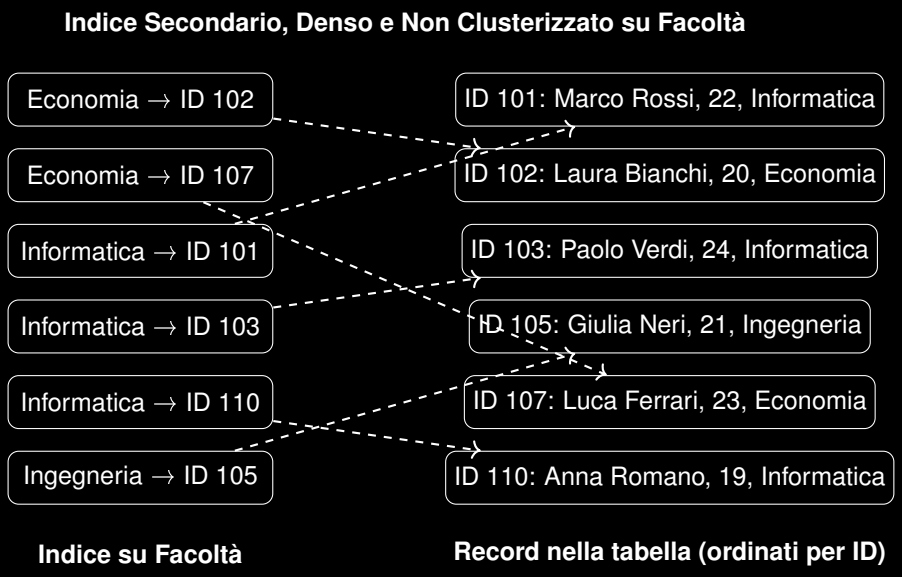


Figura 15.3: Indice Secondario, Denso e Non Clusterizzato su Facoltà

Caratteristiche:

- **Secondario:** L'indice è su un attributo diverso dalla chiave di ordinamento dei dati (i dati sono ordinati per ID, l'indice è su Facoltà).
- **Denso:** C'è una voce nell'indice per ogni record nel file dati.
- **Non Clusterizzato:** L'ordine delle voci nell'indice (alfabetico per Facoltà) è diverso dall'ordine fisico dei dati (per ID).
- **Vantaggi:** Efficiente per ricerche sulla Facoltà (es. `SELECT * FROM Studenti WHERE Facoltà = 'Informatica'`).
- **Svantaggi:** Per trovare tutti gli studenti di Informatica, i record potrebbero essere sparsi in blocchi diversi, richiedendo più accessi al disco.

Confronto dei Costi di Accesso

Tipo di Indice	Costo per Ricerca Puntuale	Costo per Ricerca per Intervallo
Denso e Clusterizzato	Basso (accesso diretto)	Molto basso (record contigui)
Sparso e Clusterizzato	Medio (ricerca nell'intervallo)	Basso (record contigui, ma più scansione sequenziale)
Secondario, Denso e Non Clusterizzato	Basso per l'attributo dell'indice, ma accessi random	Alto (record non contigui, accessi casuali a blocchi diversi)

Tabella 15.6: Confronto dei costi di accesso per tipo di indice

Esempio di query e costo:

```
-- Usando l'indice primario, denso e clusterizzato
SELECT * FROM Studenti WHERE ID = 103;  -- Accesso diretto, veloce

-- Usando l'indice sparso e clusterizzato
SELECT * FROM Studenti WHERE ID = 103;  -- Trova l'intervallo (101-105), poi scansione
↪  sequenziale dal primo record dell'intervallo. Poiché l'indice è sparso, non tutti gli ID
↪  hanno una voce nell'indice, quindi dopo aver identificato l'intervallo corretto, il DBMS
↪  deve scorrere i record in sequenza fino a trovare l'ID 103.

-- Usando l'indice secondario, denso e non clusterizzato
SELECT * FROM Studenti WHERE Facoltà = 'Informatica';  -- Trova le chiavi ma i record sono
↪  sparsi
```

15.3.2 Esercizio a Risposta Multipla

Domanda: Dato un singolo file, quale affermazione è falsa?

- A. Un indice denso potrebbe essere secondario.
- B. Un indice sparso potrebbe essere clusterizzato.
- C. Un indice sparso potrebbe essere non clusterizzato.
- D. Un indice secondario potrebbe essere clusterizzato.

Strategia e Soluzione:

- **A. VERA.** Un indice denso su una colonna non di ordinamento dei dati è secondario.
- **B. VERA.** Un indice sparso richiede che i dati siano ordinati, quindi è clusterizzato.
- **C. FALSA.** Un indice sparso DEVE essere clusterizzato per funzionare (per poter saltare record).
- **D. VERA.** Puoi ordinare fisicamente la tabella secondo un indice secondario; questo lo renderebbe clusterizzato.

Risposta corretta (la falsa): C

15.3.3 Esempio di Performance (Ricerca con Indice Denso)

Questo esempio numerico mostra i vantaggi di un indice.

- **Scenario Dati:** 1.000.000 tuple, 100.000 blocchi da 4096 byte. Spazio: 400 MB.
- **Scenario Indice Denso:**
  - Chiave: 30 byte, Puntatore (RID): 8 byte. Coppia: 38 byte.
  - Per blocco indice: $4096/38 \approx 100$ coppie (arrotondato per difetto, le slide dicono 100).
  - Blocchi per indice: $1.000.000/100 = 10.000$ blocchi.
  - Spazio per indice: 10.000×4096 byte $= 40$ MB. Può stare in memoria.
- **Ricerca:** Con indice in memoria, ricerca binaria.
  - Accessi per trovare chiave nell'indice: $\log_2(10.000) \approx 13 - 14$.
  - Molto meglio che scansionare 100.000 blocchi di dati.

15.4 Alberi B+ (B+ Trees)

Gli alberi B+ sono una struttura dinamica comune per gli indici. Mantengono i blocchi pieni tra la metà e la totalità.

15.4.1 Parametro n (Ordine)

Ogni blocco (nodo) ha spazio per:

- n chiavi di ricerca
- $n + 1$ puntatori

15.4.2 Esercizio: Trovare il Valore di n

Dati:

- Dimensione Blocco: 4096 Byte
- Dimensione Chiave: 4 Byte
- Dimensione Puntatore: 8 Byte
- Nessun header di blocco

Obiettivo: Trovare il più grande intero n tale che n chiavi e $n + 1$ puntatori stiano in un blocco.
Formula:

$$(\text{Dimensione Chiave} \times n) + (\text{Dimensione Puntatore} \times (n + 1)) \leq \text{Dimensione Blocco}$$

Svolgimento:

$$4n + 8(n + 1) \leq 4096$$
$$4n + 8n + 8 \leq 4096$$
$$12n + 8 \leq 4096$$
$$12n \leq 4088$$
$$n \leq \frac{4088}{12}$$
$$n \leq 340.66\ldots$$

Poiché n deve essere un intero, il valore massimo è $n = 340$.
Risposta corretta: A. 340

15.4.3 Regole degli Alberi B+

1. **Nodi Foglia:**
   - Contengono copie delle chiavi dal file di dati, in ordine.
   - L'ultimo puntatore punta al *prossimo nodo foglia* (lista linkata).
   - Gli altri n puntatori puntano ai *record di dati effettivi* (o RID).
   - Un nodo foglia deve essere almeno mezzo pieno (circa $\lceil (n + 1)/2 \rceil$ puntatori usati, escluso il puntatore al fratello).
2. **Nodi Interni (Non Foglia):**
   - I puntatori puntano ad altri nodi dell'albero al livello inferiore.
   - Un nodo interno con k chiavi ha $k + 1$ puntatori.
   - Se un nodo interno ha chiavi $K_1, \ldots, K_k$, il primo puntatore è per valori $< K_1$, il secondo per $\geq K_1$ e $< K_2$, ecc., l'ultimo per $\geq K_k$.
   - Un nodo interno deve avere almeno $\lceil (n + 1)/2 \rceil$ puntatori usati (figli).
3. **Radice (Root):**
   - Se non è una foglia, ha almeno due figli.
   - Può essere meno piena della metà.

15.4.4 Nodi Interni vs. Nodi Foglia

Un nodo interno contiene chiavi che definiscono intervalli e puntatori ad altri nodi figli. Un nodo foglia contiene le chiavi effettive presenti nei dati e puntatori ai record corrispondenti.

15.4.5 Esempio di Albero B+

Si visualizza un albero B+ con un certo ordine n. La radice punta a nodi interni o foglie. I nodi interni guidano la ricerca verso i livelli inferiori in base agli intervalli di chiave. I nodi foglia contengono le chiavi finali e i puntatori ai dati, e sono collegati tra loro per scansioni di intervallo.

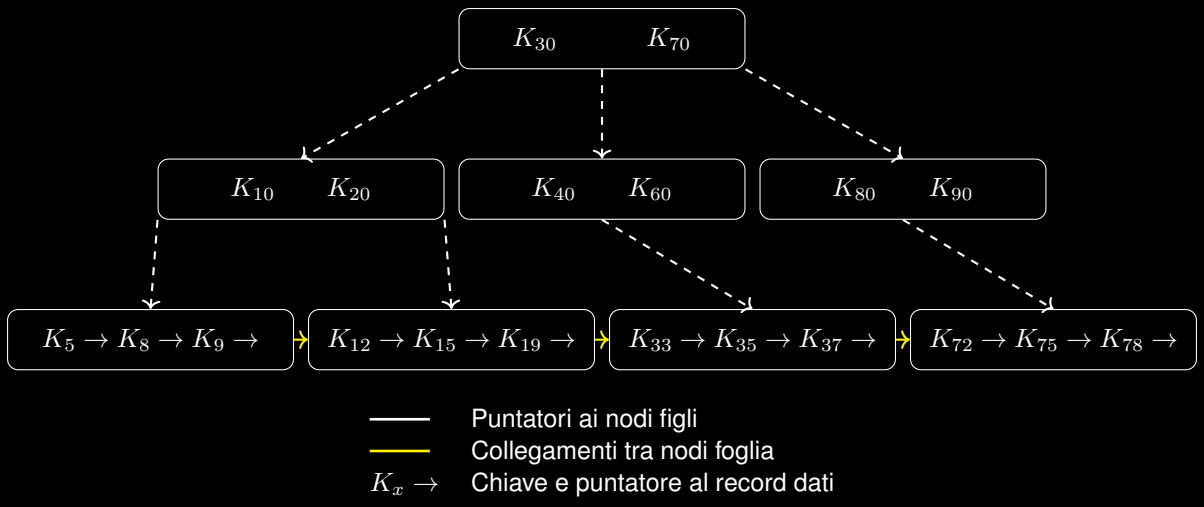


Figura 15.4: Struttura di un albero B+ di esempio (ordine $n = 3$)

Nell'esempio, notiamo che:

- I nodi interni (radice e livello intermedio) contengono chiavi che definiscono gli intervalli di ricerca.
- I nodi foglia contengono le chiavi effettive e i puntatori ai record di dati.
- I nodi foglia sono collegati tra loro (linee blu) per facilitare la scansione sequenziale.
- Per cercare un valore (es. K_{35}), seguiamo il percorso: radice $\to$ nodo intermedio $\to$ foglia appropriata.
- Per ricerche di intervallo (es. $K > 20$), troviamo il primo valore e poi scorriamo sequenzialmente le foglie.

15.4.6 Operazioni sugli Alberi B+: Esercizi

Ricerca di Uguaglianza

Obiettivo: Trovare una chiave specifica (es. K=40).

1. **Radice:** Parti dalla radice. Confronta la chiave di ricerca con le chiavi nel nodo radice per determinare quale puntatore seguire.
2. **Nodi Interni:** Continua a scendere nell'albero, seguendo il puntatore appropriato in ogni nodo interno in base all'intervallo di chiave.
3. **Nodo Foglia:** Raggiungi un nodo foglia. Cerca la chiave di ricerca all'interno di questo nodo foglia.
4. **Conclusione:** Se la chiave viene trovata nel nodo foglia, segui il suo puntatore per recuperare il record. Se non viene trovata, il record non esiste nell'indice.

Descrizione del percorso di ricerca: Si parte dalla radice e si segue una traiettoria dall'alto verso il basso attraverso i nodi interni fino a raggiungere la foglia che potrebbe contenere la chiave.

Ricerca per Intervallo

Obiettivo: Trovare tutte le chiavi in un dato intervallo (es. K > 40).

1. **Trova il primo record possibile:** Esegui una ricerca di uguaglianza per il limite inferiore dell'intervallo (es. 40). Questo ti porterà al nodo foglia che *dovrebbe* contenere il primo valore dell'intervallo.
2. **Identifica il primo valore nell'intervallo:** Esamina le chiavi nel nodo foglia trovato per identificare la prima chiave che soddisfa il criterio dell'intervallo (es. la prima chiave > 40).
3. **Recupera i valori nell'intervallo:** Per la prima chiave trovata e per tutte le chiavi successive nel nodo foglia che sono ancora nell'intervallo, segui i puntatori associati per recuperare i record di dati.
4. **Scorri foglie:** Se l'intervallo si estende oltre il nodo foglia corrente, segui il puntatore "prossimo nodo foglia" per spostarti al nodo foglia adiacente. Continua a elaborare le chiavi in questo nuovo nodo foglia e a spostarti ai successivi finché non esci dall'intervallo o raggiungi la fine della lista di foglie.
5. **Risultato:** Sono stati recuperati tutti i record con chiavi nell'intervallo specificato.

Descrizione del percorso di ricerca: Si esegue una ricerca per trovare la foglia di partenza, e poi si scorrono i nodi foglia collegati finché non si esce dall'intervallo di ricerca.

Inserimento in Alberi B+

Principio: Trova il nodo foglia corretto per la chiave, inserisci la chiave e il puntatore al record. Se il nodo foglia è pieno, dividilo (split) e sposta la chiave mediana (copiata) al nodo padre. Se il nodo padre è pieno, anche lui si divide e propaga una chiave al suo padre, ricorsivamente. Se la radice si divide, si crea una nuova radice e l'altezza dell'albero aumenta.

Descrizione dell'operazione di inserimento: Si individua la foglia corretta. Se c'è spazio, si inserisce. Se la foglia è piena, si divide in due, e una chiave viene copiata nel padre. Se il padre è pieno, si divide a sua volta, spostando una chiave al suo padre, e così via verso l'alto.

Cancellazione da Alberi B+

Principio: Trova la chiave nel nodo foglia e cancella la chiave e il puntatore al record. Se il nodo foglia scende sotto il numero minimo di chiavi/puntatori (underflow): - Prova a *prestare* una chiave dal nodo fratello adiacente che ha chiavi in eccesso. Se presti, aggiorna la chiave nel nodo padre che separa i due nodi. - Se il fratello non può prestare, *fondi* il nodo con il fratello. Durante la fusione, la chiave nel nodo padre che separava i due nodi fusi viene rimossa (se il nodo era interno) o tirata giù (se il nodo era foglia) e incorporata nel nodo fuso. Se un nodo interno va in underflow a seguito di un prestito o fusione dei suoi figli, il processo di rebalancing (prestito/fusione) si propaga al livello superiore. Se la radice scende sotto il minimo (e non è una foglia), l'altezza dell'albero si riduce.

Descrizione dell'operazione di cancellazione: Si individua e rimuove la chiave dalla foglia. Se la foglia va in underflow, si tenta di bilanciare con un fratello (prestito o fusione). Questo può causare underflow o necessità di bilanciamento nei nodi superiori, propagandosi fino alla radice.

15.4.7 Variazione dell'Altezza

- **Inserimento:** L'altezza aumenta solo se la radice corrente si divide. Una nuova radice viene creata sopra la radice divisa.
- **Cancellazione:** L'altezza diminuisce solo se la radice (che non è una foglia) rimane con un solo figlio a seguito di una fusione propagata, e quel figlio diventa la nuova radice.

Esempio di riduzione altezza per cancellazione: Se una fusione a livello inferiore porta la radice non-foglia ad avere un solo figlio, quel figlio diventa la nuova radice. Esempio di aumento altezza per inserimento: Se una radice foglia piena si divide, viene creata una nuova radice interna.

15.5 Complessità della Ricerca con Alberi B+

La complessità di una ricerca (uguaglianza o intervallo per trovare il primo elemento) in un albero B+ è proporzionale all'altezza L dell'albero. Ogni accesso a un nodo (blocco) tipicamente richiede un accesso a disco.

Assumiamo N_{leaves} è il numero totale di nodi foglia nell'albero. Assumiamo un fattore di ramificazione (fanout) medio m per i nodi interni. m è approssimativamente la metà del numero massimo di puntatori ($n + 1$). L'altezza L è approssimativamente data da:

$$L \approx \log_m(N_{\text{leaves}}) + 1$$

Il "+1" è per la radice.

Il numero di accessi a disco per una ricerca di uguaglianza è L. Per una ricerca per intervallo, è L per trovare il primo elemento, più il numero di blocchi foglia da scorrere.

15.5.1 Esempio di Complessità della Ricerca

1. **Calcolo di n (già fatto):** Dato n, si sa la capacità massima dei blocchi.
2. **Assunzioni:** Fanout medio m. Si stima un valore realistico per m basato su n e sul riempimento minimo dei nodi.
3. **Albero B+ di L livelli:** Si calcola il numero massimo (o stimato) di nodi a ogni livello e il numero totale di record indirizzabili. Con m elevato, un albero B+ anche di pochi livelli può indirizzare un numero molto grande di record.
4. **Memoria per i primi livelli:** I livelli superiori dell'albero (radice e pochi livelli sotto) sono relativamente piccoli e possono spesso risiedere nella memoria RAM del sistema di database (buffer pool).
5. **Performance di Ricerca:** Se i livelli superiori sono in RAM, la ricerca richiede solo accessi a disco per i nodi foglia e i blocchi dati puntati. Il numero di accessi a disco è quindi ridotto e dipende dall'altezza dell'albero.

15.6 Strategie Generali per gli Esercizi sugli Alberi B+

1. **Capire n:** Calcola o ti viene dato il valore di n e i requisiti minimi per i nodi.
2. **Distinguere Nodi Interni e Foglie:** Ricorda la loro struttura e funzione differente.
3. **Ricerca:** Applica la ricerca dall'alto verso il basso per trovare la foglia o il punto di partenza per un intervallo.
4. **Inserimento:** Individua la foglia. Inserisci. Gestisci gli split e la propagazione verso l'alto. Considera l'aumento dell'altezza.
5. **Cancellazione:** Individua la chiave nella foglia. Cancella. Gestisci l'underflow con prestito o fusione. Propaga il rebalancing verso l'alto. Considera la riduzione dell'altezza.
6. **Disegna l'albero:** Per esercizi che richiedono di mostrare lo stato dell'albero, disegnare l'albero passo passo è cruciale.

Capitolo 16

Laboratorio 7: Hash Table e Indici Invertiti

16.1 Indici basati su Hash

Gli indici basati su hash mappano una chiave di ricerca direttamente all'identificatore di pagina (pid) della pagina (o catena di overflow) che la contiene.

Caratteristiche principali:

- **Ottimi per ricerche di uguaglianza (equality search):** Es. `WHERE nome = 'Mario'`.
- **Non efficienti per ricerche su intervalli (range search):** Es. `WHERE eta > 30`.
- Usano **blocchi** per memorizzare i **bucket**.

Esistono due macro-categorie:

1. **Hashing Statico:** Per dati di dimensione fissa e non modificabili (es. CD-ROM).
2. **Hashing Dinamico (Estensibile e Lineare):** Quando dati e dimensioni dei dati possono variare nel tempo.

16.1.1 Hashing Statico

L'hashing statico utilizza un numero fisso N di bucket e una funzione di hash H che mappa la chiave di ricerca in un intervallo da 0 a $N-1$. Negli esempi, useremo funzioni di hash H_i che restituiscono i primi (o gli ultimi) i bit della codifica binaria della chiave.

Esempio Base:

- $i = 1$ (si usa il primo bit più significativo).
- $N = 2^i = 2^1 = 2$ bucket (bucket 0 e bucket 1).
- Ogni bucket contiene un blocco.
- Chiavi (già codificate in binario): `0001`, `1001`, `1100`.
  - $H_1(0001) \rightarrow$ primo bit è `0` $\rightarrow$ bucket 0.
  - $H_1(1001) \rightarrow$ primo bit è `1` $\rightarrow$ bucket 1.
  - $H_1(1100) \rightarrow$ primo bit è `1` $\rightarrow$ bucket 1.

Struttura risultante:

- Bucket 0: [`0001`]
- Bucket 1: [`1001`, `1100`]

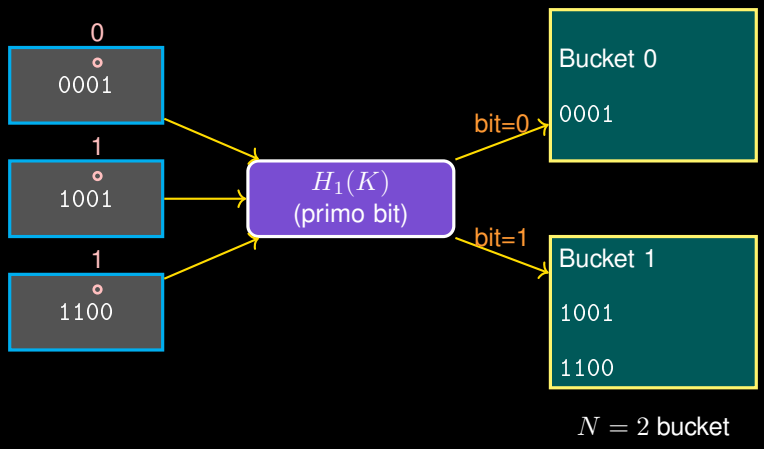


Figura 16.1: Visualizzazione dell'hashing statico con H_1 (primo bit)

Ricerca (Searching)

Si calcola l'hash della chiave data per trovare il bucket e si cerca il record all'interno di quel bucket (e delle sue eventuali catene di overflow).

Esempio:

- Cercare $K = 1100$.
- Funzione Hash: H_1 (usa il primo bit).
- $H_1(1100) \rightarrow$ il primo bit è 1.
- Si accede al bucket 1.
- Si scorre il blocco (o i blocchi) del bucket 1 fino a trovare (o non trovare) 1100.

Inserimento (Insertion)

L'hashing statico non può cambiare il numero di bucket. Se un bucket è pieno, si usa una **catena di blocchi di overflow**.

Esempio:

- Stato attuale come sopra. Inserire $K = 1010$.
- Funzione Hash: H_1.
- $H_1(1010) \rightarrow$ il primo bit è 1.
- Si accede al bucket 1.
- Supponiamo che il blocco primario del bucket 1 sia pieno (contenente 1001, 1100).
- Si crea un blocco di overflow collegato al blocco primario del bucket 1 e vi si inserisce 1010.

Struttura dopo inserimento di 1010:

- Bucket 0: [0001]
- Bucket 1: [1001, 1100] $\rightarrow$ [1010] (overflow)

Domande Tipiche: Data la struttura risultante dall'inserimento di 1010:

- **Quanti bucket ci sono?** Ci sono **2** bucket (bucket 0 e bucket 1). Il numero di bucket è fisso.
- **Quanti blocchi ci sono?**
  - Bucket 0: 1 blocco.
  - Bucket 1: 1 blocco primario + 1 blocco di overflow.
  - Totale: **3** blocchi.

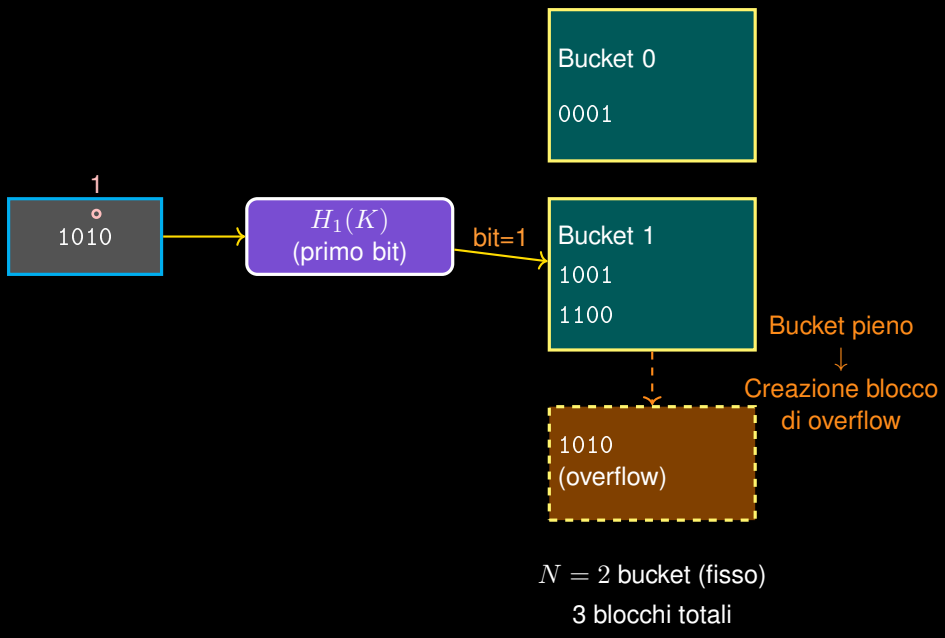


Figura 16.2: Inserimento con overflow nell'hashing statico: il bucket 1 è pieno, quindi 1010 viene inserito in un blocco di overflow

Cancellazione (Deletion)

Si individua il record e lo si rimuove. Se la rimozione svuota un blocco di overflow, questo può essere deallocato. Lunghe catene di overflow degradano le performance.

Esempio:

- Stato attuale: Bucket 0: [0001]; Bucket 1: [1001, 1100] $\rightarrow$ [1010].
- Cancellare $K = 1100$.
  1. $H_1(1100) \rightarrow$ bucket 1.
  2. Si cerca 1100 nel bucket 1 e lo si rimuove dal suo blocco (il blocco primario).
- Se successivamente si cancellasse $K = 1010$:
  1. $H_1(1010) \rightarrow$ bucket 1.
  2. Si cerca 1010 nel bucket 1, trovandolo nel blocco di overflow.
  3. Si rimuove 1010. Se il blocco di overflow diventa vuoto, può essere deallocato e il puntatore dal blocco primario rimosso.

Efficienza dell'Hashing Statico

- **Ideale:** Lookup con un solo accesso al disco se il bucket sta in un blocco e non ci sono overflow.
- **Problema:** Lunghe catene di overflow degradano rapidamente le prestazioni (un accesso al disco per blocco).
- L'efficienza dipende da:
  - Rapporto tra dimensione dell'indice e dati (numero di bucket).
  - Distribuzione delle chiavi rispetto alla funzione hash.

Per mitigare i problemi di overflow, si usano tecniche di hashing dinamico come l'hashing estensibile o lineare.

16.1.2 Hashing Estensibile (Extendible Hashing)

Introduce un livello di indirezione: una **directory di puntatori** ai blocchi.

Caratteristiche principali:

- La directory dei puntatori può crescere; la sua lunghezza è sempre una potenza di 2.
- Raddoppiando la directory, raddoppia il numero di "bucket" (entry nella directory).
- Non è necessario un blocco dati per ogni bucket della directory; più bucket possono condividere un blocco.
- **Non usa blocchi di overflow.**
- La funzione hash H_i restituisce i primi i bit più significativi. i è la **profondità globale** della directory.
- Ogni blocco ha una variabile j (chiamata **profondità locale**) che indica quanti bit sono stati usati per l'indicizzazione *di quel blocco*.

Terminologia:

- i: profondità globale (numero di bit usati per indirizzare la directory). Dimensione directory = 2^i.
- j: profondità locale (numero di bit usati per discriminare i record *all'interno* di un blocco specifico). $j \leq i$.

Ricerca

1. Calcola $H_i(K)$ (primi i bit della chiave K).
2. Usa questo valore come indice nella directory per trovare il puntatore al blocco corretto.
3. Accedi al blocco e cerca K.

Esempio:

- $i = 1$ (profondità globale). Directory ha $2^1 = 2$ entry (0 e 1).
- Blocco A (per chiavi che iniziano con 0...): contiene 0001. Profondità locale $j_A = 1$.
- Blocco B (per chiavi che iniziano con 1...): contiene 1001, 1100. Profondità locale $j_B = 1$.
- Directory:
  - 0 → Blocco A
  - 1 → Blocco B
- Cercare $K = 1100$:
  1. $H_1(1100)$ (primo bit) → 1.
  2. Directory[1] punta al Blocco B.
  3. Cerca 1100 nel Blocco B. Trovato.

Inserimento (Insertion Steps)

1. **Trova il blocco:** Usa i primi i bit della chiave K ($H_i(K)$) per trovare l'entry nella directory e quindi il blocco B.
2. **Se c'è spazio nel blocco B:** Inserisci il record. Fatto.
3. **Se non c'è spazio nel blocco B:** Controlla la profondità locale j del blocco B.
   - **Caso A:** $j < i$ (Il blocco B è condiviso da più entry della directory che differiscono solo dopo il j-esimo bit)
     (a) **Split del blocco B:** Crea un nuovo blocco B'.

(b) **Incrementa j:** La profondità locale di B e B' diventa $j + 1$.

(c) **Distribuisci i record:** Riassegna i record del vecchio B (più il nuovo record da inserire) tra B e B' basandoti sul valore del loro $(j + 1)$-esimo bit.

(d) **Aggiorna la directory:** Alcune entry della directory che prima puntavano a B ora punteranno a B'. In particolare, quelle entry che corrispondono ai record finiti in B' (cioè quelle il cui prefisso di i bit, quando si considerano $j + 1$ bit, matcha i record in B').

- **Caso B:** $j == i$ (Il blocco B è "pieno" rispetto alla capacità discriminante della directory attuale)

(a) **Incrementa i:** La profondità globale della directory diventa $i + 1$.

(b) **Raddoppia la directory:** Ogni vecchia entry w (lunga i bit) nella directory genera due nuove entry $w0$ e $w1$ (lunghe $i + 1$ bit). Inizialmente, entrambe puntano al blocco a cui puntava w.

(c) **Ora $j < i$ (nuovo i):** Procedi come nel Caso A per splittare il blocco B (la cui profondità locale j è ora minore del nuovo i). La j dei due blocchi risultanti diventerà $j_{\text{vecchio}} + 1$.

Esempio Inserimento (inserire 1010)

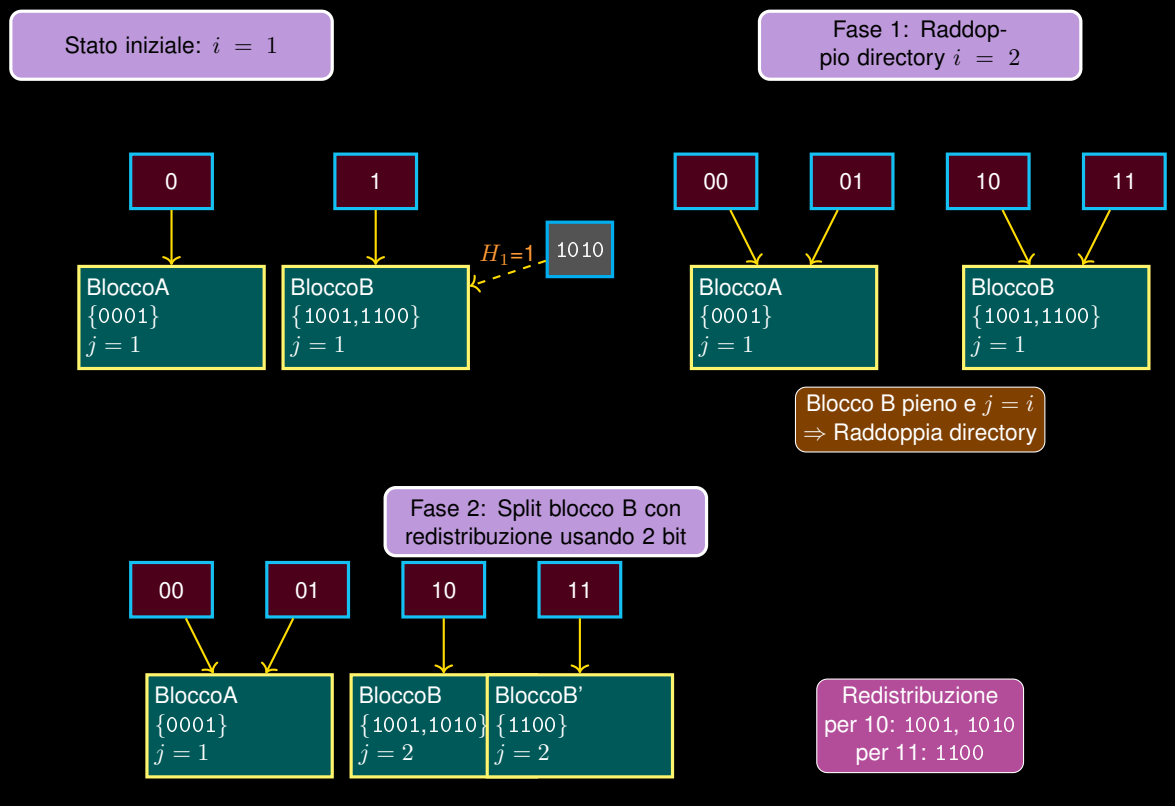


Figura 16.3: Visualizzazione dell'inserimento di 1010 nell'hashing estensibile

Domande Tipiche:

- $i = 3$.
- Directory: 000, 001, 010, 011, 100, 101, 110, 111.
- **Quanti bucket (entry della directory) ci sono?** $2^i = 2^3 = \mathbf{8}$.
- **Quanti blocchi ci sono?** Si contano i blocchi distinti puntati dalla directory (esempio):

1. Blocco_X (contiene 0001, j=2) puntato da 000 e 001.
2. Blocco_Y (contiene 0100, 0101, j=2) puntato da 010 e 011.
3. Blocco_Z (contiene 1001, 1000, j=3) puntato da 100.

4. `Blocco_W (contiene 1010, j=3)` puntato da `101`.
5. `Blocco_K (contiene 1100, j=2)` puntato da `110` e `111`.

Totale: **5** blocchi distinti (in questo esempio).

Osservazioni sull'Hashing Estensibile

- **Ricerca veloce:** Un solo accesso al disco se la directory è in memoria principale.
- **Split localizzati:** Solo i record di un blocco vengono ridistribuiti.
- **Svantaggi:**
  - Se la ridistribuzione dopo uno split è sfortunata (es. tutti i record hanno gli stessi primi $j+1$ bit), potrebbe essere necessario splittare di nuovo subito.
  - La directory cresce a scatti (raddoppia). Con distribuzioni di chiavi sbilanciate, la directory può diventare inutilmente grande.
  - La crescita esponenziale della directory può complicarne la gestione e la memorizzazione in memoria principale.

16.1.3 Hashing Lineare

Evita la necessità di una directory, ma gestisce il problema delle lunghe catene di overflow.

Caratteristiche principali:

- Il numero di bucket n cresce **uno alla volta**.
- I **blocchi di overflow sono permessi**.
- L'aumento dei bucket avviene linearmente quando si crea un blocco di overflow o si supera una soglia di **load factor** (rapporto record/bucket).
  - Esempio di policy: mantenere non più di $1.7 \times n$ record nel file.
- La funzione hash $H_i(K)$ restituisce gli i bit **meno significativi**.
- n: numero attuale di bucket.
- i: numero di bit usati dalla funzione hash H_i. $2^{i-1} < n \leq 2^i$.
- s: (split pointer, non sempre esplicito nelle slide ma concettuale) indica il prossimo bucket da splittare in sequenza. Inizia da 0 e arriva a $2^{i-1} - 1$. Quando tutti i bucket del livello $i-1$ sono stati splittati, i aumenta.

Ricerca (Searching)

1. Sia n_{buck} il numero corrente di bucket e i il numero di bit LSB usati dalla funzione hash H_i tale che $2^{i-1} < n_{buck} \leq 2^i$.
2. Calcola $m = H_i(K)$ (ultimi i bit della chiave K).
3. **Se** $m < n_{buck}$**:** La chiave si trova nel bucket m (o nella sua catena di overflow).
4. **Se** $m \geq n_{buck}$**:** La chiave si trova nel bucket $(m - 2^{i-1})$ (o nella sua catena di overflow). Questo accade perché $H_i(K)$ ha mappato a un bucket che "logicamente" esisterebbe se tutti i 2^i bucket fossero già stati creati, ma fisicamente non è ancora stato splittato per esistere a quel livello. Quindi, il record deve ancora trovarsi nel suo bucket "antenato" del livello $i-1$.

Esempio 1:

- $i = 2$, $n_{buck} = 3$ (bucket 00, 01, 10), $r = 4$ (record). $r/n_{buck} = 1.33$.
- Chiavi: B00={0000}, B01={1111, 0101}, B10={1010}.

- Cercare $K = 0101$:
  1. $H_2(0101)$ (ultimi 2 bit) $\rightarrow$ 01 (decimale 1). Quindi $m = 1$.
  2. $m < n_{buck}$? $(1 < 3) \rightarrow$ Sì.
  3. La chiave è nel bucket 01.

Esempio 2:

- Stessi $i = 2$, $n_{buck} = 3$, $r = 4$.
- Cercare $K = 1111$:
  1. $H_2(1111)$ (ultimi 2 bit) $\rightarrow$ 11 (decimale 3). Quindi $m = 3$.
  2. $m < n_{buck}$? $(3 < 3) \rightarrow$ No.
  3. Quindi $m \geq n_{buck}$ $(3 \geq 3) \rightarrow$ Sì.
  4. La chiave è nel bucket $(m - 2^{i-1}) = (3 - 2^{2-1}) = (3 - 2^1) = 3 - 2 = 1$. Bucket 01.

Inserimento (Insertion Steps)

1. **Individua il bucket:** Usa la logica di ricerca per trovare il bucket b per la chiave K.
2. **Inserisci:** Inserisci K nel bucket b. Se b è pieno, crea/usa un blocco di overflow.
3. **Aggiorna r:** Incrementa il conteggio dei record r.
4. **Controlla il load factor:** Se $r/n_{buck} > 1.7$ (o la soglia scelta):
   (a) **Split necessario (Load Balancing):**
   - **Aumentare i?** Se $n_{buck} = 2^i$ (cioè tutti i bucket del livello i sono stati creati e il prossimo split farebbe superare 2^i bucket), allora incrementa i a $i + 1$. Lo "split pointer" s si resetta a 0.
   - **Aggiungi nuovo bucket:** Si aggiunge fisicamente il bucket n_{buck}-esimo (che avrà indirizzo n_{buck} in binario).
   - **Identifica bucket da splittare:** Lo split avviene sempre in ordine. Se lo split pointer s indica il bucket s_{addr}, i record vengono spostati da s_{addr} al nuovo bucket n_{buck}. Se il bucket aggiunto è $1a_2a_3...a_k$ (nuovo n_{buck}), allora splitta il bucket $0a_2a_3...a_k$. Sposta nel nuovo bucket $1a_2a_3...a_k$ tutti i record dal bucket $0a_2a_3...a_k$ che hanno il k-esimo bit meno significativo uguale a 1 (quando si considera la funzione H_k o H_{i+1} con il nuovo i).
   - **Incrementa n_{buck}:** n_{buck} diventa $n_{buck} + 1$.
   - **Avanza split pointer s:** s diventa $s + 1$.

Esempio Inserimento Dettagliato (inserire 0101)
Stato iniziale:

- $i = 1$, $n_{buck} = 2$ (bucket 0, 1), $r = 3$. $r/n_{buck} = 1.5$.
- Bucket 0: {0000, 1010}, Bucket 1: {1111}.
- (Split pointer s è concettualmente 0).

Inserire $K = 0101$:

1. **Individua bucket:** $H_1(0101)$ (LSB) $\rightarrow$ 1. $m = 1$. $m < n_{buck}$ $(1 < 2) \rightarrow$ Sì. Bucket 1.
2. **Inserisci:** Bucket 1 ora {1111, 0101}.
3. **Aggiorna r:** r diventa 4.
4. **Controlla load factor:** $r/n_{buck} = 4/2 = 2$. $2 > 1.7 \rightarrow$ Sì, split.

Load Balancing:

1. **Aumentare i?** $n_{buck} = 2^i$? $(2 = 2^1) \rightarrow$ Sì. Incrementa i a 2. (Split pointer s si resetta concettualmente a 0 per il nuovo i).

2. **Aggiungi nuovo bucket:** Il nuovo n_{buck} (prima dello split era 2) identifica il bucket 10 (binario di 2). Questo sarà il nuovo bucket aggiunto.
3. **Split:**
   - Il bucket aggiunto è $n_{buck} = 2$ (binario 10). Qui $k = 2$ (perché i è diventato 2). $1a_2...a_k \rightarrow 10$. Quindi $a_2...a_k$ è 0.
   - Splitta il bucket $0a_2...a_k \rightarrow 00$.
   - Sposta i record dal bucket 00 al bucket 10 se il loro i-esimo (2°) LSB è 1 (cioè se finiscono in $X1$ con H_2).
     - Record in bucket 00: {0000 (LSBs 00), 1010 (LSBs 10)}.
     - 0000: LSBs 00. Resta in 00.
     - 1010: LSBs 10. Il 2° LSB (contando da destra, il più significativo dei due) è 1. Spostato a 10.
   - Bucket 00 ora {0000}. Bucket 10 ora {1010}.
4. **Incrementa n_{buck}:** n_{buck} diventa 3. (Split pointer s avanza a 1).

Stato finale: $i = 2$, $n_{buck} = 3$, $r = 4$. r/n = 1.33.

- Bucket 00: {0000}
- Bucket 01: {1111, 0101}
- Bucket 10: {1010}

Domande Tipiche: Riferendosi a uno stato dopo vari inserimenti e split:

- $i = 2$, $n_{buck} = 3$ (bucket 00, 01, 10), $r = 5$.
- Struttura (esempio):
  - B00: {0000}
  - B01: {1111, 0101} $\rightarrow$ overflow {0001}
  - B10: {1010}
- **Quanti bucket ci sono?** $n_{buck} =$ **3**. (Bucket 00, 01, 10).
- **Quanti blocchi ci sono?**
  1. Blocco per B00 (contiene 0000).
  2. Blocco primario per B01 (contiene 1111, 0101).
  3. Blocco di overflow per B01 (contiene 0001).
  4. Blocco per B10 (contiene 1010).

  Totale: **4** blocchi (in questo esempio).

16.2 Riepilogo Tecniche di Hashing Viste Finora

16.2.1 Concetti Fondamentali dell'Hashing (Comuni a Tutti)

- **Scopo:** Mappare una chiave di ricerca (es. un nome, un ID) direttamente a una pagina o a un blocco di memoria dove si trova il dato, per un recupero veloce.
- **Funzione Hash (H):** Trasforma la chiave in un indirizzo di bucket.
- **Bucket:** Un contenitore (spesso un blocco su disco) che memorizza i record.
- **Collisioni:** Quando due chiavi diverse vengono mappate allo stesso bucket.
- **Overflow:** Quando un bucket è pieno e i nuovi record devono essere memorizzati altrove (es. blocchi di overflow).

16.2.2 Hashing Statico

- **Caratteristica Chiave:** Il numero di bucket (N) è **fisso** e deciso all'inizio. Non cambia.
- **Funzione Hash Semplice:** Spesso $H(K) = K \pmod N$, o, come negli esempi, si usano i primi/ultimi bit della chiave.
- **Inserimento:**
  - Calcola l'hash della chiave per trovare il bucket.
  - Se il bucket ha spazio, inserisci.
  - **Se il bucket è pieno:** Si usa una **catena di blocchi di overflow** collegati al bucket primario. Il numero di bucket *non* aumenta.
- **Ricerca:**
  - Calcola l'hash della chiave per trovare il bucket.
  - Cerca nel bucket e, se necessario, nella sua catena di overflow.
- **Cancellazione:**
  - Trova il record e rimuovilo.
  - Se un blocco di overflow diventa vuoto, può essere deallocato.
- **Efficienza:**
  - **Ottimo** se non ci sono overflow (un accesso al disco).
  - **Peggiora** con lunghe catene di overflow (un accesso per ogni blocco della catena).
- **Domande Tipiche d'Esame:**
  - Dato un insieme di chiavi e una funzione hash, mostrare la struttura finale.
  - Contare il numero di bucket (è sempre N).
  - Contare il numero totale di blocchi (bucket primari + blocchi di overflow).
  - Descrivere i passaggi per inserire/cercare una chiave.

16.2.3 Hashing Estensibile (Extendible Hashing)

- **Scopo:** Superare il limite dei bucket fissi dell'hashing statico, gestendo dinamicamente la crescita dei dati **senza usare blocchi di overflow tradizionali.**
- **Componenti Chiave:**
  - **Directory:** Un array di puntatori ai blocchi di dati. La sua dimensione è sempre una potenza di 2 (2^i).
  - **Profondità Globale (i):** Numero di bit della chiave usati per indicizzare la directory. Determina la dimensione della directory.
  - **Blocchi Dati:** Contengono i record.
  - **Profondità Locale (j):** Associata a *ciascun blocco dati*. Indica quanti bit sono significativi per i record *in quel blocco*. $j \leq i$. Più entry della directory possono puntare allo stesso blocco se la loro profondità locale j è minore di i.
- **Ricerca:**
  1. Prendi i primi i bit della chiave.
  2. Usa questi bit come indice nella directory per trovare il puntatore al blocco.
  3. Cerca nel blocco.
- **Inserimento (Concetti Chiave):**

1. Trova il blocco corretto usando i primi i bit della chiave.
2. **Se c'è spazio nel blocco:** Inserisci. Fatto.
3. **Se il blocco è pieno:**
   - **Caso A: Profondità locale del blocco $j < i$ (Profondità globale):**
     (a) **Split del blocco:** Crea un nuovo blocco.
     (b) Incrementa la profondità locale j dei due blocchi (il vecchio e il nuovo) a $j + 1$.
     (c) **Ridistribuisci i record** del vecchio blocco (più il nuovo record) tra il vecchio e il nuovo blocco, basandoti sul $(j + 1)$-esimo bit.
     (d) **Aggiorna i puntatori nella directory:** Alcune entry che puntavano al vecchio blocco ora punteranno al nuovo blocco.
   - **Caso B: Profondità locale del blocco $j = i$ (Profondità globale):**
     (a) **Raddoppia la directory:** Incrementa la profondità globale i a $i + 1$. Ogni vecchia entry w genera due nuove entry $w0$ e $w1$, che inizialmente puntano allo stesso blocco a cui puntava w.
     (b) **Ora $j < i$ (nuovo i):** Procedi come nel Caso A per splittare il blocco che era pieno (la sua profondità locale j originale è ora minore del nuovo i). La j dei due blocchi risultanti diventerà $j_{\text{vecchio}} + 1$.

- **Vantaggi:**
  - Ricerca veloce (spesso un accesso se la directory è in RAM).
  - Non usa catene di overflow dirette.
- **Svantaggi:**
  - La directory può raddoppiare di dimensione, potendo diventare molto grande.
  - Split sfortunati possono richiedere split multipli.
- **Domande Tipiche d'Esame:**
  - Mostrare la struttura (directory e blocchi) dopo una serie di inserimenti.
  - Indicare i valori di i (profondità globale) e j (profondità locale per ogni blocco).
  - Contare il numero di "bucket" (entry nella directory, cioè 2^i).
  - Contare il numero di blocchi dati fisici (distinti).
  - Spiegare quando e come la directory raddoppia e quando un blocco splitta.

16.2.4 Hashing Lineare

- **Scopo:** Gestire la crescita dei dati in modo più graduale rispetto all'hashing estensibile, permettendo l'uso di **blocchi di overflow**.
- **Componenti Chiave:**
  - **Numero di Bucket (n o n_{buck}):** Cresce **uno alla volta**. Non è necessariamente una potenza di 2.
  - **Livello (i):** Numero di bit (di solito i **meno significativi**, LSB) usati dalla funzione hash principale H_i. Vale $2^{i-1} < n \leq 2^i$.
  - **Split Pointer (s o `next`):** Indica il prossimo bucket (del livello $i - 1$) che deve essere splittato. Va da 0 a $2^{i-1} - 1$.
  - **Load Factor:** Rapporto tra numero di record (r) e numero di bucket (n). Uno split avviene quando questo supera una soglia (es. 1.7).
  - **Funzioni Hash:** Si usano due funzioni hash concettualmente: H_i (usa i bit LSB) e H_{i+1} (usa $i + 1$ bit LSB durante gli split).
- **Ricerca:**
  1. Calcola $m = H_i(K)$ (ultimi i bit della chiave).

2. **Se $m < n$ (numero attuale di bucket):** La chiave è nel bucket m (o suoi overflow).
3. **Se $m \geq n$:** La chiave è nel bucket $m - 2^{i-1}$ (o suoi overflow). Questo bucket $m - 2^{i-1}$ è il bucket "antenato" che non è stato ancora splittato a questo round per diventare il bucket m.

- **Inserimento:**
  1. Trova il bucket corretto usando la logica di ricerca.
  2. Inserisci la chiave. Se il bucket è pieno, usa un blocco di overflow.
  3. Aggiorna il conteggio dei record r.
  4. **Controlla il Load Factor (r/n):** Se supera la soglia:
     - **Aggiungi un nuovo bucket:** Il bucket n-esimo viene aggiunto fisicamente. n si incrementa.
     - **Splitta il bucket puntato da s:** I record nel bucket s vengono ridistribuiti tra il bucket s e il nuovo bucket $s + 2^{i-1}$ (che è il bucket appena aggiunto, se s è il suo "antenato"). La ridistribuzione usa H_{i+1} (o comunque $i + 1$ bit).
     - **Avanza lo split pointer s:** $s \leftarrow s + 1$.
     - **Se $s = 2^{i-1}$ (ha "completato un giro"):** Significa che tutti i bucket del livello $i - 1$ sono stati splittati. Resetta $s = 0$ e incrementa il livello $i \leftarrow i + 1$.
- **Vantaggi:**
  - Espansione graduale, un bucket alla volta.
  - Non richiede una directory separata enorme.
- **Svantaggi:**
  - L'uso di blocchi di overflow può comunque degradare le prestazioni.
  - La logica di ricerca è un po' più complessa.
- **Domande Tipiche d'Esame:**
  - Mostrare la struttura (bucket, overflow, valori di n, i, s, r) dopo una serie di inserimenti.
  - Determinare in quale bucket cercare una chiave.
  - Spiegare quando avviene uno split e quale bucket viene splittato.
  - Contare il numero di bucket (n).
  - Contare il numero di blocchi fisici (bucket primari + blocchi di overflow).

16.2.5 Suggerimenti Generali per gli Esercizi

- **Disegna Sempre:** Visualizzare la struttura (bucket, directory, puntatori, overflow) è fondamentale.
- **Tieni Traccia dei Parametri:** Annota i valori di i, j, n, s, r e come cambiano.
- **Funzione Hash:** Presta attenzione a quanti e quali bit (primi/ultimi) usa la funzione hash.
- **Capacità dei Blocchi:** Gli esercizi di solito specificano quanti record può contenere un blocco.

Esempi Pratici di Risoluzione

Parametri	Valore	Note
Funzione hash	$H_1(K)$	Primo bit
Numero bucket	$N = 2^1 = 2$	Fisso per hashing statico
Capacità blocco	2 chiavi/blocco	

Tabella 16.1: Parametri iniziali per l'esempio di hashing statico

Esempio di Hashing Statico

Domanda	Risposta
Quanti bucket?	2 (fissi)
Quanti blocchi?	3 (2 primari + 1 overflow)

Chiave	Binario	Hash	Bucket	Note
K1	0001	0	0	Inserimento semplice
K2	1001	1	1	Inserimento semplice
K3	1100	1	1	Inserimento semplice
K4	1010	1	1	Overflow: bucket 1 pieno

Tabella 16.2: Tracciamento inserimenti nell'hashing statico

Parametro	Valore Iniziale	Dopo Split	Note
Profondità globale (i)	1	2	Raddoppiata dopo split
Profondità locale (j_A)	1	1	Blocco A non splittato
Profondità locale (j_B)	1	2	Blocco B splittato$\rightarrow$B,B'

Tabella 16.3: Evoluzione dei parametri nell'hashing estensibile

Esempio di Hashing Estensibile

Domanda	Risposta
Quanti bucket?	$2^i = 2^2 = 4$ (entry nella directory)
Quanti blocchi?	3 (A, B, B')

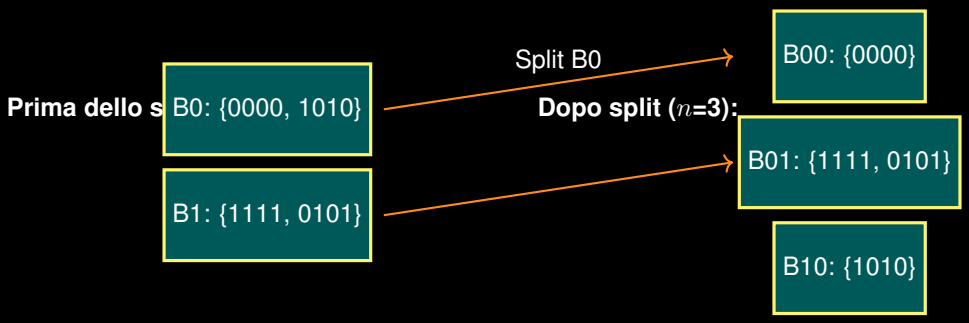


Figura 16.4: Visualizzazione dello split nell'hashing lineare

Esempio di Hashing Lineare

Domanda	Risposta
Quanti bucket?	3 (dopo split)
Quanti blocchi?	3 (ogni bucket ha un blocco)

Tabella Riassuntiva di Confronto

16.3 Recupero di Documenti (Document Retrieval) e Indici Invertiti

L'Information Retrieval (IR) si occupa di trovare materiale (solitamente documenti testuali non strutturati) che soddisfi un bisogno informativo all'interno di grandi collezioni. Il recupero di documenti basato su parole chiave è un problema classico.

16.3.1 Indici Invertiti (Inverted Indexes)

Sono usati per recuperare efficientemente documenti tramite query full-text. Due tipi di query principali:

1. Trovare tutti i documenti che contengono un **dato insieme di parole chiave** (es. "Brutus" AND "Caesar"). L'ordine non conta.
2. Trovare tutti i documenti che contengono una **data sequenza di parole chiave** (es. la frase "a clear"). L'ordine conta.

Directory	Stato		
	Iniziale (i=1)	Dopo Raddoppio	Dopo Split
00	-	Blocco A	Blocco A
01	-	Blocco A	Blocco A
10	-	Blocco B	Blocco B
11	-	Blocco B	Blocco B'

Tabella 16.4: Evoluzione della directory nell'hashing estensibile

Parametro	Valore Iniziale	Dopo Split	Note
Livello (i)	1	2	Aumentato al completare un round
Numero bucket (n)	2	3	Incrementato di 1
Split pointer (s)	0	1	Avanzato dopo ogni split
Numero record (r)	3	4	Dopo inserimento K4

Tabella 16.5: Evoluzione dei parametri nell'hashing lineare

Un indice invertito, invece di creare un indice per ogni "attributo" (parola), rappresenta per ogni parola specifica (termine) l'elenco di tutti i documenti in cui appare.

16.3.2 Costruzione di un Indice Invertito

1. **Tokenizzazione:** Dividi ogni documento in parole (token). Registra la posizione di ogni token.
   - D1: "Friends, Romans, countrymen, lend(4) me your ears; I come to(10) bury Caesar(12), not to(14) praise him..." (numeri sono posizioni indicative)
   - D2: "In a(2) broad valley(4), at the foot of a(9) sloping hillside, beside a(13) clear(14) bubbling stream, Tom(17) was building..."
   - D3: "I(1) did enact Julius Caesar(5): I(6) was killed i' the Capitol; Brutus(12) killed me. It was a brute part of him(21) to kill so capital a..."
2. **Creazione del Dizionario (Vocabulary) e delle Posting List:** Il dizionario contiene tutti i termini unici. Per ogni termine, la posting list memorizza `(DocumentID, posizione)` o `(DocumentID, [lista_posizioni])`.

Esempio di Indice Invertito risultante:

- `a`: (D2,2), (D2,9), (D2,13)
- `Brutus`: (D3,12)
- `Caesar`: (D1,12), (D3,5)
- `clear`: (D2,14)
- `him`: (D3,21)
- `I`: (D3,1), (D3,6)
- `lend`: (D1,4)
- `valley`: (D2,4)
- ... (e così via per tutti i termini)

16.3.3 Preprocessing Linguistico

Per migliorare velocità e accuratezza:

- **Normalizzazione dei Token:** Uniformare le parole. Es. "Windows" $\rightarrow$ "windows" (case folding). "U.S.A." $\rightarrow$ "USA".

Chiave	$H_1(K)$	$H_2(K)$	Bucket Iniziale	Bucket Finale
0000	0	00	0	00
1010	0	10	0	10 (dopo split)
1111	1	11	1	01
0101	1	01	1	01

Tabella 16.6: Redistribuzione delle chiavi dopo split nell'hashing lineare

Caratteristica	Hashing Statico	Hashing Estensibile	Hashing Lineare
Crescita	Nessuna (fisso)	A potenze di 2	Lineare (uno a uno)
Blocchi overflow	Sì	No	Sì
Funzione hash	$H_i(K)$ (primi i bit)	$H_i(K)$ (primi i bit)	$H_i(K)$ (ultimi i bit)
Directory	No	Sì	No
Complessità ricerca	$O(1)$ senza overflow	$O(1)$ se directory in RAM	$O(1)$ (più complessa)

Tabella 16.7: Confronto tra le tre tecniche di hashing

- **Stemming (Radicazione):** Ricondurre le parole alla loro radice (stem). Es. "fishing", "fished", "fisher" $\rightarrow$ "fish". Aiuta a trovare documenti rilevanti anche se usano forme diverse della stessa parola.
- **Stop Words:** Rimuovere parole comuni (articoli, preposizioni come "the", "a", "is", "and") perché appaiono in troppi documenti e non sono discriminanti, oltre a ridurre la dimensione dell'indice.

16.3.4 Esecuzione di Query con Indici Invertiti

Query di Insieme (Set Query - es. AND) "Trova documenti che contengono SIA 'Brutus' CHE 'Caesar'."

1. Prendi la posting list (solo DocID) per "Brutus": {D3}.
2. Prendi la posting list (solo DocID) per "Caesar": {D1, D3}.
3. Interseca le liste di DocID: {D3} $\cap$ {D1, D3} = {D3}.

Risultato: Documento D3.

Query di Sequenza (Phrase Query) "Trova documenti che contengono la sequenza 'a valley'."

1. Posting list per "a": $L_a = [(\text{D2}, 2), (\text{D2}, 9), (\text{D2}, 13)]$
2. Posting list per "valley": $L_v = [(\text{D2}, 4)]$
3. Per ogni $(doc\_id1, pos1)$ in L_a e ogni $(doc\_id2, pos2)$ in L_v:
   - Se $doc\_id1 == doc\_id2$ E $pos2 == pos1 + 1$, allora la sequenza è trovata.
4. Confronto:
   - Per "a": $(\text{D2}, 2)$. Per "valley": $(\text{D2}, 4)$. DocID uguali (D2). Posizione di "valley" (4) è uguale a posizione di "a" (2) + 1? No ($4 \neq 3$).
   - ... (altri confronti non producono match)

Risultato: Nessun documento contiene la sequenza "a valley".

"Trova documenti che contengono la sequenza 'a clear'."

1. Posting list per "a": $L_a = [(\text{D2}, 2), (\text{D2}, 9), (\text{D2}, 13)]$
2. Posting list per "clear": $L_c = [(\text{D2}, 14)]$
3. Confronto:
   - Per "a": $(\text{D2}, 13)$. Per "clear": $(\text{D2}, 14)$. DocID D2. $14 == 13 + 1$? Sì!

Risultato: Documento D2 contiene la sequenza "a clear" (dove "a" è in pos 13 e "clear" in pos 14).

16.4 Strategie per affrontare gli esercizi

- **Hashing:**
  - Identifica chiaramente il tipo di hashing (statico, estensibile, lineare).
  - Tieni traccia dei parametri chiave (i, j per estensibile; i, n_{buck}, r, soglia, split pointer s per lineare).
  - Per le inserzioni, segui meticolosamente i passi, specialmente quando avvengono split (di blocco o di directory/bucket). Disegnare la struttura aiuta molto.
  - Quando conti bucket/blocchi, ricorda la definizione: i bucket nell'hashing estensibile sono le entry della directory, i blocchi sono le strutture dati fisiche. Nell'hashing lineare, i bucket sono n_{buck}, i blocchi includono quelli di overflow.
- **Indici Invertiti:**
  - Per la costruzione, sii preciso nel tracciare le posizioni.
  - Per le query AND, si tratta di intersecare le liste di documenti.
  - Per le query di sequenza, dopo aver trovato i documenti comuni, devi verificare le condizioni sulle posizioni.