Appunti di Database e SQL

DBMS, Modellazione Dati e Linguaggio SQL

Prof. Luca Campion

Anno Scolastico 2025-2026

Istituto Tecnico Antonio Scarpa

Indice

Elenco delle figure

Listings

Prefazione

A chi si rivolge questo manuale

Questi appunti sono stati pensati per studenti di istituti tecnici e professionali che si avvicinano per la prima volta al mondo dei database e del linguaggio SQL. Il percorso è strutturato per accompagnare progressivamente dalla teoria alla pratica, dalla progettazione concettuale all'implementazione fisica di un database.

Struttura del corso

Il corso è organizzato in 13 capitoli che coprono l'intero ciclo di vita di un database:

Parte I - Teoria e Modellazione (Capitoli 1-5)

- Introduzione ai DBMS e ai sistemi informativi
- Modellazione concettuale con diagrammi Entity-Relationship
- Modello logico relazionale e algebra relazionale
- Normalizzazione per garantire integrità dei dati
- Progettazione fisica e ottimizzazione

Parte II - Linguaggio SQL (Capitoli 6-12)

- DDL: creazione e gestione strutture dati
- DML: interrogazione e manipolazione dati
- Query complesse con join e subquery
- Funzioni aggregate e raggruppamenti
- Gestione transazioni e concorrenza

Parte III - Pratica (Capitolo 13 + Appendice)

- Esercizi completi di progettazione e implementazione
- Casi di studio reali
- Soluzioni commentate

Prerequisiti

Per affrontare questo corso è consigliabile avere:

- Conoscenze base di informatica e algoritmi
- Familiarità con la rappresentazione dei dati
- Capacità di ragionamento logico e problem solving
- (Opzionale) Conoscenza di un linguaggio di programmazione

Strumenti necessari

Software consigliato:

- **MySQL/MariaDB**: DBMS relazionale open source
- **MySQL Workbench**: Tool grafico per progettazione e query
- **phpMyAdmin**: Interfaccia web per gestione database
- **DBeaver**: Client universale multi-database
- **SQLite**: Database embedded per esercitazioni rapide

Tool di modellazione:

- **draw.io**: Diagrammi ER online gratuiti
- **MySQL Workbench**: Modellazione integrata
- **Lucidchart**: Diagrammi professionali

Come studiare

Per ottenere il massimo da questi appunti:

1. **Leggi attentamente la teoria**: Ogni concetto è spiegato con esempi e diagrammi
2. **Disegna i diagrammi**: La modellazione si impara praticando
3. **Scrivi le query SQL**: Digita personalmente ogni query, non copiare
4. **Testa sul database**: Crea database di prova e verifica i risultati
5. **Risolvi gli esercizi**: Prova autonomamente prima di vedere le soluzioni
6. **Progetta casi reali**: Applica i concetti a scenari del mondo reale

Nota

Questo manuale segue lo standard **SQL ANSI/ISO** con focus su **MySQL** per gli esempi pratici. La maggior parte delle query funziona su tutti i DBMS relazionali (PostgreSQL, SQL Server, Oracle) con minime modifiche.

Convenzioni tipografiche

Nel testo vengono utilizzate le seguenti convenzioni:

- `Codice SQL`: Comandi e query in carattere monospace
- **Parole chiave SQL**: In grassetto nelle spiegazioni (SELECT, WHERE)
- *Nomi di tabelle/attributi*: In corsivo per riferimenti (Studente, nome)
- Box colorati: Attenzioni, Note, Errori Comuni, Esempi

Sito web e risorse

Materiale aggiuntivo disponibile su:

- Repository GitHub: https://github.com/campionluca/Appunti
- Database di esempio scaricabili
- Script SQL per esercitazioni
- Soluzioni interattive

Ringraziamenti

Si ringrazia l'Istituto Tecnico Antonio Scarpa per il supporto nella realizzazione di questo materiale didattico.

Prof. Luca Campion
Novembre 2025

Note sulla versione

Versione 1.0 - Novembre 2025

- Prima release completa
- 13 capitoli + appendice soluzioni
- Coverage SQL: DDL, DML, DCL, TCL
- Esempi MySQL 8.0+
- Diagrammi ER con TikZ

Capitolo 1

Introduzione ai DBMS

Introduzione

Un Database Management System (DBMS) è un software specializzato che gestisce la memorizzazione, l'organizzazione e l'accesso ai dati. Questo capitolo introduce i concetti fondamentali dei sistemi informativi e le proprietà ACID che garantiscono l'affidabilità dei dati.

Obiettivi di apprendimento

- Comprendere la differenza tra dati e informazioni
- Conoscere i componenti di un sistema informativo
- Capire il ruolo di un DBMS
- Apprendere le proprietà ACID delle transazioni
- Identificare vantaggi e caratteristiche dei DBMS

1.1 Dati, Informazioni e Sistemi Informativi

1.1.1 Differenza tra dati e informazioni

I **dati** sono fatti grezzi e non strutturati. Le **informazioni** sono dati elaborati e organizzati che hanno significato e utilità.

Esempio: Dati vs Informazioni

Dato: `25, 1990, Milano, Rossi`
Informazione: Il cliente Luigi Rossi, nato nel 1990 a Milano, ha 25 anni ed è residente in Lombardia.

1.1.2 Componenti di un sistema informativo

Un sistema informativo è composto da:

1. **Hardware**: computer, server, dispositivi di storage

2. **Software**: applicazioni, DBMS, sistemi operativi
3. **Dati**: il patrimonio informativo dell'organizzazione
4. **Processi**: le operazioni e le procedure aziendali
5. **Risorse umane**: amministratori, sviluppatori, utenti

1.2 Database Management System (DBMS)

1.2.1 Definizione e ruolo

Un DBMS è un software che consente di:

- Memorizzare dati in modo organizzato e persistente
- Recuperare dati efficientemente
- Aggiornare dati mantenendo la coerenza
- Controllare l'accesso ai dati
- Proteggere i dati da accessi non autorizzati

1.2.2 Vantaggi del DBMS

Vantaggi chiave

- **Centralizzazione**: un'unica fonte di verità
- **Efficienzaad accesso**: strutture dati ottimizzate
- **Sicurezza**: controllo degli accessi e crittografia
- **Affidabilità**: backup e recovery automatici
- **Integrità**: regole e vincoli garantiscono coerenza
- **Scalabilità**: gestisce grandi volumi di dati
- **Concorrenza**: accesso multiplo simultaneo

1.3 Le proprietà ACID

1.3.1 Proprietà ACID delle transazioni

Una transazione è una sequenza di operazioni che costituisce un'unità di lavoro logica. Le proprietà ACID garantiscono l'affidabilità:

A - Atomicità La transazione è indivisibile: si esegue completamente o non si esegue per nulla (all-or-nothing).

C - Coerenza Lo stato del database deve rimanere coerente prima e dopo la transazione.

I - Isolamento Transazioni concorrenti non si interferiscono l'una con l'altra.

D - Durabilità Una volta eseguita, una transazione persiste anche in caso di guasto.

1.3.2 Atomicità

Ogni transazione è atomica: non può rimanere nello stato intermedio.

Attenzione: Problema senza atomicità

Trasferimento di 100 euro tra due conti. Se il sistema crasha dopo il prelievo ma prima del deposito, il denaro è perso!

1.3.3 Coerenza

Il database mantiene regole e vincoli di integrità.

```
-- Il saldo non può mai essere negativo
ALTER TABLE conto ADD CONSTRAINT saldo_positivo
    CHECK (saldo >= 0);
```

Listing 1.1: Esempio: Vincolo di coerenza

1.3.4 Isolamento

Transazioni concorrenti non si vedono i reciproci effetti intermedi.

Nota: Livelli di isolamento

Existono diversi livelli di isolamento (READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE) per bilanciare performance e sicurezza.

1.3.5 Durabilità

I dati sono permanentemente memorizzati.

```
START TRANSACTION;
UPDATE conto SET saldo = saldo - 100 WHERE id = 1;
UPDATE conto SET saldo = saldo + 100 WHERE id = 2;
COMMIT; -- I dati sono ora permanenti
```

Listing 1.2: Esempio: Commit garantisce durabilità

1.4 Tipi di DBMS

1.4.1 DBMS Relazionali

Organizzano i dati in tabelle con relazioni tra loro. Esempi: MySQL, PostgreSQL, Oracle, SQL Server.

1.4.2 DBMS NoSQL

Memorizzano dati in formato non relazionale. Esempi: MongoDB, Cassandra, Redis.

1.4.3 DBMS NewSQL

Combinano vantaggi relazionali e NoSQL. Esempi: CockroachDB, TiDB.

1.5 Architettura di un DBMS

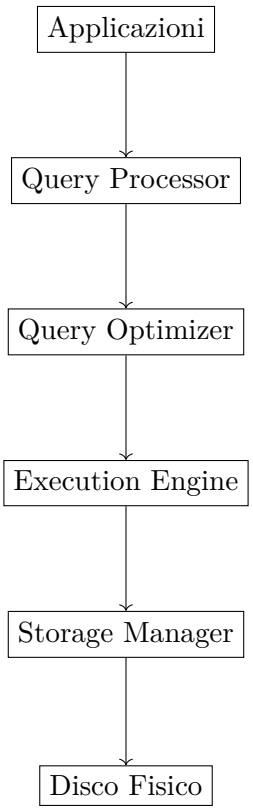


Figura 1.1: Architettura a livelli di un DBMS

Riepilogo concetti chiave

Concetti fondamentali

- Un **DBMS** centralizza e gestisce i dati di un'organizzazione
- Le proprietà **ACID** garantiscono transazioni affidabili
- L'**atomicità** garantisce tutto-o-nulla
- La **coerenza** mantiene l'integrità dei vincoli
- L'**isolamento** separa le transazioni concorrenti
- La **durabilità** rende i dati permanenti

Esercizi

1. Spiega con un esempio concreto la differenza tra data e informazione.
2. Descrivi come le proprietà ACID garantiscono un trasferimento bancario sicuro tra due conti.
3. Un'operazione di trasferimento fallisce a metà. Quale proprietà ACID evita che il denaro scompaia? Spiega.
4. Elenca almeno tre vantaggi di un DBMS rispetto a un archivio basato su file di testo.
5. Qual è la differenza tra DBMS relazionali e NoSQL? Fai un esempio di situazione dove sceglieresti l'uno o l'altro.

Capitolo 2

Modello Concettuale - Diagrammi ER

Introduzione

Il modello Entità-Relazione (ER) è uno strumento per rappresentare la struttura logica dei dati indipendentemente dall'implementazione fisica. Questo capitolo presenta i diagrammi ER, le entità, gli attributi e le relazioni con le loro cardinalità.

Obiettivi di apprendimento

- Comprendere i componenti del modello ER
- Disegnare entità e attributi
- Identificare e modellare relazioni tra entità
- Comprendere la cardinalità delle relazioni
- Applicare le chiavi primarie e esterne nel modello concettuale
- Tradurre requisiti aziendali in diagrammi ER

2.1 Componenti del Modello ER

2.1.1 Entità

Un'**entità** è un oggetto del mondo reale di interesse per il sistema informativo. Rappresenta una classe di oggetti con proprietà comuni.

Esempio: Entità

Entità: **Cliente**, **Prodotto**, **Ordine**, **Dipendente**
Ogni entità rappresenta un concetto del dominio aziendale.

Nel diagramma ER, un'entità è rappresentata come un rettangolo.

2.1.2 Attributi

Un **attributo** è una proprietà di un'entità. Descrive caratteristiche specifiche dell'entità.

Esempio: Attributi

Entità **Cliente** ha attributi:

- `idCliente` (identificativo univoco)
- `nome`
- `cognome`
- `email`
- `dataRegistrazione`

Tipi di attributi

Semplice Non è scomponibile (es. `email`)

Composto Scomponibile in attributi più piccoli (es. `indirizzo` = `via, numero, città, CAP`)

Univoco/Chiave Identifica univocamente l'entità (es. `idCliente`)

Opzionale Può non avere un valore (es. `numeroCellulare`)

Multivalore Può assumere più valori (es. `telefoni` di una ditta)

2.1.3 Relazioni

Una **relazione** (o associazione) descrive come due o più entità interagiscono tra loro.

Esempio: Relazioni

- Un **Cliente** *effettua* un **Ordine**
- Un **Ordine** *contiene* **Prodotti**
- Un **Dipendente** *lavora in* un **Dipartimento**

2.2 Cardinalità delle Relazioni

La **cardinalità** specifica quanti elementi di un'entità possono essere correlati con elementi dell'altra entità.

2.2.1 Tipi di cardinalità

1:1 (uno a uno) Ogni elemento di A è associato a esattamente un elemento di B e viceversa.

1:N (uno a molti) Ogni elemento di A è associato a uno o più elementi di B, ma ogni elemento di B è associato a un solo elemento di A.

N:M (molti a molti) Ogni elemento di A può essere associato a più elementi di B e viceversa.

0..1 Opzionale: da zero a uno

1..N Obbligatorio: da uno a molti

2.2.2 Esempio 1: Relazione 1:1

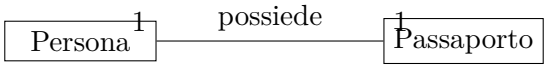


Figura 2.1: Una persona possiede esattamente un passaporto

Una persona può avere un solo passaporto e ogni passaporto appartiene a una sola persona.

2.2.3 Esempio 2: Relazione 1:N

Figura 2.2: Un cliente può effettuare molti ordini

Un cliente può fare molti ordini, ma ogni ordine è fatto da un solo cliente.

2.2.4 Esempio 3: Relazione N:M

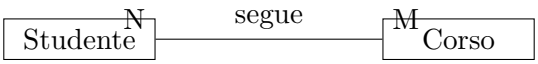


Figura 2.3: Uno studente segue molti corsi e un corso è seguito da molti studenti

Uno studente può seguire più corsi e ogni corso può avere più studenti.

2.3 Diagramma ER Completo: Sistema di Vendita Online

2.4 Chiavi nel Modello ER

2.4.1 Chiave Primaria

La **chiave primaria** è un attributo (o insieme di attributi) che identifica univocamente ogni istanza dell'entità. Non può essere nulla e deve essere univoca.

Esempio: Chiave Primaria

Entità **Cliente**:

- **idCliente** è la chiave primaria (numero univoco assegnato a ogni cliente)
- Anche **email** potrebbe essere chiave primaria (ogni cliente ha un'email unica)

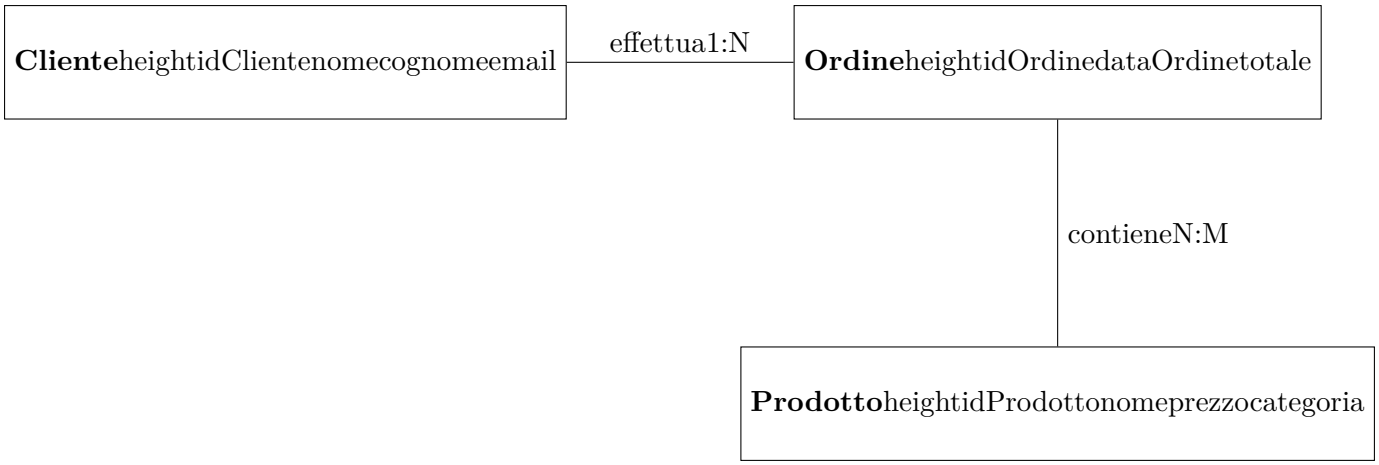


Figura 2.4: Diagramma ER semplificato per un e-commerce

2.4.2 Chiave Esterna

Una **chiave esterna** è un attributo che riferisce la chiave primaria di un'altra entità, creando un collegamento.

Esempio: Chiave Esterna

Entità **Ordine**:

- **idOrdine** è la chiave primaria
- **idCliente** è una chiave esterna (riferisce la chiave primaria di Cliente)

Riepilogo concetti chiave

Concetti fondamentali

- Un'**entità** rappresenta una classe di oggetti del dominio
- Un **attributo** descrive una proprietà dell'entità
- Una **relazione** collega due o più entità
- La **cardinalità** specifica il numero di associazioni (1:1, 1:N, N:M)
- La **chiave primaria** identifica univocamente l'entità
- La **chiave esterna** crea il collegamento tra entità

Esercizi

1. Disegna un diagramma ER per una biblioteca con le entità: Libro, Autore, Membro, Prestito. Includi attributi appropriati e cardinalità.
2. Spiega la differenza tra cardinalità 1:N e N:M con due esempi concreti diversi da quelli del capitolo.

3. Un'università ha Studenti, Corsi e Professori. Quali sono le relazioni tra questi? Quale cardinalità?

4. Identifica la chiave primaria e le chiavi esterne nel diagramma dell'e-commerce presentato nel capitolo.

5. Trasforma il seguente requisito aziendale in un diagramma ER: "Una società ha Dipartimenti. Ogni Dipartimento ha più Dipendenti. Un Dipendente lavora in un solo Dipartimento. Un Dipendente gestisce Progetti. Un Progetto può essere gestito da un solo Dipendente. Un Progetto coinvolge più Dipendenti."

Capitolo 3

Modello Logico - Modello Relazionale

Introduzione

Il modello logico traduce il modello concettuale (diagrammi ER) in un modello implementabile nei DBMS. Il modello relazionale organizza i dati in tabelle (relazioni) con colonne (attributi) e righe (tuple). Questo capitolo presenta la struttura del modello relazionale, le chiavi e i vincoli di integrità.

Obiettivi di apprendimento

- Comprendere la struttura del modello relazionale
- Tradurre diagrammi ER in tabelle relazionali
- Definire e utilizzare chiavi primarie e esterne
- Comprendere i vincoli di integrità referenziale
- Applicare le regole di trasformazione ER-relazionale
- Progettare schemi logici corretti

3.1 Il Modello Relazionale

3.1.1 Componenti fondamentali

Il modello relazionale è basato su tre concetti chiave:

Relazione (Tabella) Una collezione di tuple (righe) con lo stesso insieme di attributi (colonne).

Attributo (Colonna) Una proprietà della relazione con un nome e un dominio (tipo di dato).

Tupla (Riga) Un record che contiene un valore per ogni attributo.

Esempio: Tabella Cliente

idCliente	nome	cognome	email
1	Luigi	Rossi	luigi@email.com
2	Maria	Bianchi	maria@email.com
3	Giovanni	Verdi	giovanni@email.com

Ogni riga è una tupla, ogni colonna è un attributo.

3.1.2 Dominio di un attributo

Ogni attributo ha un **dominio**, cioè l'insieme di valori che può assumere.

Esempio: Domini

- `idCliente`: dominio = interi positivi (1, 2, 3, ...)
- `nome`: dominio = stringhe di caratteri (fino a 100 caratteri)
- `dataIscrizione`: dominio = date (formato YYYY-MM-DD)
- `saldo`: dominio = numeri decimali positivi

3.2 Chiavi nel Modello Relazionale

3.2.1 Chiave Primaria

Una **chiave primaria** è un attributo (o insieme di attributi) che:

- Identifica univocamente ogni tupla della relazione
- Non può contenere valori NULL
- Non può avere duplicati

```sql
CREATE TABLE cliente (
    idCliente INT PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    cognome VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE
);
```

Listing 3.1: Dichiarazione di chiave primaria

3.2.2 Chiave Candidata

Una **chiave candidata** è un attributo che potrebbe essere chiave primaria. Una relazione può avere più chiavi candidate, ma solo una viene scelta come primaria.

Tabella Cliente:

- `idCliente`: chiave candidata (univoca, non nulla)

- `email`: chiave candidata (univoca, non nulla)
- Sceglieremo `idCliente` come primaria

C

3.2.3 Chiave Esterna

Una **chiave esterna** è un attributo (o insieme di attributi) che referenzia la chiave primaria di un'altra relazione. Crea il collegamento tra tabelle.

BCREATE TABLE ordine (idOrdine INT PRIMARY KEY, dataOrdine DATE NOT NULL, idCliente INT NOT NULL, FOREIGN KEY (idCliente) REFERENCES cliente(idCliente));

3.2.4 Chiave Composta

Una chiave può essere formata da più attributi.

```sql
CREATE TABLE ordine_prodotto (
    idOrdine INT NOT NULL,
    idProdotto INT NOT NULL,
    quantita INT NOT NULL,
    PRIMARY KEY (idOrdine, idProdotto),
    FOREIGN KEY (idOrdine) REFERENCES ordine(idOrdine),
    FOREIGN KEY (idProdotto) REFERENCES prodotto(idProdotto)
);
```

Listing 3.2: Chiave composta come chiave primaria

3.3 Vincoli di Integrità

3.3.1 Vincolo di dominio

Ogni valore di un attributo deve appartenere al dominio definito.

```sql
CREATE TABLE prodotto (
    idProdotto INT PRIMARY KEY,
    nome VARCHAR(200) NOT NULL,       -- Non null
    prezzo DECIMAL(8, 2) NOT NULL,    -- Numero con 2 decimali
    categoria ENUM('Elettronica', 'Libri', 'Abbigliamento'),
    quantitaDisponibile INT CHECK (quantitaDisponibile >= 0)
);
```

Listing 3.3: Vincoli di dominio

3.3.2 Vincolo di unicità

Un attributo (o insieme di attributi) può avere solo valori unici (o nulli).

```sql
CREATE TABLE utente (
    idUtente INT PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    dataRegistrazione DATE DEFAULT CURDATE()
);
```

Listing 3.4: Vincolo di unicità

3.3.3 Vincolo di chiave primaria

Garantisce l'univocità e l'obbligatorietà (NOT NULL).

3.3.4 Vincolo di chiave esterna e integrità referenziale

La chiave esterna assicura che ogni valore riferisca effettivamente un'entità in un'altra relazione.

Attenzione: Violazione di integrità referenziale

Se in ordine abbiamo `idCliente = 5`, questo `idCliente` DEVE esistere in cliente. Altrimenti il DBMS rifiuta l'inserimento.

```sql
CREATE TABLE ordine (
    idOrdine INT PRIMARY KEY,
    idCliente INT NOT NULL,
    FOREIGN KEY (idCliente) REFERENCES cliente(idCliente)
        ON DELETE CASCADE       -- Elimina ordini se cliente è
            eliminato
        ON UPDATE CASCADE       -- Aggiorna idCliente se cambia in
            cliente
);
```

Listing 3.5: Azioni su integrità referenziale

3.4 Trasformazione da ER a Modello Relazionale

3.4.1 Regola 1: Entità singole

Ogni entità diventa una tabella. Gli attributi della tabella sono i rispettivi attributi dell'entità. La chiave primaria dell'entità diventa chiave primaria della tabella.

Esempio: Entità Cliente

ER: Entità Cliente con attributi (idCliente, nome, cognome, email)
Relazionale:

```
CREATE TABLE cliente (
    idCliente INT PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    cognome VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE
);
```

3.4.2 Regola 2: Relazione 1:N

Il lato N contiene una chiave esterna che referenzia la chiave primaria del lato 1.

ER: Cliente effettua Ordine (1:N)

Relazionale:

```
CREATE TABLE cliente (
    idCliente INT PRIMARY KEY,
    nome VARCHAR(100)
);

CREATE TABLE ordine (
    idOrdine INT PRIMARY KEY,
    dataOrdine DATE NOT NULL,
    idCliente INT NOT NULL,
    FOREIGN KEY (idCliente) REFERENCES cliente(idCliente)
);
```

3.4.3 Regola 3: Relazione 1:1

Esiste una chiave esterna in una delle due tabelle. Generalmente nel lato con partecipazione opzionale.

3.4.4 Regola 4: Relazione N:M

Si crea una **tabella di giunzione** (o tabella di associazione) con chiavi esterne che referenziano entrambe le entità. La chiave primaria è la combinazione delle due chiavi esterne.

```
    responsabile INT,
    FOREIGN KEY (responsabile) REFERENCES impiegato(matricola)
);
```

5. Trasforma il seguente diagramma ER in schema relazionale: Azienda ha Sedi. Una Sede ha più Dipendenti. Un Dipendente appartiene a una Sede. Un Dipendente gestisce Progetti. Un Progetto è gestito da un Dipendente. Un Progetto ha più Risorse. Una Risorsa è utilizzata da più Progetti.

Capitolo 4

Normalizzazione del Database

Introduzione

La normalizzazione è un processo sistematico per organizzare i dati eliminando ridondanze e anomalie. Attraverso forme normali progressive (1NF, 2NF, 3NF, BCNF), garantiamo un design del database efficiente, coerente e facile da mantenere.

Obiettivi di apprendimento

- Comprendere il concetto di dipendenza funzionale
- Applicare le regole della prima forma normale (1NF)
- Applicare le regole della seconda forma normale (2NF)
- Applicare le regole della terza forma normale (3NF)
- Comprendere la forma normale di Boyce-Codd (BCNF)
- Identificare e risolvere anomalie di inserimento, aggiornamento e cancellazione

4.1 Dipendenze Funzionali

4.1.1 Definizione

Una **dipendenza funzionale** (DF) X -> Y fra due insiemi di attributi X e Y di una relazione R significa che, se due tuple hanno gli stessi valori per gli attributi in X, allora devono avere gli stessi valori per gli attributi in Y.

Esempio: Dipendenza funzionale

Relazione: Studente(idStudente, nome, email, idCorso, nomeCorso)
Dipendenze funzionali:

- `idStudente` -> `nome, email`: dato uno studente, c'è un solo nome e email
- `idCorso` -> `nomeCorso`: dato un corso, c'è un solo nome

4.1.2 Chiave primaria e dipendenze

Se un attributo è parte della chiave primaria, determina funzionalmente tutti gli altri attributi della relazione.

4.2 Anomalie nei Database non normalizzati

4.2.1 Anomalia di inserimento

Non è possibile inserire dati fino a quando non sono disponibili tutti i dati della chiave primaria.

Esempio: Anomalia di inserimento

Tabella: Corso(idCorso, nomeCorsso, idDocente, nomeDocente)
Non posso inserire un docente che non insegna ancora nessun corso perché `idCorso` è parte della chiave primaria.

4.2.2 Anomalia di aggiornamento

Aggiornare un valore può richiedere di aggiornare più tuple, rischiando inconsistenze.

Esempio: Anomalia di aggiornamento

Tabella: Studente(idStudente, nome, idFacoltà, nomeFacoltà)
Se cambio il nome della Facoltà da "Ingegneria" a "Ingegneria e Architettura", devo aggiornare tutte le righe degli studenti di quella facoltà. Se dimentico una riga, il database diventa inconsistente.

4.2.3 Anomalia di cancellazione

Cancellare un dato può comportare la perdita di informazioni non correlate.

Esempio: Anomalia di cancellazione

Tabella: Fornitore$_{prodotto}$($idFornitore, nomeFornitore, idProdotto, nomeProdotto$)
Se cancello l'ultimo prodotto fornito da un fornitore, perdo anche i dati del fornitore dal database.

4.3 Prima Forma Normale (1NF)

La forma normale più basilare: **una relazione è in 1NF se tutti gli attributi contengono solo valori atomici** (non divisibili).

4.3.1 Requisiti

- Nessun attributo può contenere liste o insiemi di valori
- Nessun attributo ripetuto

- Ogni riga deve avere lo stesso numero di colonne
- Nessun gruppo ripetuto

4.3.2 Esempio di violazione 1NF

idStudente	nome	corsi
1	Mario	Matematica, Fisica, Chimica
2	Luigi	Matematica, Biologia

L'attributo `corsi` contiene valori non atomici (liste).

idStudente	nome	corso
1	Mario	Matematica
1	Mario	Fisica
1	Mario	Chimica
2	Luigi	Matematica
2	Luigi	Biologia

Oppure creare una tabella separata per gli inscritti ai corsi (relazione N:M).

C

bCREATE TABLE studente (idStudente INT PRIMARY KEY, nome VARCHAR(100));

CREATE TABLE corso (idCorso INT PRIMARY KEY, nomeCors VARCHAR(100));

CREATE TABLE iscrizione (idStudente INT NOT NULL, idCorso INT NOT NULL, PRIMARY KEY (idStudente, idCorso), FOREIGN KEY (idStudente) REFERENCES studente(idStudente), FOREIGN KEY (idCorso) REFERENCES corso(idCorso));

4.4 Seconda Forma Normale (2NF)

Una relazione è in 2NF se:

1. È in 1NF
2. **Ogni attributo non-chiave è dipendente dalla chiave primaria completa**, non solo da parte di essa.

Problema: Dipendenza parziale. Un attributo non-chiave dipende solo da una parte della chiave primaria.

4.4.1 Esempio di violazione 2NF

Tabella: Iscrizione(idStudente, idCorso, nomeCors, voto)

Chiave primaria: (idStudente, idCorso)

Problema: `nomeCors` dipende solo da `idCorso`, non dalla chiave completa. Questa è una dipendenza parziale.

C

4.4.2 Correzione: Applicare 2NF

BCREATE TABLE studente (idStudente INT PRIMARY KEY, nome VARCHAR(100));

CREATE TABLE corso (idCorso INT PRIMARY KEY, nomeCors VARCHAR(100));

– Ora in 2NF: nomeCors è in tabella corso, non in iscrizione CREATE TABLE iscrizione (idStudente INT NOT NULL, idCorso INT NOT NULL, voto INT, PRIMARY KEY (idStudente, idCorso), FOREIGN KEY (idStudente) REFERENCES studente(idStudente), FOREIGN KEY (idCorso) REFERENCES corso(idCorso));

4.5 Terza Forma Normale (3NF)

Una relazione è in 3NF se:

1. È in 2NF
2. **Nessun attributo non-chiave dipende da un altro attributo non-chiave**.

Problema: Dipendenza transitiva. Un attributo non-chiave dipende funzionalmente da un altro attributo non-chiave.

4.5.1 Esempio di violazione 3NF

Tabella: Ordine(idOrdine, dataOrdine, idCliente, nomeCliente, indirizzo)

Dipendenze funzionali:

- `idOrdine` -> `dataOrdine, idCliente, nomeCliente, indirizzo`
- `idCliente` -> `nomeCliente, indirizzo` (dipendenza transitiva!)

`nomeCliente` e `indirizzo` dipendono da `idCliente`, non direttamente da `idOrdine`.

C

4.5.2 Correzione: Applicare 3NF

BCREATE TABLE cliente (idCliente INT PRIMARY KEY, nomeCliente VARCHAR(100), indirizzo VARCHAR(200));

CREATE TABLE ordine (idOrdine INT PRIMARY KEY, dataOrdine DATE, idCliente INT NOT NULL, FOREIGN KEY (idCliente) REFERENCES cliente(idCliente));

4.6 Forma Normale di Boyce-Codd (BCNF)

Una relazione è in BCNF se:

- Per ogni dipendenza funzionale X -> Y, X è una chiave candidata.

BCNF è una forma più stringente di 3NF. La maggior parte dei database in 3NF è già in BCNF.

4.6.1 Esempio di violazione BCNF

Tabella: Professore$_{C}orso(idProfessore, idCorso, orario)$

Chiave primaria: (idProfessore, idCorso)

Dipendenza funzionale: (idProfessore, idCorso) -> orario (OK)

Ma se esiste anche: idCorso -> orario (tutti i corsi hanno un orario fisso)

Allora idCorso determina orario, ma idCorso non è una chiave candidata. Violazione BCNF!

4.7 Procedura di Normalizzazione

Riepilogo concetti chiave

Esercizi

1. Identifica tutte le dipendenze funzionali nella seguente tabella: Libro(idLibro, titolo, autore, idAutore, nazionaleAutore, genere, prezzo)

2. La seguente tabella è in 1NF? Se no, normalizza: Impiegato(matricola, nome, skills) bLa seguente tabella è in 2NF? Se no, normalizza: Voto(idStudente, nomeStudente, idCorso, nomeCorso, voto)

   La seguente tabella è in 3NF? Se no, normalizza: Ospedale(idOspedale, nomeOspedale, città, idRepartimento, nomeRepartimento, idMedico, nomeMedico)

   Progetta uno schema completamente normalizzato (3NF) per un sistema di prenotazioni alberghiere con: Hotel, Stanze, Clienti, Prenotazioni. Includi vincoli appropriati.

Capitolo 5

Modello Fisico e Ottimizzazione

Introduzione

Il modello fisico descrive come i dati sono effettivamente memorizzati e organizzati nel disco fisico. Include indici, viste materializzate, partizionamento e altre strategie di ottimizzazione per migliorare performance e scalabilità.

Obiettivi di apprendimento

- Comprendere come i dati sono memorizzati nel disco
- Progettare e utilizzare indici efficaci
- Comprendere le viste materializzate e il caching
- Applicare strategie di partizionamento
- Ottimizzare query e accessi ai dati
- Bilanciare velocità di lettura e velocità di scrittura

5.1 Memorizzazione dei Dati

5.1.1 Struttura di memorizzazione

I dati vengono memorizzati in blocchi (pagine) sul disco:

- **Blocco**: unità minima di trasferimento tra disco e memoria (tipicamente 4-16 KB)
- **Pagina**: blocco logico del DBMS
- **Record**: singola tupla della tabella
- **Campo**: singolo attributo di un record

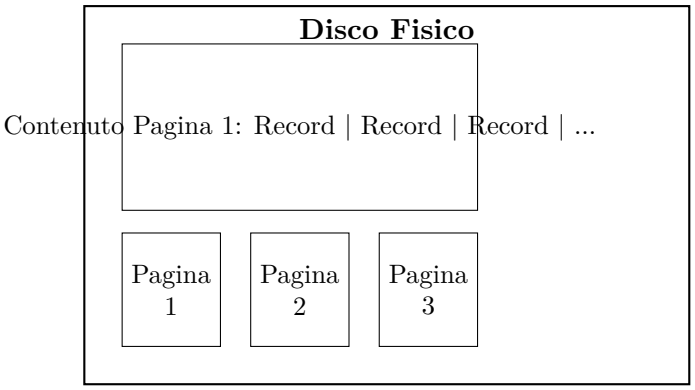


Figura 5.1: Struttura di memorizzazione su disco

5.1.2 Tipi di accesso

Sequential access : Leggere i record uno dopo l'altro sequenzialmente. Lento per record specifici.

Random access : Accedere direttamente a un record senza leggerne altri. Veloce ma necessita di indici.

5.2 Indici

Un **indice** è una struttura dati che accelera il recupero di record da una tabella. Funziona come un indice di un libro: invece di leggere tutte le pagine, consulti l'indice per trovare la pagina che cerchi.

5.2.1 Vantaggi degli indici

- Accelera ricerche e filtri (WHERE)
- Accelera ordinamenti (ORDER BY)
- Accelera join tra tabelle
- Supporta query range (es. WHERE prezzo BETWEEN 10 AND 100)

5.2.2 Svantaggi degli indici

- Occupano spazio aggiuntivo su disco
- Ralentano INSERT, UPDATE, DELETE (l'indice deve essere aggiornato)
- Richiedono manutenzione periodica

5.2.3 Tipi di indici

Indice primario

Un indice su una chiave primaria. È automaticamente creato quando si definisce una PRIMARY KEY.

```sql
CREATE TABLE cliente (
    idCliente INT PRIMARY KEY,  -- Indice primario automatico
    nome VARCHAR(100)
);
```

Listing 5.1: Indice primario (creato automaticamente)

Indice secondario

Un indice su attributi non-chiave per accelerare ricerche frequenti.

```sql
CREATE TABLE cliente (
    idCliente INT PRIMARY KEY,
    nome VARCHAR(100),
    email VARCHAR(100),
    città VARCHAR(50)
);

-- Indici su campi frequentemente cercati
CREATE INDEX idx_email ON cliente(email);
CREATE INDEX idx_città ON cliente(città);
```

Listing 5.2: Creare un indice secondario

Indice composito

Un indice su più attributi, utile per query che filtrano su più colonne.

```sql
CREATE TABLE ordine (
    idOrdine INT PRIMARY KEY,
    dataOrdine DATE,
    idCliente INT,
    stato VARCHAR(20)
);

-- Indice su due colonne per query come:
-- WHERE idCliente = 5 AND stato = 'Spedito'
CREATE INDEX idx_cliente_stato ON ordine(idCliente, stato);
```

Listing 5.3: Indice composito

Indice a hash

Usa una funzione hash per mappare i valori agli indirizzi di memoria. Veloce per uguaglianza, non per range.

Indice B-Tree

La struttura più comune. Ordinato, efficiente per range e ordinamenti. Bilanciato automaticamente.

```sql
-- Mostra dettagli degli indici di una tabella
SHOW INDEX FROM cliente;

-- Analizza l'efficienza dell'indice
EXPLAIN SELECT * FROM cliente WHERE email = 'mario@email.com';
```

Listing 5.4: Statistiche dell'indice (MySQL)

5.2.4 Strategie di indicizzazione

Buone pratiche per gli indici

- Indicizza campi usati frequentemente in WHERE, JOIN, ORDER BY
- Non indicizzare campi usati raramente (spreco di spazio)
- Indici compositi per query multi-colonna frequenti
- Evita indici su colonne con pochi valori distinti (low cardinality)
- Monitora e mantieni gli indici regolarmente

5.3 Viste Materializzate

Una **vista materializzata** è il risultato di una query pre-calcolato e memorizzato fisicamente sul disco. A differenza di una vista normale, occupa spazio e deve essere aggiornato.

5.3.1 Vantaggi

- Query molto veloce (dati pre-calcolati)
- Riduce carico computazionale
- Utile per query complesse su dati storici

5.3.2 Svantaggi

- Occupa spazio su disco
- Deve essere aggiornato periodicamente
- Dati possono essere non aggiornati (stali)

5.3.3 Esempio di vista materializzata

```sql
-- Vista normale (calcolata ogni volta)
CREATE VIEW vendite_mensili AS
    SELECT
        MONTH(dataOrdine) AS mese,
        YEAR(dataOrdine) AS anno,
        SUM(totale) AS ricavi,
        COUNT(*) AS numOrdini
    FROM ordine
    GROUP BY YEAR(dataOrdine), MONTH(dataOrdine);

-- In MySQL, simula una vista materializzata con tabella
CREATE TABLE vendite_mensili_cache (
    mese INT,
    anno INT,
    ricavi DECIMAL(12, 2),
    numOrdini INT,
    dataAggiornamento TIMESTAMP
) AS
SELECT
    MONTH(dataOrdine) AS mese,
    YEAR(dataOrdine) AS anno,
    SUM(totale) AS ricavi,
    COUNT(*) AS numOrdini
FROM ordine
GROUP BY YEAR(dataOrdine), MONTH(dataOrdine);

-- Aggiorna periodicamente (es. ogni notte)
-- DELETE FROM vendite_mensili_cache;
-- INSERT INTO vendite_mensili_cache (SELECT ...);
```

Listing 5.5: Vista materializzata

5.4 Partizionamento

Il **partizionamento** divide una tabella grande in parti più piccole (partizioni) basate su criteri specifici, migliorando performance e manutenzione.

5.4.1 Tipi di partizionamento

Partizionamento per range

Divide i dati in base a intervalli di valori.

```sql
CREATE TABLE ordine (
    idOrdine INT,
    dataOrdine DATE,
    totale DECIMAL(10, 2)
)
PARTITION BY RANGE (YEAR(dataOrdine)) (
    PARTITION p2020 VALUES LESS THAN (2021),
    PARTITION p2021 VALUES LESS THAN (2022),
```

```
    PARTITION p2022 VALUES LESS THAN (2023),
    PARTITION p2023 VALUES LESS THAN (2024),
    PARTITION pmax VALUES LESS THAN MAXVALUE
);
```

Listing 5.6: Partizionamento per range (anno)

Partizionamento per list

Divide i dati in base a valori specifici.

```
CREATE TABLE ordine (
    idOrdine INT,
    stato VARCHAR(20),
    totale DECIMAL(10, 2)
)
PARTITION BY LIST (stato) (
    PARTITION p_pending VALUES IN ('Pendente'),
    PARTITION p_shipped VALUES IN ('Spedito'),
    PARTITION p_delivered VALUES IN ('Consegnato'),
    PARTITION p_other VALUES IN (DEFAULT)
);
```

Listing 5.7: Partizionamento per list

Partizionamento per hash

Distribuisce i dati in base a una funzione hash.

```
CREATE TABLE ordine (
    idOrdine INT,
    idCliente INT,
    totale DECIMAL(10, 2)
)
PARTITION BY HASH(idCliente)
PARTITIONS 4;  -- 4 partizioni
```

Listing 5.8: Partizionamento per hash

5.4.2 Vantaggi del partizionamento

- Query più veloci (scansione solo partizioni rilevanti)
- Manutenzione più agevole (backup per partizione)
- Scalabilità orizzontale

5.5 Ottimizzazione di Query

5.5.1 Utilizzo di EXPLAIN

Il comando EXPLAIN analizza il piano di esecuzione di una query.

```
EXPLAIN SELECT *
FROM ordine o
JOIN cliente c ON o.idCliente = c.idCliente
WHERE c.città = 'Milano'
ORDER BY o.dataOrdine DESC;
```

Listing 5.9: Analizzare il piano di esecuzione

L'output mostra:

- Quale indice viene usato
- Quante righe vengono esaminate
- Il costo relativo di ogni operazione

5.5.2 Statistiche e manutenzione

```
-- Ricalcola statistiche tabella
ANALYZE TABLE ordine;

-- Ottimizza spazio tabella
OPTIMIZE TABLE ordine;

-- Mostra statistiche tabella
SHOW TABLE STATUS FROM database_name;
```

Listing 5.10: Manutenzione degli indici

Riepilogo concetti chiave

Concetti fondamentali

- I dati sono memorizzati in **pagine/blocchi** sul disco
- Gli **indici** accelerano ricerche, JOIN e ordinamenti
- Le **viste materializzate** pre-calcolano risultati frequenti
- Il **partizionamento** divide grandi tabelle in parti più piccole
- Bilanciare velocità di lettura (indici) con velocità di scrittura
- Monitorare con EXPLAIN e statistiche per identificare colli di bottiglia

Esercizi

1. Progetta una strategia di indicizzazione per una tabella con milioni di ordini. Quali campi indicizzaresti? Perché?
2. Una query SELECT * FROM cliente WHERE città = 'Roma' AND età > 30 è lenta. Come potrebbe aiutare un indice composito?
3. Qual è la differenza tra una vista normal e una vista materializzata? Quando useresti ciascuna?
4. Proponi un partizionamento per una tabella logs con milioni di record inseriti ogni giorno.
5. Analizza il seguente piano di esecuzione e proponi ottimizzazioni:

```sql
EXPLAIN SELECT *
FROM ordine
WHERE dataOrdine > '2023-01-01' AND stato = 'Spedito';
```

6. Crea una vista materializzata per memorizzare le vendite totali per cliente nel 2023. Come la aggiornare settimanalmente?

Capitolo 6

SQL DDL - Definizione dei Dati

Introduzione

DDL (Data Definition Language) è il linguaggio SQL per definire la struttura del database. Consente di creare, modificare e eliminare tabelle, indici, viste e altri oggetti dello schema. Questo capitolo presenta i principali comandi DDL: CREATE, ALTER, DROP.

Obiettivi di apprendimento

- Creare tabelle con tipi di dati appropriati
- Definire vincoli (PRIMARY KEY, FOREIGN KEY, UNIQUE, CHECK, DEFAULT)
- Modificare strutture di tabelle esistenti (ALTER TABLE)
- Eliminare tabelle, colonne, indici (DROP)
- Creare e gestire indici
- Definire chiavi esterne con azioni ON DELETE e ON UPDATE
- Usare tipi di dati appropriati per i dati

6.1 Tipi di Dati in MySQL

6.1.1 Tipi numerici

INT Numero intero a 4 byte. Range: -2.147.483.648 a 2.147.483.647

SMALLINT Numero intero a 2 byte. Range: -32.768 a 32.767

BIGINT Numero intero a 8 byte. Per numeri molto grandi

DECIMAL(p,s) Numero decimale con precisione (p) e scala (s). Es: DECIMAL(10,2)

FLOAT, DOUBLE Numeri decimali in virgola mobile. Meno precisi di DECIMAL

TINYINT Numero intero a 1 byte. Range: -128 a 127 (o 0-255 se UNSIGNED)

6.1.2 Tipi stringa

CHAR(n) Stringa a lunghezza fissa di n caratteri. Spreca spazio se la stringa è più breve

VARCHAR(n) Stringa a lunghezza variabile fino a n caratteri. Efficiente

TEXT Stringa di lunghezza variabile lunga (fino a 4 GB)

BLOB Dati binari (immagini, file)

ENUM Stringa da un set predefinito. Es: ENUM('Attivo', 'Inattivo')

6.1.3 Tipi data/ora

DATE Data in formato YYYY-MM-DD

TIME Ora in formato HH:MM:SS

DATETIME Data e ora in formato YYYY-MM-DD HH:MM:SS

TIMESTAMP Data e ora con timestamp Unix. Auto-aggiornamento

YEAR Anno in formato YYYY

6.2 CREATE TABLE

Il comando CREATE TABLE crea una nuova tabella con i relativi attributi e vincoli.

6.2.1 Sintassi base

```
CREATE TABLE nome_tabella (
    nome_attributo tipo [vincoli],
    nome_attributo tipo [vincoli],
    ...
);
```

Listing 6.1: Sintassi CREATE TABLE

6.2.2 Esempio semplice

```
CREATE TABLE cliente (
    idCliente INT PRIMARY KEY AUTO_INCREMENT,
    nome VARCHAR(100) NOT NULL,
    cognome VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    dataRegistrazione DATE DEFAULT CURDATE(),
    saldo DECIMAL(10, 2) DEFAULT 0.00
);
```

Listing 6.2: Creazione tabella Cliente

6.2.3 Vincoli

PRIMARY KEY

Identifica univocamente ogni riga. Non può essere NULL e deve essere unico.

```sql
CREATE TABLE studente (
    matricola INT PRIMARY KEY,
    nome VARCHAR(100)
);

-- O alternativamente
CREATE TABLE studente (
    matricola INT,
    nome VARCHAR(100),
    PRIMARY KEY (matricola)
);

-- Chiave primaria composta
CREATE TABLE iscrizione (
    idStudente INT,
    idCorso INT,
    voto INT,
    PRIMARY KEY (idStudente, idCorso)
);
```

Listing 6.3: PRIMARY KEY

FOREIGN KEY

Referenzia la chiave primaria di un'altra tabella. Garantisce l'integrità referenziale.

```sql
CREATE TABLE ordine (
    idOrdine INT PRIMARY KEY AUTO_INCREMENT,
    dataOrdine DATE NOT NULL,
    idCliente INT NOT NULL,
    FOREIGN KEY (idCliente) REFERENCES cliente(idCliente)
);

-- Con azioni su DELETE e UPDATE
CREATE TABLE ordine (
    idOrdine INT PRIMARY KEY AUTO_INCREMENT,
    dataOrdine DATE NOT NULL,
    idCliente INT NOT NULL,
    FOREIGN KEY (idCliente) REFERENCES cliente(idCliente)
        ON DELETE CASCADE        -- Elimina ordini se cliente è
            eliminato
        ON UPDATE CASCADE        -- Aggiorna idCliente se cambia
);
```

Listing 6.4: FOREIGN KEY

UNIQUE

Garantisce che i valori di un attributo siano unici (ma possono essere NULL).

```
CREATE TABLE utente (
    idUtente INT PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL
);
```

Listing 6.5: Vincolo UNIQUE

NOT NULL

L'attributo deve avere sempre un valore.

```
CREATE TABLE prodotto (
    idProdotto INT PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    descrizione TEXT,           -- Può essere NULL
    prezzo DECIMAL(10, 2) NOT NULL
);
```

Listing 6.6: Vincolo NOT NULL

CHECK

Garantisce che i valori soddisfino una condizione.

```
CREATE TABLE conto (
    idConto INT PRIMARY KEY,
    saldo DECIMAL(10, 2),
    -- Saldo non può essere negativo
    CHECK (saldo >= 0)
);

CREATE TABLE corso (
    idCorso INT PRIMARY KEY,
    nome VARCHAR(100),
    maxStudenti INT,
    minStudenti INT,
    -- Minimo <= Massimo
    CHECK (minStudenti <= maxStudenti)
);
```

Listing 6.7: Vincolo CHECK

DEFAULT

Assegna un valore di default se non specificato.

```sql
CREATE TABLE articolo (
    idArticolo INT PRIMARY KEY,
    titolo VARCHAR(200) NOT NULL,
    stato VARCHAR(20) DEFAULT 'Bozza',
    dataCreazione DATETIME DEFAULT CURRENT_TIMESTAMP,
    visualizzazioni INT DEFAULT 0
);
```

Listing 6.8: Vincolo DEFAULT

AUTO$_I NCREMENT$

Genera automaticamente valori incrementali per una colonna (tipicamente chiave primaria).

```sql
CREATE TABLE cliente (
    idCliente INT PRIMARY KEY AUTO_INCREMENT,
    nome VARCHAR(100),
    email VARCHAR(100)
);
-- Primo cliente inserito avrà idCliente=1, secondo idCliente=2, etc.
```

Listing 6.9: AUTO$_I NCREMENT$

6.3 ALTER TABLE

Il comando ALTER TABLE modifica la struttura di una tabella esistente.

6.3.1 Aggiungere una colonna

```sql
ALTER TABLE cliente ADD COLUMN telefono VARCHAR(20);

-- Aggiungere con vincoli
ALTER TABLE cliente ADD COLUMN dataNascita DATE DEFAULT NULL;

-- Aggiungere in posizione specifica
ALTER TABLE cliente ADD COLUMN indirizzo VARCHAR(200) AFTER nome;
```

Listing 6.10: Aggiungere una colonna

6.3.2 Modificare una colonna

```sql
-- Cambiare tipo di dato
ALTER TABLE cliente MODIFY COLUMN telefono VARCHAR(50);

-- Cambiare nome della colonna
ALTER TABLE cliente CHANGE COLUMN telefono cellulare VARCHAR(20);

```

```sql
-- Aggiungere vincolo NOT NULL
ALTER TABLE cliente MODIFY COLUMN email VARCHAR(100) NOT NULL;
```

Listing 6.11: Modificare una colonna

6.3.3 Eliminare una colonna

```sql
ALTER TABLE cliente DROP COLUMN fax;
```

Listing 6.12: Eliminare una colonna

6.3.4 Aggiungere vincoli

```sql
-- Aggiungere PRIMARY KEY
ALTER TABLE cliente ADD PRIMARY KEY (idCliente);

-- Aggiungere UNIQUE
ALTER TABLE cliente ADD CONSTRAINT uq_email UNIQUE (email);

-- Aggiungere CHECK
ALTER TABLE cliente ADD CONSTRAINT ck_eta CHECK (eta >= 18);

-- Aggiungere FOREIGN KEY
ALTER TABLE ordine ADD CONSTRAINT fk_cliente
    FOREIGN KEY (idCliente) REFERENCES cliente(idCliente);
```

Listing 6.13: Aggiungere vincoli

6.3.5 Rinominare una tabella

```sql
ALTER TABLE cliente RENAME TO customer;
```

Listing 6.14: Rinominare una tabella

6.4 DROP

Il comando DROP elimina oggetti del database.

6.4.1 DROP TABLE

```sql
-- Elimina la tabella cliente
DROP TABLE cliente;

-- Elimina solo se esiste (evita errore se non esiste)
```

```sql
DROP TABLE IF EXISTS cliente;

-- Eliminare più tabelle
DROP TABLE ordine, prodotto, cliente;
```

Listing 6.15: Eliminare una tabella

Attenzione: DROP è irreversibile

DROP TABLE elimina completamente la tabella e tutti i dati. Non c'è possibilità di recuperarli (a meno che non sia stato fatto un backup). Usa sempre IF EXISTS per sicurezza.

6.4.2 DROP COLUMN

```sql
ALTER TABLE cliente DROP COLUMN numeroFax;
```

Listing 6.16: Eliminare una colonna

6.4.3 DROP INDEX

```sql
DROP INDEX idx_email ON cliente;
```

Listing 6.17: Eliminare un indice

6.5 CREATE INDEX

6.5.1 Creare indici

```sql
-- Indice semplice
CREATE INDEX idx_email ON cliente(email);

-- Indice composito (su più colonne)
CREATE INDEX idx_cliente_data ON ordine(idCliente, dataOrdine);

-- Indice UNIQUE
CREATE UNIQUE INDEX idx_codice_prodotto ON prodotto(codice);
```

Listing 6.18: Creare indici

6.5.2 Eliminare indici

```sql
DROP INDEX idx_email ON cliente;
```

Listing 6.19: Eliminare indici

6.6 Esempio Completo: Schema Biblioteca

```sql
-- Tabella Autori
CREATE TABLE autore (
    idAutore INT PRIMARY KEY AUTO_INCREMENT,
    nome VARCHAR(100) NOT NULL,
    cognome VARCHAR(100) NOT NULL,
    nazionalita VARCHAR(50),
    dataAnniversario DATE
);

-- Tabella Libri
CREATE TABLE libro (
    idLibro INT PRIMARY KEY AUTO_INCREMENT,
    titolo VARCHAR(200) NOT NULL,
    idAutore INT NOT NULL,
    anno_pubblicazione INT CHECK (anno_pubblicazione > 1000),
    genere VARCHAR(50),
    prezzo DECIMAL(10, 2) DEFAULT 0,
    quantita_disponibile INT DEFAULT 0,
    FOREIGN KEY (idAutore) REFERENCES autore(idAutore)
        ON DELETE RESTRICT
        ON UPDATE CASCADE
);

-- Tabella Membri
CREATE TABLE membro (
    idMembro INT PRIMARY KEY AUTO_INCREMENT,
    nome VARCHAR(100) NOT NULL,
    cognome VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE,
    dataIscrizione DATE DEFAULT CURDATE(),
    stato ENUM('Attivo', 'Sospeso') DEFAULT 'Attivo'
);

-- Tabella Prestiti
CREATE TABLE prestito (
    idPrestito INT PRIMARY KEY AUTO_INCREMENT,
    idLibro INT NOT NULL,
    idMembro INT NOT NULL,
    dataPrestito DATE DEFAULT CURDATE(),
    dataReso DATE,
    FOREIGN KEY (idLibro) REFERENCES libro(idLibro),
    FOREIGN KEY (idMembro) REFERENCES membro(idMembro),
    CHECK (dataReso IS NULL OR dataReso >= dataPrestito)
);

-- Creare indici per query frequenti
CREATE INDEX idx_libro_autore ON libro(idAutore);
CREATE INDEX idx_prestito_libro ON prestito(idLibro);
CREATE INDEX idx_prestito_membro ON prestito(idMembro);
CREATE INDEX idx_membro_email ON membro(email);
```

Listing 6.20: Schema completo per una biblioteca

Riepilogo concetti chiave

Concetti fondamentali

- **CREATE TABLE** definisce la struttura della tabella con attributi e vincoli
- I **vincoli** (PRIMARY KEY, FOREIGN KEY, UNIQUE, CHECK, NOT NULL, DEFAULT) garantiscono integrità
- **ALTER TABLE** modifica tabelle esistenti (aggiunta, modifica, eliminazione colonne)
- **DROP** elimina tabelle, colonne, indici (irreversibile!)
- Scegliere **tipi di dati appropriati** ottimizza spazio e velocità
- **Indici** accelerano ricerche e devono essere creati strategicamente

Esercizi

1. Crea una tabella `dipendente` con: idDipendente (INT, chiave primaria, auto-incremento), nome (VARCHAR), cognome (VARCHAR), stipendio (DECIMAL, >= 0), dipartimento (VARCHAR, default 'Non assegnato'), dataAssunzione (DATE, default data odierna).
2. Modifica la tabella precedente aggiungendo una colonna `email` (VARCHAR, UNIQUE, NOT NULL).
3. Crea una tabella `reparto` e un'associazione con `dipendente` tramite FOREIGN KEY. La cancellazione di un reparto non deve eliminare i dipendenti.
4. Progetta l'intero schema DDL per un sistema ospedaliero con: Ospedali, Reparti, Pazienti, Medici, Appuntamenti, con tutti i vincoli appropriati.
5. Crea indici appropriati per le seguenti query frequenti:

```
SELECT * FROM ordine WHERE idCliente = ? AND stato = 'Spedito';
SELECT * FROM prodotto WHERE categoria = ?;
SELECT * FROM cliente WHERE email = ?;
```

Capitolo 7

SQL DML - SELECT

Introduzione

DML (Data Manipulation Language) è il linguaggio SQL per interrogare e manipolare i dati. Il comando SELECT è il più importante e potente di DML, permettendo di recuperare dati da una o più tabelle con filtri, ordinamenti e proiezioni sofisticate.

Obiettivi di apprendimento

- Scrivere query SELECT di base
- Filtrare dati con WHERE
- Ordinare risultati con ORDER BY
- Limitare risultati con LIMIT
- Usare funzioni di aggregazione (COUNT, SUM, AVG, MIN, MAX)
- Raggruppare dati con GROUP BY
- Filtrare gruppi con HAVING
- Usare alias per colonne e tabelle
- Scrivere query sub-SELECT (subquery)

7.1 Sintassi SELECT di Base

7.1.1 Sintassi generale

```
SELECT [DISTINCT] colonna1, colonna2, ...
FROM tabella1
WHERE condizioni
ORDER BY colonna [ASC|DESC]
LIMIT numero_righe;
```

Listing 7.1: Sintassi SELECT

7.1.2 Selezionare tutte le colonne

```sql
-- Seleziona tutte le colonne
SELECT * FROM cliente;

-- Limita ai primi 10 risultati
SELECT * FROM cliente LIMIT 10;
```

Listing 7.2: Selezionare tutte le colonne

7.1.3 Selezionare colonne specifiche

```sql
SELECT idCliente, nome, cognome, email FROM cliente;
```

Listing 7.3: Selezionare colonne specifiche

7.1.4 Alias per colonne

```sql
-- Renominare colonne nel risultato
SELECT
    idCliente AS ID,
    nome AS Nome,
    email AS 'Indirizzo Email'
FROM cliente;
```

Listing 7.4: Alias per colonne

7.2 WHERE - Filtri

Il WHERE filtra le righe in base a condizioni.

7.2.1 Operatori di confronto

```sql
-- Uguaglianza
SELECT * FROM cliente WHERE stato = 'Attivo';

-- Diverso
SELECT * FROM cliente WHERE stato != 'Inattivo';
SELECT * FROM cliente WHERE stato <> 'Inattivo';

-- Maggiore/minore
SELECT * FROM ordine WHERE totale > 100.00;
SELECT * FROM ordine WHERE totale <= 50.00;

-- BETWEEN (inclusivo)
SELECT * FROM ordine WHERE dataOrdine BETWEEN '2023-01-01' AND '
    2023-12-31';
```

Listing 7.5: Operatori di confronto

7.2.2 Operatori logici

```sql
-- AND: tutte le condizioni devono essere vere
SELECT * FROM cliente
WHERE stato = 'Attivo' AND città = 'Milano';

-- OR: almeno una condizione deve essere vera
SELECT * FROM cliente
WHERE città = 'Milano' OR città = 'Roma';

-- NOT: negazione
SELECT * FROM cliente
WHERE NOT stato = 'Inattivo';
```

Listing 7.6: Operatori logici AND

7.2.3 IN e NOT IN

```sql
-- IN: verifica se il valore è in un elenco
SELECT * FROM ordine
WHERE stato IN ('Spedito', 'Consegnato');

-- NOT IN: verifica se il valore NON è in un elenco
SELECT * FROM ordine
WHERE stato NOT IN ('Cancellato', 'Rifiutato');
```

Listing 7.7: Operatori IN e NOT IN

7.2.4 LIKE - Pattern matching

```sql
-- % rappresenta 0 o più caratteri, _ rappresenta un carattere singolo

-- Nome che inizia con 'M'
SELECT * FROM cliente WHERE nome LIKE 'M%';

-- Nome che finisce con 'o'
SELECT * FROM cliente WHERE nome LIKE '%o';

-- Email che contiene 'gmail'
SELECT * FROM cliente WHERE email LIKE '%gmail%';

-- Parola di esattamente 5 caratteri
SELECT * FROM cliente WHERE nome LIKE '_____';

-- Case-insensitive (dipende dal collation)
SELECT * FROM cliente WHERE nome LIKE 'mario%';
```

Listing 7.8: LIKE per pattern matching

7.2.5 IS NULL e IS NOT NULL

```sql
-- Valori NULL
SELECT * FROM cliente WHERE telefono IS NULL;

-- Valori non NULL
SELECT * FROM cliente WHERE telefono IS NOT NULL;

-- Nota: = NULL non funziona! Usare IS NULL
SELECT * FROM cliente WHERE telefono = NULL;  -- Restituisce 0 righe!
```

Listing 7.9: NULL in WHERE

7.3 ORDER BY - Ordinamento

Ordina i risultati in base a una o più colonne.

```sql
-- Ordinamento ascendente (default)
SELECT * FROM ordine ORDER BY dataOrdine ASC;

-- Ordinamento discendente
SELECT * FROM ordine ORDER BY dataOrdine DESC;

-- Ordinamento su più colonne
SELECT * FROM cliente
ORDER BY città ASC, cognome ASC, nome ASC;

-- Ordinamento per numero di colonna (sconsigliato)
SELECT nome, cognome, email FROM cliente
ORDER BY 2, 1;  -- Ordina per colonna 2 (cognome), poi colonna 1 (nome
    )
```

Listing 7.10: ORDER BY

7.4 LIMIT - Limitazione Risultati

Limita il numero di righe restituite.

```sql
-- Primi 10 risultati
SELECT * FROM cliente LIMIT 10;

-- 10 risultati a partire dal 20° (offset=20, limit=10)
SELECT * FROM cliente LIMIT 20, 10;

-- Sintassi alternativa
```

```sql
SELECT * FROM cliente LIMIT 10 OFFSET 20;

-- Ultimi 5 clienti registrati
SELECT * FROM cliente
ORDER BY dataRegistrazione DESC
LIMIT 5;
```

Listing 7.11: LIMIT

7.5 DISTINCT - Eliminare Duplicati

Restituisce solo valori unici.

```sql
-- Tutte le città
SELECT DISTINCT città FROM cliente;

-- Combinazione unica di città e stato
SELECT DISTINCT città, stato FROM cliente;

-- Contare città distinte
SELECT COUNT(DISTINCT città) FROM cliente;
```

Listing 7.12: DISTINCT

7.6 Funzioni di Stringa

```sql
-- Lunghezza stringa
SELECT nome, LENGTH(nome) AS lunghezza FROM cliente;

-- Concatenazione
SELECT CONCAT(nome, ' ', cognome) AS nomeCompleto FROM cliente;

-- Maiuscolo/minuscolo
SELECT UPPER(nome), LOWER(cognome) FROM cliente;

-- Substring
SELECT SUBSTRING(email, 1, POSITION('@' IN email) - 1) AS username
    FROM cliente;

-- Sostituzione
SELECT REPLACE(email, 'gmail.com', 'email.com') FROM cliente;

-- Trim (rimuove spazi)
SELECT TRIM(nome) FROM cliente;
```

Listing 7.13: Funzioni di stringa

7.7 Funzioni di Data

```sql
-- Data odierna
SELECT CURDATE() AS oggi;
SELECT CURRENT_DATE() AS oggi;

-- Data e ora attuali
SELECT NOW() AS adesso;
SELECT CURRENT_TIMESTAMP() AS adesso;

-- Estrarre parti della data
SELECT
    YEAR(dataOrdine) AS anno,
    MONTH(dataOrdine) AS mese,
    DAY(dataOrdine) AS giorno
FROM ordine;

-- Differenza tra date (in giorni)
SELECT
    idOrdine,
    DATEDIFF(CURDATE(), dataOrdine) AS giorni_passati
FROM ordine;

-- Aggiungere giorni a una data
SELECT DATE_ADD(CURDATE(), INTERVAL 7 DAY) AS tra_una_settimana;
SELECT DATE_ADD(CURDATE(), INTERVAL 1 MONTH) AS tra_un_mese;

-- Formato data
SELECT DATE_FORMAT(dataOrdine, '%d/%m/%Y') FROM ordine;
```

Listing 7.14: Funzioni di data

7.8 Funzioni Numeriche

```sql
-- Arrotondamento
SELECT ROUND(prezzo, 2) FROM prodotto;

-- Arrotondamento per difetto/eccesso
SELECT FLOOR(prezzo), CEIL(prezzo) FROM prodotto;

-- Valore assoluto
SELECT ABS(-10);

-- Potenza
SELECT POWER(2, 3);  -- 2^3 = 8

-- Radice quadrata
SELECT SQRT(16);  -- 4

-- Modulo (resto divisione)
SELECT MOD(10, 3);  -- 1
```

Listing 7.15: Funzioni numeriche

7.9 Subquery (Sub-SELECT)

Una subquery è una query dentro un'altra query.

```sql
-- Trovare clienti che hanno fatto ordini superiori alla media
SELECT *
FROM cliente
WHERE idCliente IN (
    SELECT idCliente FROM ordine WHERE totale > (
        SELECT AVG(totale) FROM ordine
    )
);

-- Trovare prodotti con prezzo superiore al prezzo medio della
    categoria
SELECT *
FROM prodotto p1
WHERE prezzo > (
    SELECT AVG(prezzo) FROM prodotto p2
    WHERE p2.categoria = p1.categoria
);
```

Listing 7.16: Subquery in WHERE

```sql
-- Aggiungere il numero di ordini accanto a ogni cliente
SELECT
    idCliente,
    nome,
    (SELECT COUNT(*) FROM ordine o WHERE o.idCliente = c.idCliente) AS
        numOrdini
FROM cliente c;
```

Listing 7.17: Subquery in SELECT

7.10 Esempio Completo

```sql
-- Trovare i 5 clienti più attivi del 2023
-- ordinati per numero di ordini decrescente
SELECT
    c.idCliente,
    c.nome,
    c.cognome,
    COUNT(o.idOrdine) AS numOrdini,
    SUM(o.totale) AS totalSpeso
FROM cliente c
LEFT JOIN ordine o ON c.idCliente = o.idCliente
    AND YEAR(o.dataOrdine) = 2023
WHERE c.stato = 'Attivo'
GROUP BY c.idCliente, c.nome, c.cognome
ORDER BY numOrdini DESC
LIMIT 5;
```

Listing 7.18: Query complessa

Riepilogo concetti chiave

Concetti fondamentali

- **SELECT** recupera dati da tabelle
- **WHERE** filtra righe in base a condizioni
- **ORDER BY** ordina risultati (ASC/DESC)
- **LIMIT** limita il numero di righe
- **DISTINCT** elimina duplicati
- **Funzioni** (stringa, data, numero) trasformano i dati
- **Subquery** permettono query annidate
- Combinare clausole per query sofisticate

Esercizi

1. Scrivi una query per trovare tutti i clienti della città di 'Milano' registrati negli ultimi 30 giorni, ordinati per cognome.
2. Scrivi una query per elencare prodotti il cui prezzo è tra 50 e 100 euro, escludendo la categoria 'Saldi'.
3. Conta quanti ordini sono stati fatti da cliente 'id=5' e mostra il numero come colonna 'NumOrdini'.
4. Usa una subquery per trovare i clienti che hanno speso più della media totale di tutti gli ordini.
5. Scrivi una query che mostra, per ogni cliente: nome, cognome, numero di ordini effettuati, importo totale speso, ordinato per importo totale decrescente. Usa LEFT JOIN se necessario.
6. Estrai da una tabella 'articolo' gli articoli con titolo che contiene 'Database' (case-insensitive), creati nel 2023 o successivamente, ordinati per data di creazione.

Capitolo 8

SQL DML - Modifiche Dati

Introduzione

Oltre a leggere dati con SELECT, DML (Data Manipulation Language) permette di inserire nuovi dati (INSERT), aggiornare dati esistenti (UPDATE) e eliminare dati (DELETE). Questo capitolo presenta questi tre comandi essenziali con attenzione particolare alla preservazione dell'integrità dei dati.

Obiettivi di apprendimento

- Inserire nuove righe con INSERT
- Inserire dati da altre tabelle
- Aggiornare dati esistenti con UPDATE
- Eliminare dati con DELETE
- Gestire transazioni per operazioni sicure
- Comprendere i vincoli e le loro violazioni
- Usare controlli e prevenire errori

8.1 INSERT - Inserimento Dati

Il comando INSERT aggiunge nuove righe a una tabella.

8.1.1 Sintassi base

```
INSERT INTO tabella (colonna1, colonna2, ...)
VALUES (valore1, valore2, ...);
```

Listing 8.1: Sintassi INSERT

8.1.2 Inserire una riga

```sql
INSERT INTO cliente (idCliente, nome, cognome, email)
VALUES (1, 'Mario', 'Rossi', 'mario@email.com');
```

Listing 8.2: Inserimento semplice

8.1.3 Inserire senza specificare colonne

Se ometti le colonne, devi fornire valori per TUTTE le colonne nell'ordine definito.

```sql
-- Se la tabella ha: idCliente, nome, cognome, email, città
INSERT INTO cliente
VALUES (2, 'Luigi', 'Bianchi', 'luigi@email.com', 'Roma');
```

Listing 8.3: Inserimento senza specificare colonne

8.1.4 Inserire più righe

```sql
INSERT INTO cliente (idCliente, nome, cognome, email)
VALUES
    (3, 'Anna', 'Verdi', 'anna@email.com'),
    (4, 'Giovanni', 'Gialli', 'giovanni@email.com'),
    (5, 'Maria', 'Neri', 'maria@email.com');
```

Listing 8.4: Inserimento multiplo

8.1.5 Inserire da SELECT

```sql
-- Copiare dati da un'altra tabella
INSERT INTO cliente_backup
SELECT * FROM cliente WHERE città = 'Milano';

-- Inserire dati trasformati
INSERT INTO cliente_attivi (nome, cognome, città)
SELECT nome, cognome, città FROM cliente WHERE stato = 'Attivo';
```

Listing 8.5: INSERT ... SELECT

8.1.6 Gestione di valori NULL

```sql
-- Valori NULL per colonne opzionali
INSERT INTO cliente (idCliente, nome, cognome, email, telefono)
VALUES (6, 'Paolo', 'Rossi', 'paolo@email.com', NULL);

-- Omettere colonna opzionale (diventa NULL)
```

```sql
INSERT INTO cliente (idCliente, nome, cognome, email)
VALUES (7, 'Laura', 'Bianchi', 'laura@email.com');
-- Il campo telefono sarà NULL
```

Listing 8.6: Inserire NULL

8.1.7 Auto-increment

```sql
-- Se la colonna è AUTO_INCREMENT, omettere il valore
INSERT INTO cliente (nome, cognome, email)
VALUES ('Marco', 'Verdi', 'marco@email.com');
-- idCliente verrà assegnato automaticamente

-- Recuperare l'ID appena inserito
INSERT INTO cliente (nome, cognome, email)
VALUES ('Silvia', 'Gialli', 'silvia@email.com');
SELECT LAST_INSERT_ID();  -- Restituisce l'ID appena assegnato
```

Listing 8.7: AUTO$_I NCREMENT$

Nota: Gestire AUTO$_I NCREMENT$

Per auto-increment, non inserire esplicitamente il valore. Lascia che il DBMS lo generi. Questo garantisce unicità.

8.1.8 Errori comuni di INSERT

Errore: Violazione vincolo PRIMARY KEY

```sql
-- Errore: tentare di inserire un id già esistente
INSERT INTO cliente (idCliente, nome, cognome, email)
VALUES (1, 'Antonio', 'Neri', 'antonio@email.com');
-- Errore: Duplicate entry for primary key
```

Errore: Violazione FOREIGN KEY

```sql
-- Errore: tentare di inserire un ordine per un cliente
   inesistente
INSERT INTO ordine (idOrdine, idCliente, dataOrdine)
VALUES (1, 999, CURDATE());
-- Errore: Foreign key constraint fails (se cliente 999 non
   esiste)
```

8.2 UPDATE - Aggiornamento Dati

Il comando UPDATE modifica dati esistenti.

8.2.1 Sintassi base

```sql
UPDATE tabella
SET colonna1 = valore1, colonna2 = valore2, ...
WHERE condizione;
```

Listing 8.8: Sintassi UPDATE

Attenzione: WHERE è obbligatorio!

Senza WHERE, TUTTI i record verranno aggiornati! Usa sempre WHERE per evitare danni.

8.2.2 Aggiornare una colonna

```sql
-- Aggiornare l'email di un cliente
UPDATE cliente
SET email = 'mario.rossi@newemail.com'
WHERE idCliente = 1;
```

Listing 8.9: UPDATE semplice

8.2.3 Aggiornare più colonne

```sql
-- Aggiornare nome e cognome
UPDATE cliente
SET nome = 'Marco', cognome = 'Verdi'
WHERE idCliente = 1;
```

Listing 8.10: UPDATE multiplo

8.2.4 Aggiornare con condizioni complesse

```sql
-- Aumentare lo sconto del 10% per i clienti di Milano attivi
UPDATE cliente
SET sconto = sconto * 1.10
WHERE città = 'Milano' AND stato = 'Attivo';

-- Spostare ordini cancellati agli archivi
UPDATE ordine
SET stato = 'Archiviato'
WHERE stato = 'Cancellato'
```

```
10      AND dataOrdine < DATE_SUB(CURDATE(), INTERVAL 1 YEAR);
```

Listing 8.11: UPDATE con WHERE complesso

8.2.5 Aggiornare da SELECT

```
1  -- Aumentare prezzo di prodotti basato su categoria
2  UPDATE prodotto
3  SET prezzo = prezzo * 1.05
4  WHERE categoria IN (
5      SELECT categoria FROM categoria_promozionale
6  );
7
8  -- Aggiornare il numero di ordini di ogni cliente
9  UPDATE cliente c
10 SET numOrdini = (SELECT COUNT(*) FROM ordine o WHERE o.idCliente = c.
       idCliente);
```

Listing 8.12: UPDATE da subquery

8.2.6 Aggiornare con espressioni

```
1  -- Aumentare il prezzo del 10% arrotondato a 2 decimali
2  UPDATE prodotto
3  SET prezzo = ROUND(prezzo * 1.10, 2)
4  WHERE categoria = 'Elettronica';
5
6  -- Aggiornare data ultima modifica a data odierna
7  UPDATE cliente
8  SET dataUltimaModifica = NOW()
9  WHERE idCliente = 1;
10
11 -- Concatenare stringhe
12 UPDATE cliente
13 SET nomePieno = CONCAT(nome, ' ', cognome)
14 WHERE nomePieno IS NULL;
```

Listing 8.13: UPDATE con espressioni

8.3 DELETE - Eliminazione Dati

Il comando DELETE rimuove righe da una tabella.

8.3.1 Sintassi base

```
1  DELETE FROM tabella
2  WHERE condizione;
```

Listing 8.14: Sintassi DELETE

Attenzione: WHERE è obbligatorio!

Senza WHERE, TUTTI i record verranno eliminati! Usa sempre WHERE.

8.3.2 Eliminare una riga

```sql
-- Eliminare un cliente specifico
DELETE FROM cliente
WHERE idCliente = 1;
```

Listing 8.15: DELETE singola riga

8.3.3 Eliminare con condizioni

```sql
-- Eliminare ordini cancellati più di un anno fa
DELETE FROM ordine
WHERE stato = 'Cancellato'
    AND dataOrdine < DATE_SUB(CURDATE(), INTERVAL 1 YEAR);

-- Eliminare clienti inattivi senza ordini
DELETE FROM cliente
WHERE stato = 'Inattivo'
    AND NOT EXISTS (SELECT 1 FROM ordine WHERE ordine.idCliente =
        cliente.idCliente);
```

Listing 8.16: DELETE con WHERE

8.3.4 Errori di integrità referenziale

Errore: Violazione FOREIGN KEY su DELETE

```sql
-- Se ordine ha FOREIGN KEY verso cliente
-- e non è definito ON DELETE CASCADE,
-- non puoi eliminare un cliente che ha ordini
DELETE FROM cliente WHERE idCliente = 5;
-- Errore: Cannot delete: foreign key constraint fails
```

8.3.5 Soluzioni per vincoli di integrità

```sql
-- Opzione 1: Eliminare prima gli ordini dipendenti
DELETE FROM ordine WHERE idCliente = 5;
```

```sql
DELETE FROM cliente WHERE idCliente = 5;

-- Opzione 2: Se FK ha ON DELETE CASCADE
-- Eliminare il cliente elimina automaticamente gli ordini
DELETE FROM cliente WHERE idCliente = 5;

-- Opzione 3: Soft delete (marcazione come inattivo)
UPDATE cliente SET stato = 'Inattivo' WHERE idCliente = 5;
-- Permette recupero successivo se necessario
```

Listing 8.17: Soluzione: Gestire dipendenze

8.4 Transazioni ACID

Una transazione raggruppa più operazioni SQL in un'unità atomica.

```sql
-- Iniziare transazione
START TRANSACTION;

-- Operazioni
INSERT INTO cliente (nome, cognome, email) VALUES ('Mario', 'Rossi', '
    mario@email.com');
INSERT INTO ordine (idCliente, dataOrdine, totale) VALUES (
    LAST_INSERT_ID(), CURDATE(), 150.00);

-- Se tutto ok, confermare
COMMIT;

-- Se c'è un errore, annullare
-- ROLLBACK;
```

Listing 8.18: Transazioni

8.4.1 Scenari transazionali

```sql
START TRANSACTION;

-- Prelievo dal conto A
UPDATE conto SET saldo = saldo - 100 WHERE idConto = 1;

-- Deposito al conto B
UPDATE conto SET saldo = saldo + 100 WHERE idConto = 2;

-- Se entrambi ok, commit. Se uno fallisce, tutto viene rollback
COMMIT;
```

8.5 Disabilitare Vincoli (Avanzato)

Talvolta è necessario disabilitare temporaneamente i vincoli per operazioni bulk.

-

b Disabilitare i vincoli di chiave esterna (MySQL) SET FOREIGN$_KEY_CHECKS = 0$;

– Eseguire operazioni bulk DELETE FROM ordine; DELETE FROM cliente;

– Riabilitare vincoli SET FOREIGN$_KEY_CHECKS = 1$;

Attenzione: Disabilitare vincoli è rischioso

Disabilitando i vincoli, il DBMS non controlla l'integrità referenziale. Usa solo se sai cosa fai e riabilita subito dopo.

Riepilogo concetti chiave

Concetti fondamentali

- **INSERT** aggiunge nuove righe. Usa AUTO$_INCREMENT per ID$.
- **INSERT ... SELECT** copia dati da altre tabelle
- **UPDATE** modifica dati. WHERE specifica quali righe.
- **DELETE** elimina righe. WHERE è essenziale!
- I **vincoli** (PRIMARY KEY, FOREIGN KEY) proteggono l'integrità
- Le **transazioni** (BEGIN, COMMIT, ROLLBACK) garantiscono atomicità
- Sempre testare con SELECT prima di UPDATE/DELETE in massa

Esercizi

1. Inserisci 5 clienti nella tabella cliente con nomi, cognomi e email distinti.
2. Usa INSERT ... SELECT per copiare tutti gli ordini del 2023 in una tabella ordini_2023.
3. Aggiorna il prezzo di tutti i prodotti della categoria 'Elettronica' con un aumento del 15%.
4. Scrivi una transazione che: a) inserisce un nuovo cliente, b) inserisce un ordine per quel cliente, c) aggiorna il totale speso del cliente.
5. Elimina tutti gli ordini di un cliente che hanno stato 'Cancellato' e non sono stati modificati da più di 6 mesi.
6. Crea uno script che archiva ordini vecchi: copia ordini più vecchi di 2 anni in una tabella archivio_ordini, poi elimina gli originali. Usa una transazione!

Capitolo 9

SQL JOIN - Combinazione Tabelle

Introduzione

Il JOIN è il meccanismo SQL per combinare dati da più tabelle correlate tramite chiavi esterne. Questo capitolo presenta i diversi tipi di JOIN (INNER, LEFT, RIGHT, FULL OUTER) con diagrammi visivi e esempi pratici per comprendere come e quando usarli.

Obiettivi di apprendimento

- Comprendere il concetto di JOIN e la necessità di combinare tabelle
- Scrivere query con INNER JOIN
- Scrivere query con LEFT JOIN e RIGHT JOIN
- Scrivere query con FULL OUTER JOIN
- Usare CROSS JOIN
- Usare alias per tabelle nei JOIN
- Eseguire JOIN su più di due tabelle
- Evitare cartesiani (combinazioni errate)

9.1 Concetti di Base

9.1.1 Perché JOIN?

Nel modello relazionale, i dati sono distribuiti su più tabelle. Per rispondere a domande che coinvolgono dati di più tabelle, devi combinarle con JOIN.

Esempio: Domanda che richiede JOIN

"Quali clienti hanno fatto ordini nel 2023?"
Per rispondere devo:

1. Trovare ordini con dataOrdine nel 2023 (tabella ordine)
2. Per ogni ordine, trovare il cliente corrispondente (tabella cliente)
3. Mostrare i dati del cliente

Questo richiede un JOIN tra ordine e cliente.

9.1.2 Condizione JOIN

Un JOIN è basato su una condizione che specifica come abbinare le righe. Tipicamente è un'uguaglianza tra chiave primaria e chiave esterna.

9.1.3 Panoramica Tipi di JOIN

Prima di esaminare ogni tipo in dettaglio, ecco una visione d'insieme dei principali tipi di JOIN con diagrammi Venn:

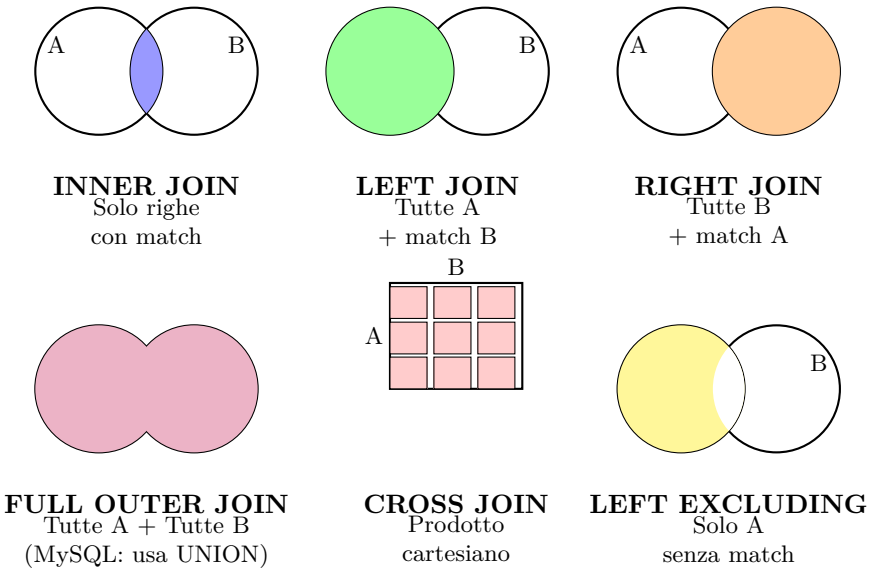


Figura 9.1: Tipi di JOIN: confronto visuale

Quale JOIN usare?

- **INNER JOIN**: quando vuoi solo le righe che hanno corrispondenza in entrambe le tabelle
- **LEFT JOIN**: quando vuoi tutte le righe della tabella principale (anche senza match)
- **RIGHT JOIN**: come LEFT JOIN ma al contrario (raramente usato, preferisci LEFT)
- **FULL OUTER JOIN**: quando vuoi tutte le righe di entrambe (raro, MySQL non supporta)
- **CROSS JOIN**: quando vuoi tutte le combinazioni possibili (attenzione: risultati enormi!)
- **LEFT EXCLUDING**: usa `LEFT JOIN + WHERE destra.chiave IS NULL` per trovare righe senza match

9.2 INNER JOIN

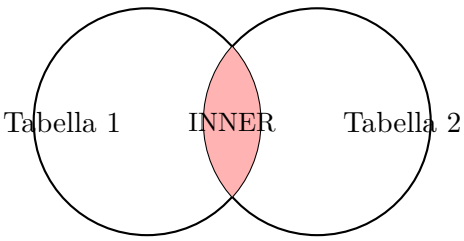
INNER JOIN restituisce solo le righe dove la condizione è vera in ENTRAMBE le tabelle.

9.2.1 Sintassi

```sql
SELECT colonne
FROM tabella1
INNER JOIN tabella2 ON tabella1.chiave = tabella2.chiave
WHERE ...;
```

Listing 9.1: Sintassi INNER JOIN

9.2.2 Diagramma Venn



INNER JOIN
Solo righe comuni

Figura 9.2: INNER JOIN: intersezione

9.2.3 Esempio

```sql
-- Trovare clienti e loro ordini
SELECT
    c.idCliente,
    c.nome,
    c.cognome,
    o.idOrdine,
    o.dataOrdine,
    o.totale
FROM cliente c
INNER JOIN ordine o ON c.idCliente = o.idCliente;
```

Listing 9.2: INNER JOIN

Risultato: Solo clienti che hanno fatto almeno un ordine.

9.2.4 INNER JOIN su condizioni multiple

```sql
-- Trovare ordini e i prodotti ordinati
SELECT
    o.idOrdine,
    p.nome AS nomeProdotto,
    op.quantita,
    op.prezzo_unitario
FROM ordine o
INNER JOIN ordine_prodotto op ON o.idOrdine = op.idOrdine
INNER JOIN prodotto p ON op.idProdotto = p.idProdotto;
```

Listing 9.3: INNER JOIN su più colonne

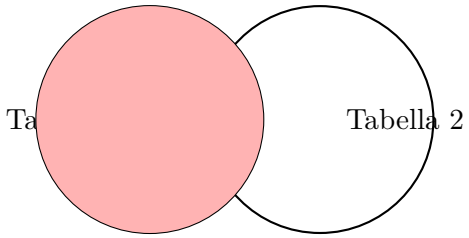
9.3 LEFT JOIN

LEFT JOIN restituisce TUTTE le righe della tabella di sinistra e le righe corrispondenti della tabella di destra. Se non c'è corrispondenza, le colonne di destra sono NULL.

9.3.1 Diagramma Venn

9.3.2 Esempio

```sql
-- Trovare TUTTI i clienti e i loro ordini
-- (inclusi clienti senza ordini)
SELECT
    c.idCliente,
    c.nome,
    c.cognome,
    COUNT(o.idOrdine) AS numOrdini,
    COALESCE(SUM(o.totale), 0) AS totalSpeso
FROM cliente c
```



LEFT JOIN
Tutte le righe di sinistra

Figura 9.3: LEFT JOIN: tutte le righe della tabella di sinistra

```
LEFT JOIN ordine o ON c.idCliente = o.idCliente
GROUP BY c.idCliente, c.nome, c.cognome;
```

Listing 9.4: LEFT JOIN

Risultato: Tutti i clienti, anche quelli senza ordini (che avranno numOrdini=0).

9.3.3 Usare LEFT JOIN per trovare record ASSENTI

```
-- Trovare clienti che NON hanno fatto ordini
SELECT
    c.idCliente,
    c.nome,
    c.cognome
FROM cliente c
LEFT JOIN ordine o ON c.idCliente = o.idCliente
WHERE o.idOrdine IS NULL;
```

Listing 9.5: LEFT JOIN per trovare mancanze

Filtrando WHERE o.idOrdine IS NULL trovi esattamente i clienti senza ordini.

9.4 RIGHT JOIN

RIGHT JOIN è l'inverso di LEFT JOIN: restituisce tutte le righe della tabella di destra e le corrispondenze della sinistra.

9.4.1 Esempio

```
-- Trovare TUTTI i prodotti e quante volte sono stati ordinati
SELECT
    p.idProdotto,
    p.nome,
    COUNT(op.idOrdine) AS volteOrdinato
FROM ordine_prodotto op
RIGHT JOIN prodotto p ON op.idProdotto = p.idProdotto
```

```
8 GROUP BY p.idProdotto, p.nome;
```

Listing 9.6: RIGHT JOIN

RIGHT JOIN può sempre essere convertito in LEFT JOIN invertendo l'ordine:

```
1 -- Equivalenti:
2 FROM ordine_prodotto op RIGHT JOIN prodotto p ...
3 FROM prodotto p LEFT JOIN ordine_prodotto op ...
```

Preferisci LEFT JOIN per coerenza (maggior leggibilità).

MySQL non supporta nativamente FULL OUTER JOIN. Usa un'alternativa con UNION:

```
1 SELECT * FROM tabella1 LEFT JOIN tabella2 ...
2 UNION
3 SELECT * FROM tabella1 RIGHT JOIN tabella2 ...;
```

9.4.2 Simulare FULL OUTER JOIN in MySQL

-

b Trovare tutti i dati cliente e ordine SELECT c.idCliente, c.nome, o.idOrdine, o.dataOrdine FROM cliente c LEFT JOIN ordine o ON c.idCliente = o.idCliente UNION SELECT c.idCliente, c.nome, o.idOrdine, o.dataOrdine FROM cliente c RIGHT JOIN ordine o ON c.idCliente = o.idCliente WHERE c.idCliente IS NULL;

9.5 CROSS JOIN

CROSS JOIN produce il prodotto cartesiano: combina ogni riga della prima tabella con ogni riga della seconda (senza condizione ON).

9.5.1 Sintassi

```
1 SELECT *
2 FROM tabella1
3 CROSS JOIN tabella2;
```

Listing 9.7: CROSS JOIN

9.5.2 Esempio pratico

```
1 -- Generare tutte le combinazioni possibili di prodotti e colori
2 SELECT
3     p.idProdotto,
4     p.nome,
```

```
    c.colore
FROM prodotto p
CROSS JOIN colore c;

-- Risultato: ogni prodotto combinato con ogni colore
-- Se 100 prodotti e 5 colori, risultato = 500 righe
```

Listing 9.8: CROSS JOIN pratico

Attenzione: CROSS JOIN può creare troppi dati!

Un CROSS JOIN tra tabelle grandi produce MOLTISSIME righe. Usa con cautela!

9.6 JOIN su Più Tabelle

Puoi combinare più di due tabelle con più JOIN.

9.6.1 Sintassi

```
SELECT colonne
FROM tabella1
JOIN tabella2 ON ...
JOIN tabella3 ON ...
JOIN tabella4 ON ...
WHERE ...;
```

Listing 9.9: JOIN multipli

9.6.2 Esempio: JOIN su 4 tabelle

```
-- Trovare cliente, suoi ordini, prodotti ordinati e categorie
SELECT
    c.nome,
    o.dataOrdine,
    p.nome AS nomeProdotto,
    op.quantita,
    cat.nome AS categoria
FROM cliente c
INNER JOIN ordine o ON c.idCliente = o.idCliente
INNER JOIN ordine_prodotto op ON o.idOrdine = op.idOrdine
INNER JOIN prodotto p ON op.idProdotto = p.idProdotto
INNER JOIN categoria cat ON p.idCategoria = cat.idCategoria
WHERE c.idCliente = 5
ORDER BY o.dataOrdine DESC;
```

Listing 9.10: JOIN multipli pratico

9.7 Self-Join

Un self-join è un join tra una tabella e se stessa. Utile per dati gerarchici o correlati.

```sql
-- Tabella: dipendente (idDipendente, nome, idDirigente)
-- Trovare ogni dipendente e il nome del suo dirigente

SELECT
    d.nome AS dipendente,
    m.nome AS dirigente
FROM dipendente d
LEFT JOIN dipendente m ON d.idDirigente = m.idDipendente;
```

Listing 9.11: Self-join

Nota: Usi alias (d e m) per distinguere i due ruoli della stessa tabella.

9.8 Alias per Tabelle

Gli alias rendono le query più leggibili, specialmente con JOIN.

```sql
-- Con alias (consigliato)
SELECT
    c.idCliente,
    c.nome,
    COUNT(o.idOrdine) AS numOrdini
FROM cliente c
LEFT JOIN ordine o ON c.idCliente = o.idCliente
GROUP BY c.idCliente, c.nome;

-- Senza alias (lungo e meno leggibile)
SELECT
    cliente.idCliente,
    cliente.nome,
    COUNT(ordine.idOrdine) AS numOrdini
FROM cliente
LEFT JOIN ordine ON cliente.idCliente = ordine.idCliente
GROUP BY cliente.idCliente, cliente.nome;
```

Listing 9.12: Alias per tabelle

9.9 Errori Comuni in JOIN

9.9.1 Dimenticare la condizione ON

Errore: Mancante condizione ON

```sql
-- Sbagliato: crea un CROSS JOIN involontario
SELECT * FROM cliente JOIN ordine;

-- Corretto
SELECT * FROM cliente JOIN ordine ON cliente.idCliente = ordine.
    idCliente;
```

9.9.2 Usare WHERE invece di ON

```sql
-- LEFT JOIN con filtro errato
SELECT *
FROM cliente c
LEFT JOIN ordine o
WHERE c.idCliente = o.idCliente;  -- Sbagliato!
-- Questo filtra dopo il JOIN, trasformando LEFT in INNER

-- Corretto
SELECT *
FROM cliente c
LEFT JOIN ordine o ON c.idCliente = o.idCliente
WHERE o.totale > 100;  -- Filtra dopo JOIN
```

Listing 9.13: ON vs WHERE

Riepilogo concetti chiave

Concetti fondamentali

- **INNER JOIN**: solo righe comuni
- **LEFT JOIN**: tutte le righe di sinistra + corrispondenze
- **RIGHT JOIN**: tutte le righe di destra + corrispondenze
- **FULL OUTER JOIN**: tutte le righe da entrambe (usa UNION in MySQL)
- **CROSS JOIN**: prodotto cartesiano (usa con cautela)
- **Self-join**: tabella con se stessa (usa alias)
- Usa **ON** per la condizione, WHERE per filtri successivi
- **Alias** rendono query complesse più leggibili

Esercizi

1. Scrivi un INNER JOIN per trovare ordini e i relativi clienti (nome, cognome) per ordini dell'anno 2023.
2. Usa LEFT JOIN per trovare TUTTI i clienti e il numero di ordini effettuati. Includi clienti senza ordini (che avranno count=0).
3. Usa LEFT JOIN con WHERE IS NULL per trovare clienti che NON hanno mai fatto ordini.
4. Crea un query con 3 JOIN: cliente, ordine, ordine_prodotto, prodotto. Mostra cliente, numero ordine e prodotto ordinato.
5. Usa CROSS JOIN per generare tutte le combinazioni di città clienti e categorie prodotti (utilità: analisi di mercato per ogni territorio).
6. Scrivi un self-join su una tabella dipendente per mostrare ogni dipendente con il nome del suo manager (dirigente).
7. Analizza questa query e spiega perché LEFT JOIN diventa INNER JOIN:

```sql
SELECT * FROM cliente c
LEFT JOIN ordine o ON c.idCliente = o.idCliente
WHERE o.dataOrdine > '2023-01-01';
```

Capitolo 10

SQL Aggregate - COUNT, SUM, AVG, MIN, MAX

Introduzione

Le funzioni di aggregazione calcolano valori singoli da un insieme di righe. Sono essenziali per analisi, report e sintesi dei dati. Questo capitolo presenta COUNT, SUM, AVG, MIN, MAX con GROUP BY e HAVING per elaborazioni complesse.

Obiettivi di apprendimento

- Comprendere le funzioni di aggregazione di base
- Usare COUNT per contare righe
- Usare SUM per somme
- Usare AVG, MIN, MAX per media, minimo, massimo
- Raggruppare dati con GROUP BY
- Filtrare gruppi con HAVING
- Combinare aggregazioni in query complesse
- Gestire NULL nelle aggregazioni
- Usare DISTINCT con funzioni di aggregazione

10.1 Funzioni di Aggregazione di Base

10.1.1 COUNT

COUNT conta il numero di righe o di valori non NULL.

```
1 -- Contare tutte le righe
2 SELECT COUNT(*) AS totalClienti FROM cliente;
3
```

```sql
-- Contare valori non NULL di una colonna
SELECT COUNT(email) AS clientiConEmail FROM cliente;

-- Contare valori distinti
SELECT COUNT(DISTINCT città) AS numCittà FROM cliente;

-- Contare con condizione
SELECT COUNT(*) AS clientiAttivi FROM cliente WHERE stato = 'Attivo';
```

Listing 10.1: COUNT

10.1.2 SUM

SUM calcola la somma di una colonna numerica.

```sql
-- Somma totale di tutti gli ordini
SELECT SUM(totale) AS totalVendite FROM ordine;

-- Somma con condizione
SELECT SUM(totale) AS vendite2023 FROM ordine WHERE YEAR(dataOrdine) =
    2023;

-- SUM restituisce NULL se nessuna riga (usa COALESCE)
SELECT COALESCE(SUM(totale), 0) AS totalOrdini FROM ordine WHERE stato
    = 'Cancellato';
```

Listing 10.2: SUM

10.1.3 AVG

AVG calcola la media di una colonna numerica.

```sql
-- Media prezzo prodotti
SELECT AVG(prezzo) AS prezzoMedio FROM prodotto;

-- Media ordini per cliente
SELECT AVG(totale) AS importoMedio FROM ordine WHERE stato = '
    Completato';

-- Media con rounding
SELECT ROUND(AVG(prezzo), 2) AS prezzoMedio FROM prodotto WHERE
    categoria = 'Elettronica';
```

Listing 10.3: AVG

10.1.4 MIN e MAX

MIN e MAX trovano il valore minimo e massimo.

```sql
-- Prezzo più basso e più alto
```

```sql
SELECT
    MIN(prezzo) AS prezzoMin,
    MAX(prezzo) AS prezzoMax
FROM prodotto;

-- Cliente con ordine più recente
SELECT MAX(dataOrdine) AS ultimoOrdine FROM ordine;

-- Data primo cliente registrato
SELECT MIN(dataRegistrazione) AS primoCliente FROM cliente;

-- Data cliente più recente e numero giorni da oggi
SELECT
    MAX(dataRegistrazione) AS clientePiuRecente,
    DATEDIFF(CURDATE(), MAX(dataRegistrazione)) AS giorniDaOggi
FROM cliente;
```

Listing 10.4: MIN e MAX

10.2 GROUP BY - Raggruppamento

GROUP BY raggruppa righe per uno o più attributi e applica funzioni di aggregazione a ogni gruppo.

10.2.1 Sintassi

```sql
SELECT colonna_gruppo, AGGREGAZIONE(colonna)
FROM tabella
WHERE condizioni_filtro
GROUP BY colonna_gruppo
ORDER BY ...;
```

Listing 10.5: Sintassi GROUP BY

10.2.2 Esempio semplice

```sql
-- Vendite per ogni cliente
SELECT
    idCliente,
    COUNT(*) AS numOrdini,
    SUM(totale) AS totalSpeso,
    AVG(totale) AS importoMedio
FROM ordine
GROUP BY idCliente;
```

Listing 10.6: GROUP BY semplice

10.2.3 GROUP BY su più colonne

```sql
-- Vendite per città e anno
SELECT
    YEAR(dataOrdine) AS anno,
    c.città,
    COUNT(*) AS numOrdini,
    SUM(o.totale) AS totalVendite
FROM ordine o
JOIN cliente c ON o.idCliente = c.idCliente
GROUP BY YEAR(o.dataOrdine), c.città
ORDER BY anno DESC, totalVendite DESC;
```

Listing 10.7: GROUP BY multiplo

10.2.4 GROUP BY con funzioni di data

```sql
-- Vendite per mese
SELECT
    DATE_TRUNC('month', dataOrdine) AS mese,
    COUNT(*) AS numOrdini,
    SUM(totale) AS totalVendite
FROM ordine
GROUP BY DATE_TRUNC('month', dataOrdine);

-- Alternativa MySQL (senza DATE_TRUNC)
SELECT
    DATE_FORMAT(dataOrdine, '%Y-%m') AS mese,
    COUNT(*) AS numOrdini,
    SUM(totale) AS totalVendite
FROM ordine
GROUP BY DATE_FORMAT(dataOrdine, '%Y-%m')
ORDER BY mese DESC;
```

Listing 10.8: GROUP BY con date

10.2.5 Errore comune: colonne non raggruppate

```sql
-- Errore in SQL stretto (MySQL con ONLY_FULL_GROUP_BY)
SELECT idCliente, nome, SUM(totale) FROM ordine
GROUP BY idCliente;
-- Errore: nome non è in GROUP BY

-- Corretto
SELECT idCliente, MAX(nome) AS nome, SUM(totale) FROM ordine
GROUP BY idCliente;
```

Tutte le colonne SELECT devono essere in GROUP BY o in una funzione di aggregazione.

S

10.3 HAVING - Filtro su Aggregazioni

HAVING filtra i GRUPPI basato su condizioni di aggregazione. È come WHERE, ma per i risultati di GROUP BY.

10.3.1 Sintassi

bËECT colonna$_gruppo, AGGREGAZIONE(colonna)FROMtabellaGROUPBYcolonna_gruppoHAVINGcon$

10.3.2 Differenza WHERE vs HAVING

WHERE vs HAVING

- **WHERE**: filtra le RIGHE prima di GROUP BY
- **HAVING**: filtra i GRUPPI dopo GROUP BY

10.3.3 Esempi

```sql
-- Clienti che hanno speso più di 1000 euro
SELECT
    idCliente,
    SUM(totale) AS totalSpeso
FROM ordine
GROUP BY idCliente
HAVING SUM(totale) > 1000
ORDER BY totalSpeso DESC;

-- Clienti con più di 5 ordini
SELECT
    idCliente,
    COUNT(*) AS numOrdini
FROM ordine
GROUP BY idCliente
HAVING COUNT(*) > 5;

-- Categorie con prezzo medio superiore a 50 euro
SELECT
    categoria,
    AVG(prezzo) AS prezzoMedio,
    COUNT(*) AS numProdotti
FROM prodotto
GROUP BY categoria
HAVING AVG(prezzo) > 50
ORDER BY prezzoMedio DESC;
```

Listing 10.9: HAVING

10.3.4 WHERE e HAVING insieme

```sql
-- Clienti di Milano che hanno speso più di 500 euro nel 2023
SELECT
    c.idCliente,
    c.nome,
    SUM(o.totale) AS totalSpeso
FROM ordine o
JOIN cliente c ON o.idCliente = c.idCliente
WHERE c.città = 'Milano'            -- Filtra prima di GROUP BY
    AND YEAR(o.dataOrdine) = 2023
GROUP BY c.idCliente, c.nome
HAVING SUM(o.totale) > 500          -- Filtra dopo GROUP BY
ORDER BY totalSpeso DESC;
```

Listing 10.10: WHERE e HAVING insieme

10.4 DISTINCT con Aggregazioni

10.4.1 COUNT(DISTINCT)

Conta valori unici di una colonna.

```sql
-- Numero di città diverse dove abitano i clienti
SELECT COUNT(DISTINCT città) AS numCittà FROM cliente;

-- Numero di categorie di prodotti ordinati
SELECT COUNT(DISTINCT categoria) AS numCategorie
FROM ordine_prodotto op
JOIN prodotto p ON op.idProdotto = p.idProdotto;
```

Listing 10.11: COUNT(DISTINCT)

10.4.2 SUM(DISTINCT) e AVG(DISTINCT)

```sql
-- Somma dei prezzi unici (ogni prezzo conta una sola volta)
SELECT SUM(DISTINCT prezzo) AS sommaPrezziUnici FROM prodotto;

-- Media dei prezzi unici
SELECT AVG(DISTINCT prezzo) AS mediaPreziUnici FROM prodotto;
```

Listing 10.12: SUM(DISTINCT) e AVG(DISTINCT)

10.5 Funzioni Avanzate di Aggregazione

10.5.1 GROUP$_CONCAT$

Concatena i valori di una colonna per ogni gruppo (MySQL).

```sql
-- Elencare i prodotti ordinati per ogni cliente
```

```sql
SELECT
    c.nome,
    GROUP_CONCAT(p.nome SEPARATOR ', ') AS prodottiOrdinati
FROM cliente c
LEFT JOIN ordine o ON c.idCliente = o.idCliente
LEFT JOIN ordine_prodotto op ON o.idOrdine = op.idOrdine
LEFT JOIN prodotto p ON op.idProdotto = p.idProdotto
GROUP BY c.idCliente, c.nome;
```

Listing 10.13: GROUP$_{C}ONCAT$

10.5.2 Aggregazioni condizionali

Usare CASE dentro le aggregazioni per logica condizionale.

```sql
-- Contare ordini completati e cancellati separatamente
SELECT
    COUNT(*) AS totalOrdini,
    SUM(CASE WHEN stato = 'Completato' THEN 1 ELSE 0 END) AS
        ordinCompletati,
    SUM(CASE WHEN stato = 'Cancellato' THEN 1 ELSE 0 END) AS
        ordiniCancellati
FROM ordine;

-- Vendite per stato
SELECT
    SUM(CASE WHEN stato = 'Completato' THEN totale ELSE 0 END) AS
        venditeCmpl,
    SUM(CASE WHEN stato = 'Spedito' THEN totale ELSE 0 END) AS
        venditeSpedite,
    SUM(CASE WHEN stato = 'Pendente' THEN totale ELSE 0 END) AS
        venditePercetti
FROM ordine;
```

Listing 10.14: Aggregazioni condizionali

10.6 NULL nelle Aggregazioni

Le funzioni di aggregazione ignorano i NULL.

```sql
-- Conteggio con NULL
SELECT
    COUNT(*) AS totalRighe,           -- Conta tutti (incluso NULL)
    COUNT(telefono) AS righeConTelefono  -- Conta solo non-NULL
FROM cliente;

-- Risultato: se 100 clienti, 10 senza telefono
-- totalRighe = 100, righeConTelefono = 90

-- SUM ignora NULL
SELECT SUM(sconto) FROM cliente;
-- I clienti senza sconto (NULL) non contribuiscono alla somma
```

Listing 10.15: NULL in aggregazioni

10.7 Esempio Completo: Report Vendite

```sql
-- Report vendite mensili per ogni categoria
SELECT
    DATE_FORMAT(o.dataOrdine, '%Y-%m') AS mese,
    p.categoria,
    COUNT(DISTINCT o.idOrdine) AS numOrdini,
    SUM(op.quantita) AS totalQuantita,
    SUM(op.quantita * op.prezzo_unitario) AS totalVendite,
    AVG(op.prezzo_unitario) AS prezzoMedio,
    MIN(op.prezzo_unitario) AS prezzoMin,
    MAX(op.prezzo_unitario) AS prezzoMax
FROM ordine o
JOIN ordine_prodotto op ON o.idOrdine = op.idOrdine
JOIN prodotto p ON op.idProdotto = p.idProdotto
WHERE o.stato IN ('Completato', 'Spedito')
GROUP BY DATE_FORMAT(o.dataOrdine, '%Y-%m'), p.categoria
HAVING SUM(op.quantita * op.prezzo_unitario) > 1000
ORDER BY mese DESC, totalVendite DESC;
```

Listing 10.16: Report completo con aggregazioni

Riepilogo concetti chiave

Concetti fondamentali

- **COUNT**: conta righe o valori non-NULL
- **SUM**: somma colonne numeriche
- **AVG, MIN, MAX**: media, minimo, massimo
- **GROUP BY**: raggruppa dati per una o più colonne
- **HAVING**: filtra i gruppi (applicato dopo GROUP BY)
- **WHERE**: filtra le righe (applicato prima di GROUP BY)
- **DISTINCT**: conta o somma solo valori unici
- Le aggregazioni ignorano NULL
- Combina con JOIN per aggregazioni multi-tabella

Esercizi

1. Scrivi una query per contare il numero totale di ordini e il numero di ordini completati.

2. Calcola il totale speso, la media e il massimo importo di ordini per ogni cliente. Ordina per totale speso decrescente.

3. Usa GROUP BY per trovare quali categorie di prodotti hanno vendite superiori a 5000 euro. Mostra categoria, numero prodotti, e totale vendite.

4. Scrivi una query con WHERE e HAVING: trova clienti di Milano che hanno fatto più di 3 ordini nel 2023.

5. Usa COUNT(DISTINCT) per trovare quanti clienti diversi hanno ordinato ogni prodotto.

6. Crea un report che mostra, per ogni mese del 2023, il numero di ordini, il totale vendite, la media ordine e il numero di clienti unici.

7. Usa aggregazioni condizionali (CASE) per mostrare, per ogni categoria, il numero di prodotti attivi e inattivi.

8. Spiega perché la seguente query dà errore e corriggila:

```
1 SELECT idCliente, nome, COUNT(*) FROM ordine GROUP BY idCliente;
```

Capitolo 11

SQL Subquery e Viste

Introduzione

Le subquery (query annidate) permettono di usare il risultato di una query come fonte per un'altra query. Le viste sono query salvate che si comportano come tabelle virtuali. Questi strumenti avanzati consentono di scrivere query complesse e riutilizzabili, organizzando la logica in modo strutturato.

Obiettivi di apprendimento

- Comprendere la differenza tra subquery scalare, riga e tabella
- Usare subquery nella clausola SELECT, WHERE e FROM
- Usare gli operatori EXISTS, IN, ANY, ALL con subquery
- Scrivere query correlate
- Creare e gestire viste con CREATE VIEW
- Modificare e eliminare viste
- Usare viste per semplificare query complesse
- Comprendere limitazioni e prestazioni delle viste

11.1 Subquery - Query Annidate

Una subquery (o inner query) è una query che si trova dentro un'altra query (outer query). La subquery fornisce dati che la query esterna utilizza.

11.1.1 Tipi di Subquery

Subquery Scalare

Una subquery scalare restituisce una singola riga e una singola colonna. Può essere usata dove è previsto un valore singolo (SELECT, WHERE, etc.).

```sql
-- Trovare il cliente con l'ordine più recente
SELECT nome, cognome
FROM cliente
WHERE idCliente = (
    -- Subquery scalare: restituisce un singolo idCliente
    SELECT idCliente FROM ordine
    ORDER BY dataOrdine DESC
    LIMIT 1
);

-- Aggiungere il prezzo medio a ogni prodotto
SELECT nome, prezzo,
    (SELECT AVG(prezzo) FROM prodotto) AS prezzoMedio
FROM prodotto;
```

Listing 11.1: Subquery Scalare

Nota: Subquery Scalara

Una subquery scalare deve restituire esattamente una riga e una colonna. Se restituisce 0 righe, il risultato è NULL. Se restituisce più di una riga, causa errore.

Subquery Riga

Una subquery riga restituisce una singola riga ma più colonne. Utile per confrontare più valori contemporaneamente.

```sql
-- Trovare ordini con lo stesso cliente e data del primo ordine
SELECT idOrdine, dataOrdine, totale
FROM ordine
WHERE (idCliente, dataOrdine) = (
    -- Subquery riga: restituisce una riga con due colonne
    SELECT idCliente, dataOrdine FROM ordine
    LIMIT 1
);

-- Trovare clienti con stessa città e provincia di Milano
SELECT nome, città, provincia
FROM cliente
WHERE (città, provincia) = (
    SELECT città, provincia FROM cliente
    WHERE città = 'Milano' LIMIT 1
);
```

Listing 11.2: Subquery Riga

Subquery Tabella

Una subquery tabella restituisce più righe e/o più colonne. Spesso usata nella clausola FROM per creare una tabella temporanea (derived table).

```sql
-- Trovare prodotti con prezzo sopra la media
SELECT nome, prezzo
FROM prodotto
WHERE prezzo > (
    -- Subquery tabella nella clausola WHERE
    SELECT AVG(prezzo) FROM prodotto
);

-- Usare subquery nella clausola FROM (derived table)
SELECT categoria, numProdotti, prezzoMedio
FROM (
    SELECT categoria, COUNT(*) AS numProdotti, AVG(prezzo) AS
       prezzoMedio
    FROM prodotto
    GROUP BY categoria
) AS statProdotto
WHERE numProdotti > 5
ORDER BY prezzoMedio DESC;
```

Listing 11.3: Subquery Tabella

Tip: Alias per Derived Table

Quando usi una subquery nella clausola FROM, devi dare un alias con AS. Questo alias può essere usato nel resto della query per fare riferimento alle colonne della subquery.

11.2 Operatori di Subquery

11.2.1 Operatore IN

IN controlla se un valore è contenuto in un insieme di valori restituito dalla subquery.

```sql
-- Trovare clienti che hanno fatto almeno un ordine
SELECT nome, cognome, email
FROM cliente
WHERE idCliente IN (
    -- Subquery: restituisce set di idCliente
    SELECT DISTINCT idCliente FROM ordine
);

-- Trovare ordini di clienti di Roma o Milano
SELECT idOrdine, dataOrdine, totale
FROM ordine
WHERE idCliente IN (
    SELECT idCliente FROM cliente
    WHERE città IN ('Roma', 'Milano')
);

-- NOT IN: clienti senza ordini
SELECT nome, cognome
FROM cliente
WHERE idCliente NOT IN (
    SELECT DISTINCT idCliente FROM ordine
);
```

Listing 11.4: Operatore IN

Attenzione: IN e NULL

Se la subquery contiene NULL e usi NOT IN, il risultato sarà sempre vuoto (perché NULL non è comparabile). Usa NOT IN solo se sei sicuro che la subquery non contiene NULL.

11.2.2 Operatore EXISTS

EXISTS verifica se la subquery restituisce almeno una riga. È più efficiente di IN per grandi dataset.

```sql
-- Trovare clienti con almeno un ordine (usando EXISTS)
SELECT nome, cognome
FROM cliente c
WHERE EXISTS (
    -- Subquery correlata: fa riferimento a c
    SELECT 1 FROM ordine o
    WHERE o.idCliente = c.idCliente
);

-- Trovare categorie con almeno 3 prodotti
SELECT DISTINCT categoria
FROM prodotto p1
WHERE EXISTS (
    SELECT COUNT(*) FROM prodotto p2
    WHERE p2.categoria = p1.categoria
    GROUP BY p2.categoria
    HAVING COUNT(*) >= 3
);

-- NOT EXISTS: clienti senza ordini
SELECT nome, cognome
FROM cliente c
WHERE NOT EXISTS (
    SELECT 1 FROM ordine o
    WHERE o.idCliente = c.idCliente
);
```

Listing 11.5: Operatore EXISTS

Subquery Correlata

Una subquery correlata fa riferimento a colonne della query esterna. Viene eseguita una volta per ogni riga della tabella esterna. È meno efficiente della versione non correlata, ma talvolta necessaria.

11.2.3 Operatori ANY e ALL

ANY (o SOME) verifica se il confronto è vero per almeno un valore. ALL verifica se è vero per tutti i valori.

```sql
-- Trovare ordini con totale maggiore di QUALSIASI ordine di Roma
SELECT idOrdine, totale
FROM ordine o
WHERE totale > ANY (
    SELECT totale FROM ordine
    WHERE idCliente IN (SELECT idCliente FROM cliente WHERE città = '
        Roma')
);

-- Trovare ordini con totale MINORE della media per categoria
SELECT idOrdine, totale
FROM ordine o
WHERE totale < ALL (
    -- ALL: minore di TUTTI i totali medi per categoria
    SELECT AVG(totale) FROM ordine
    GROUP BY idCliente
);

-- ANY equivalente a IN
SELECT * FROM ordine WHERE idCliente = ANY (
    SELECT idCliente FROM cliente WHERE città = 'Milano'
);
-- Equivalente a:
SELECT * FROM ordine WHERE idCliente IN (
    SELECT idCliente FROM cliente WHERE città = 'Milano'
);
```

Listing 11.6: Operatori ANY e ALL

11.3 Viste (Views)

Una vista è una query salvata che può essere usata come una tabella virtuale. Le viste consentono di:

- Semplificare query complesse
- Riutilizzare logica comune
- Fornire livelli di astrazione (nascondere dettagli di implementazione)
- Gestire la sicurezza (utenti accedono solo a colonne autorizzate)

11.3.1 CREATE VIEW

Il comando CREATE VIEW crea una nuova vista salvata nel database.

```sql
CREATE VIEW nome_vista AS
SELECT colonne
FROM tabelle
WHERE condizioni;
```

Listing 11.7: Sintassi CREATE VIEW

```sql
-- Vista semplice: clienti attivi
CREATE VIEW clientiAttivi AS
SELECT idCliente, nome, cognome, email, città
FROM cliente
WHERE stato = 'Attivo';

-- Vista con aggregazione: totale ordini per cliente
CREATE VIEW totaleOrdiniPerCliente AS
SELECT
    c.idCliente,
    c.nome,
    c.cognome,
    COUNT(o.idOrdine) AS numOrdini,
    SUM(o.totale) AS totalSpeso,
    AVG(o.totale) AS ordineMediano
FROM cliente c
LEFT JOIN ordine o ON c.idCliente = o.idCliente
GROUP BY c.idCliente, c.nome, c.cognome;

-- Vista con JOIN: informazioni ordine complete
CREATE VIEW ordineCompleto AS
SELECT
    o.idOrdine,
    o.dataOrdine,
    o.totale,
    c.nome AS nomeCliente,
    c.cognome AS cognomeCliente,
    c.email,
    p.nome AS nomeProdotto,
    d.quantità,
    d.prezzoUnitario,
    d.subtotale
FROM ordine o
INNER JOIN cliente c ON o.idCliente = c.idCliente
INNER JOIN dettaglioOrdine d ON o.idOrdine = d.idOrdine
INNER JOIN prodotto p ON d.idProdotto = p.idProdotto;
```

Listing 11.8: Esempi CREATE VIEW

11.3.2 Usare le Viste

Le viste si usano come tabelle normali nella clausola FROM.

```sql
-- Usare una vista come una tabella
SELECT nome, cognome, email
FROM clientiAttivi
WHERE città = 'Milano';

-- Aggregare dati da una vista
SELECT SUM(totalSpeso) AS totalVendite
FROM totaleOrdiniPerCliente
WHERE numOrdini > 5;

-- JOIN con una vista
SELECT oc.nome, oc.numOrdini, oc.totalSpeso
```

```
FROM ordineCompleto oc
WHERE oc.totale > 100
ORDER BY oc.totale DESC;
```

Listing 11.9: Usare le Viste

11.3.3 Modificare le Viste

ALTER VIEW

ALTER VIEW modifica la definizione di una vista esistente.

```
-- Modificare la definizione di una vista
ALTER VIEW clientiAttivi AS
SELECT idCliente, nome, cognome, email, città, dataRegistrazione
FROM cliente
WHERE stato = 'Attivo'
AND dataRegistrazione > DATE_SUB(NOW(), INTERVAL 1 YEAR);
```

Listing 11.10: ALTER VIEW

DROP VIEW

DROP VIEW elimina una vista dal database.

```
-- Eliminare una vista
DROP VIEW clientiAttivi;

-- Eliminare solo se esiste (evita errori)
DROP VIEW IF EXISTS clientiAttivi;

-- Eliminare multiple viste
DROP VIEW IF EXISTS clientiAttivi, totaleOrdiniPerCliente,
    ordineCompleto;
```

Listing 11.11: DROP VIEW

11.3.4 SHOW VIEWS

Per visualizzare tutte le viste nel database.

```
-- Elencare tutte le viste nel database attuale
SHOW FULL TABLES WHERE table_type = 'VIEW';

-- Visualizzare la definizione di una vista
SHOW CREATE VIEW clientiAttivi;

-- Interrogare il catalogo di sistema
SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_SCHEMA = 'database_name'
```

```
10 AND TABLE_TYPE = 'VIEW';
```

Listing 11.12: Visualizzare le Viste

11.4 Viste Aggiornabili (Updatable Views)

Una vista è aggiornabile se contiene UPDATE o DELETE, purché soddisfi certe condizioni:

- Nessun UNION, GROUP BY, HAVING, LIMIT
- Nessuna subquery
- Nessuna aggregazione
- Può contenere solo una tabella (con JOIN non è aggiornabile)

```
1  -- Vista aggiornabile: tutti i campi di cliente
2  CREATE VIEW clientiViewAggiornabile AS
3  SELECT idCliente, nome, cognome, email, città
4  FROM cliente;
5
6  -- Aggiornare tramite vista
7  UPDATE clientiViewAggiornabile
8  SET città = 'Roma'
9  WHERE idCliente = 1;
10
11 -- Equivalente all'aggiornamento della tabella sottostante
12 UPDATE cliente SET città = 'Roma' WHERE idCliente = 1;
13
14 -- Inserire tramite vista
15 INSERT INTO clientiViewAggiornabile (nome, cognome, email, città)
16 VALUES ('Rossi', 'Mario', 'mario@example.com', 'Milano');
17
18 -- Vista NON aggiornabile (con GROUP BY)
19 CREATE VIEW totaleOrdiniPerCliente AS
20 SELECT
21     idCliente,
22     COUNT(*) AS numOrdini,
23     SUM(totale) AS totalSpeso
24 FROM ordine
25 GROUP BY idCliente;
26
27 -- ERRORE: non è possibile aggiornare perché contiene GROUP BY
28 -- UPDATE totaleOrdiniPerCliente SET numOrdini = 5 WHERE idCliente =
       1;
```

Listing 11.13: Viste Aggiornabili

11.5 Viste Materialized (Snapshot)

Una vista materializzata è una copia dei dati della query salvata fisicamente nel database. È più veloce per query complesse, ma non è sempre aggiornata.

```sql
-- Creare una tabella per immagazzinare i dati della vista
CREATE TABLE totaleOrdiniPerClienteMaterializzato AS
SELECT
    idCliente,
    nome,
    cognome,
    COUNT(*) AS numOrdini,
    SUM(totale) AS totalSpeso
FROM cliente c
LEFT JOIN ordine o ON c.idCliente = o.idCliente
GROUP BY idCliente, nome, cognome;

-- Aggiornare periodicamente i dati (es. ogni notte)
-- Cancellare i dati vecchi
TRUNCATE TABLE totaleOrdiniPerClienteMaterializzato;

-- Ricaricare i dati aggiornati
INSERT INTO totaleOrdiniPerClienteMaterializzato
SELECT
    idCliente,
    nome,
    cognome,
    COUNT(*) AS numOrdini,
    SUM(totale) AS totalSpeso
FROM cliente c
LEFT JOIN ordine o ON c.idCliente = o.idCliente
GROUP BY idCliente, nome, cognome;
```

Listing 11.14: Simulare Viste Materializzate

Vantaggi e Svantaggi Viste

Vantaggi:

- Semplificano query complesse
- Migliorano leggibilità
- Facilitano manutenzione
- Nascondono complessità

Svantaggi:

- Prestazioni peggiori per query complicate (non materializzate)
- Difficili da debuggare
- Dipendenze sulle tabelle sottostanti

11.6 Riepilogo Concetti Chiave

Subquery Query annidata che fornisce dati a un'altra query. Può essere scalare (1 valore), riga (1 riga) o tabella (più righe/colonne).

IN Operatore che verifica se un valore è in un insieme. Efficiente per piccoli set.

EXISTS Operatore che verifica se una subquery restituisce righe. Efficiente per grandi dataset.

ANY/ALL Operatori che confrontano un valore con qualsiasi/tutti i valori di un insieme.

Vista Query salvata che si comporta come una tabella virtuale.

Subquery Correlata Subquery che fa riferimento a colonne della query esterna.

Derived Table Subquery nella clausola FROM con alias.

Capitolo 12

Transazioni e Controllo della Concorrenza

Introduzione

Una transazione è un'unità di lavoro atomica che raggruppa una o più operazioni SQL. Le transazioni garantiscono la consistenza del database anche in caso di errori o guasti. Questo capitolo presenta i concetti ACID, comandi di controllo delle transazioni (START, COMMIT, ROLLBACK, SAVEPOINT) e i livelli di isolamento per gestire la concorrenza.

Obiettivi di apprendimento

- Comprendere il concetto di transazione e la sua importanza
- Comprendere le proprietà ACID
- Usare i comandi START TRANSACTION, COMMIT, ROLLBACK
- Usare SAVEPOINT per rollback parziali
- Comprendere i livelli di isolamento (READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE)
- Gestire anomalie di concorrenza (dirty read, non-repeatable read, phantom read)
- Identificare e risolvere deadlock
- Configurare e monitorare transazioni

12.1 Proprietà ACID

Le proprietà ACID garantiscono che le transazioni siano affidabili e mantengono la integrità del database.

Atomicità (Atomicity) Una transazione è "tutto o nulla". O tutte le operazioni vengono completate (COMMIT), o nessuna (ROLLBACK). Non può esistere uno stato intermedio.

Coerenza (Consistency) Le transazioni portano il database da uno stato coerente a un altro. Non è possibile violare vincoli di integrità (PK, FK, CHECK).

Isolamento (Isolation) Le transazioni concorrenti non si interferiscono. Ogni transazione vede il database come se fosse l'unica in esecuzione.

Durabilità (Durability) Una volta che una transazione è COMMIT, i dati sono permanentemente salvati, anche in caso di crash o blackout.

Esempio ACID

Un trasferimento di denaro tra conti:

1. **Atomicità**: Debito dal conto A E credito nel conto B avvengono insieme. Se uno fallisce, entrambi falliscono.
2. **Coerenza**: Il totale di denaro nei due conti rimane costante prima e dopo il trasferimento.
3. **Isolamento**: Altre transazioni non vedono lo stato intermedio (A diminuito, B non ancora aumentato).
4. **Durabilità**: Dopo COMMIT, il trasferimento è permanente anche se il server crasha un secondo dopo.

12.2 Comandi di Controllo Transazioni

12.2.1 START TRANSACTION

Inizia una nuova transazione. Tutte le operazioni successive fanno parte della transazione finché non viene COMMIT o ROLLBACK.

```
START TRANSACTION;
-- Oppure in alcuni DBMS:
BEGIN TRANSACTION;
BEGIN;
```

Listing 12.1: Sintassi START TRANSACTION

12.2.2 COMMIT

COMMIT salva permanentemente tutte le modifiche della transazione nel database.

```
START TRANSACTION;

-- Operazione 1: Aggiornare saldo cliente
UPDATE cliente SET saldo = saldo - 100 WHERE idCliente = 1;

-- Operazione 2: Registrare la transazione
INSERT INTO movimento (idCliente, tipo, importo, data)
VALUES (1, 'Prelievo', 100, NOW());

```

```
-- Se tutto è corretto, salvare permanentemente
COMMIT;

-- Dopo COMMIT, i dati sono permanenti
SELECT saldo FROM cliente WHERE idCliente = 1;  -- saldo ridotto di
    100
```

Listing 12.2: Esempio COMMIT

12.2.3 ROLLBACK

ROLLBACK annulla tutte le modifiche della transazione. Il database torna allo stato prima di START TRANSACTION.

```
START TRANSACTION;

-- Operazione 1
UPDATE cliente SET saldo = saldo - 100 WHERE idCliente = 1;

-- Operazione 2
INSERT INTO movimento (idCliente, tipo, importo, data)
VALUES (1, 'Prelievo', 100, NOW());

-- Se si verifica un errore o vuoi annullare
ROLLBACK;

-- Dopo ROLLBACK, i dati tornano allo stato precedente
SELECT saldo FROM cliente WHERE idCliente = 1;  -- saldo invariato
```

Listing 12.3: Esempio ROLLBACK

Attenzione: ROLLBACK è Definitivo

Una volta che eseguito ROLLBACK, le modifiche sono perse per sempre. Non è possibile "undo del ROLLBACK".

12.2.4 SAVEPOINT

SAVEPOINT crea un punto di salvataggio all'interno di una transazione. Permette di fare ROLLBACK parziale fino a quel punto, mantenendo le operazioni precedenti.

```
SAVEPOINT nome_savepoint;

-- Fare rollback fino a un savepoint
ROLLBACK TO SAVEPOINT nome_savepoint;

-- Rilasciare un savepoint (liberare memoria)
RELEASE SAVEPOINT nome_savepoint;
```

Listing 12.4: Sintassi SAVEPOINT

```sql
START TRANSACTION;

-- Operazione 1: Debito dal cliente
UPDATE cliente SET saldo = saldo - 100 WHERE idCliente = 1;
INSERT INTO movimento VALUES (1, 'Prelievo', 100, NOW());

-- Creare un savepoint
SAVEPOINT dopo_prelievo;

-- Operazione 2: Credito al cliente 2
UPDATE cliente SET saldo = saldo + 100 WHERE idCliente = 2;

-- Se si verifica un errore con il cliente 2
-- Fare rollback solo all'ultima operazione
ROLLBACK TO SAVEPOINT dopo_prelievo;

-- Il prelievo dal cliente 1 rimane
-- Ma il credito al cliente 2 è annullato

-- Riprovare l'operazione 2 con cliente diverso
UPDATE cliente SET saldo = saldo + 100 WHERE idCliente = 3;
INSERT INTO movimento VALUES (3, 'Credito', 100, NOW());

-- Se tutto è corretto, salvare permanentemente
COMMIT;
```

Listing 12.5: Esempio SAVEPOINT

12.3 Livelli di Isolamento

Il livello di isolamento determina come le transazioni concorrenti interagiscono. Un isolamento più alto garantisce meno anomalie, ma riduce le prestazioni.

12.3.1 Anomalie di Concorrenza

Prima di descrivere i livelli, ecco le anomalie che possono verificarsi:

Dirty Read Una transazione legge dati non ancora committati scritti da un'altra transazione. Se quella transazione fa ROLLBACK, i dati letti erano "sporchi".

Non-Repeatable Read Una transazione legge la stessa riga due volte e ottiene risultati diversi perché un'altra transazione ha modificato i dati nel mezzo.

Phantom Read Una transazione legge un insieme di righe due volte e ottiene risultati diversi perché un'altra transazione ha inserito/eliminato righe nel mezzo.

12.3.2 Livelli di Isolamento in SQL

READ UNCOMMITTED

Livello di isolamento più basso. Una transazione può leggere dati non ancora committati (dirty read).

```sql
-- Impostare il livello di isolamento
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
START TRANSACTION;

-- Lettura "sporca": legge dati non committati
SELECT saldo FROM cliente WHERE idCliente = 1;

COMMIT;
```

Listing 12.6: READ UNCOMMITTED

Attenzione: READ UNCOMMITTED è Rischioso

Questo livello è molto pericoloso per dati critici (finanze, inventario). Usa solo in scenari dove precisione non è critica (report, cache).

READ COMMITTED

Livello intermedio. Una transazione legge solo dati committati, evitando dirty read. Ma permite non-repeatable read e phantom read.

```sql
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;

-- Legge solo dati committati
SELECT saldo FROM cliente WHERE idCliente = 1;

-- Altra transazione potrebbe modificare il saldo nel frattempo
SELECT saldo FROM cliente WHERE idCliente = 1;  -- Potrebbe essere
    diverso

COMMIT;
```

Listing 12.7: READ COMMITTED

REPEATABLE READ

Una transazione legge gli stessi dati sempre allo stesso modo durante la transazione. Evita dirty read e non-repeatable read, ma permite phantom read (inserimenti/eliminazioni di nuove righe).

```sql
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION;

-- Prima lettura
SELECT COUNT(*) FROM ordine WHERE idCliente = 1;  -- Es: 5

-- Altra transazione inserisce un nuovo ordine
SELECT COUNT(*) FROM ordine WHERE idCliente = 1;  -- Ancora 5 (phantom
    read possibile se cambiano colonne)

COMMIT;
```

Listing 12.8: REPEATABLE READ

SERIALIZABLE

Livello di isolamento più alto. Le transazioni si comportano come se fossero eseguite in sequenza (serialmente), non concorrentemente.

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
START TRANSACTION;

-- Nessuna anomalia: dirty read, non-repeatable read, phantom read
SELECT saldo FROM cliente WHERE idCliente = 1;

-- ... altre operazioni ...

COMMIT;
```

Listing 12.9: SERIALIZABLE

Trade-off: Isolamento vs Prestazioni

- READ UNCOMMITTED: Massime prestazioni, rischio massimo di anomalie
- READ COMMITTED: Buon compromesso, livello di default in molti DBMS
- REPEATABLE READ: Isolamento quasi completo, piccolo calo prestazioni
- SERIALIZABLE: Massimo isolamento, prestazioni minime (possibili deadlock)

12.3.3 Impostare il Livello di Isolamento

```
-- Impostare per la sessione attuale
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

-- Impostare per una transazione specifica (MySQL 5.7+)
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
START TRANSACTION;
-- ... operazioni ...
COMMIT;

-- Visualizzare il livello attuale
SELECT @@transaction_isolation;

-- Impostare globalmente (richiede SUPER privilege)
SET GLOBAL transaction_isolation='READ-COMMITTED';
```

Listing 12.10: Impostare Livello di Isolamento

12.4 Deadlock

Un deadlock si verifica quando due o più transazioni si bloccano reciprocamente, aspettando risorse che l'altra transazione detiene. Nessuna può procedere.

12.4.1 Causa Comune di Deadlock

```
-- Transazione 1
START TRANSACTION;
UPDATE cliente SET saldo = saldo - 100 WHERE idCliente = 1;
-- Aspetta il lock su cliente 2
UPDATE cliente SET saldo = saldo + 100 WHERE idCliente = 2;
COMMIT;

-- Transazione 2 (parallelamente)
START TRANSACTION;
UPDATE cliente SET saldo = saldo - 100 WHERE idCliente = 2;
-- Aspetta il lock su cliente 1 (DEADLOCK!)
UPDATE cliente SET saldo = saldo + 100 WHERE idCliente = 1;
COMMIT;
```

Listing 12.11: Scenario Deadlock

12.4.2 Evitare Deadlock

```
-- Strategie 1: Sempre aggiornare le righe nello stesso ordine
-- Transazione 1 e 2: aggiornare sempre cliente 1 prima di cliente 2
START TRANSACTION;
UPDATE cliente SET saldo = saldo - 100 WHERE idCliente = 1;  -- Sempre
    prima
UPDATE cliente SET saldo = saldo + 100 WHERE idCliente = 2;  -- Sempre
    dopo
COMMIT;

-- Strategia 2: Usare timeout per il lock
SET innodb_lock_wait_timeout = 5;  -- Timeout dopo 5 secondi

-- Strategia 3: Usare transaction_isolation più bassa (se accettabile)
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

-- Strategia 4: Minimizzare la durata della transazione
START TRANSACTION;
-- Fare il meno possibile
UPDATE cliente SET saldo = saldo - 100 WHERE idCliente = 1;
COMMIT;  -- Rilasciare i lock il prima possibile
```

Listing 12.12: Strategie per Evitare Deadlock

12.4.3 Gestire Deadlock in Applicazione

```
-- Pseudocodice per gestire deadlock
MAX_RETRIES = 3
retry_count = 0

WHILE retry_count < MAX_RETRIES:
    TRY:
        START TRANSACTION
        -- Operazioni
        UPDATE cliente SET saldo = ... WHERE idCliente = 1
        UPDATE cliente SET saldo = ... WHERE idCliente = 2
        COMMIT
        BREAK  -- Successo, esci dal loop
    EXCEPT Deadlock:
        retry_count += 1
        ROLLBACK
        WAIT(random(1, 5) seconds)  -- Attendi con backoff casuale
    EXCEPT OTHER_ERROR:
        ROLLBACK
        RAISE error

IF retry_count >= MAX_RETRIES:
    RAISE "Deadlock non risolto dopo 3 tentativi"
```

Listing 12.13: Gestione Deadlock in Pseudocodice

12.5 Monitoraggio Transazioni

12.5.1 Visualizzare Transazioni Attive

```
-- Visualizzare tutti i processi in esecuzione
SHOW PROCESSLIST;

-- Dettagli completi
SHOW FULL PROCESSLIST;

-- Mostrare solo transazioni lunghe
SELECT * FROM INFORMATION_SCHEMA.PROCESSLIST
WHERE TIME > 300  -- Più di 5 minuti
AND COMMAND != 'Sleep';

-- Visualizzare transazioni InnoDB
SELECT * FROM INFORMATION_SCHEMA.INNODB_TRX;

-- Visualizzare lock InnoDB
SELECT * FROM INFORMATION_SCHEMA.INNODB_LOCKS;
```

Listing 12.14: Monitorare Transazioni Active

12.5.2 Terminare Transazioni

```
-- Terminare un processo specifico
```

```sql
KILL 123;  -- Dove 123 è l'ID del processo

-- Terminare tutti i processi di un utente
KILL QUERY 456;  -- Termina solo la query, non la connessione

-- Script per killare processi lunghi
SELECT CONCAT('KILL ', ID, ';')
FROM INFORMATION_SCHEMA.PROCESSLIST
WHERE TIME > 300 AND COMMAND != 'Sleep';
```

Listing 12.15: Terminare Transazioni o Processi

12.6 Transazioni Implicite vs Esplicite

12.6.1 Modalità Autocommit

La modalità autocommit determina se i comandi SQL sono automaticamente committati.

```sql
-- Visualizzare stato autocommit
SELECT @@autocommit;  -- 1 = attivo, 0 = disattivo

-- Disattivare autocommit (transazioni esplicite)
SET autocommit = 0;

UPDATE cliente SET saldo = saldo - 100 WHERE idCliente = 1;
-- Modifica non è ancora salvata finché non fai COMMIT
COMMIT;

-- Attivare autocommit (transazioni implicite)
SET autocommit = 1;

UPDATE cliente SET saldo = saldo - 100 WHERE idCliente = 1;
-- Modifica è automaticamente salvata immediatamente
```

Listing 12.16: Autocommit

12.7 Riepilogo Concetti Chiave

Transazione Unità di lavoro atomica che raggruppa una o più operazioni SQL.

ACID Proprietà che garantiscono affidabilità: Atomicità, Coerenza, Isolamento, Durabilità.

COMMIT Comando che salva permanentemente le modifiche della transazione.

ROLLBACK Comando che annulla tutte le modifiche della transazione.

SAVEPOINT Punto di salvataggio per rollback parziale all'interno di una transazione.

Livello di Isolamento Grado in cui le transazioni sono isolate l'una dall'altra.

Deadlock Situazione in cui due transazioni aspettano reciprocamente, bloccandosi a vicenda.

Dirty Read Anomalia: leggere dati non committati.

Non-Repeatable Read Anomalia: leggere dati diversi nella stessa transazione.

Phantom Read Anomalia: ottenere risultati diversi per lo stesso query a causa di inserimenti/eliminazioni.

Capitolo 13

Esercizi Completi

Introduzione

Questo capitolo contiene 20 esercizi progressivi che coprono tutti gli argomenti del corso, dalla progettazione ER alla scrittura di query SQL complesse. Gli esercizi sono divisi per livello di difficoltà: base (esercizi 1-6), intermedio (7-13) e avanzato (14-20). Ogni esercizio include la descrizione, il diagramma ER (dove appropriato), schema SQL e query di esempio.

Esercizi Base (1-6)

13.1 Esercizio 1: Biblioteca Semplice

13.1.1 Descrizione

Progettare un database per una biblioteca che gestisce libri, autori e prestiti. Ogni libro ha un titolo, codice ISBN, autore e numero di copie disponibili. Gli autori hanno nome e email. I prestiti registrano quando un cliente prende e restituisce un libro.

13.1.2 Diagramma ER

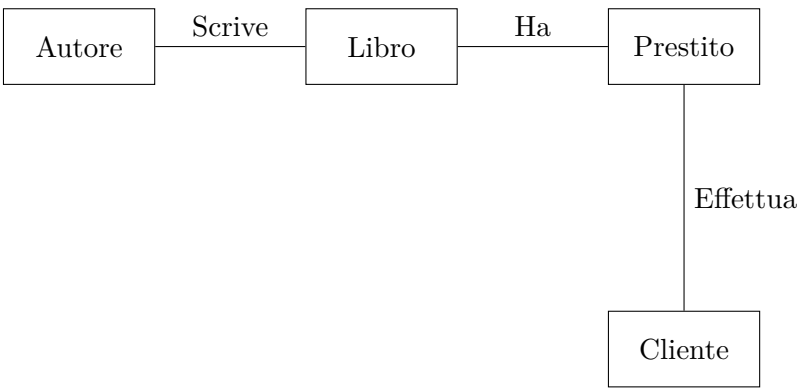


Figura 13.1: Diagramma ER Biblioteca

13.1.3 Schema SQL

```sql
CREATE TABLE autore (
    idAutore INT PRIMARY KEY AUTO_INCREMENT,
    nome VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL
);

CREATE TABLE libro (
    idLibro INT PRIMARY KEY AUTO_INCREMENT,
    isbn VARCHAR(13) UNIQUE NOT NULL,
    titolo VARCHAR(200) NOT NULL,
    idAutore INT NOT NULL,
    copieDisponibili INT DEFAULT 1,
    FOREIGN KEY (idAutore) REFERENCES autore(idAutore)
);

CREATE TABLE cliente (
    idCliente INT PRIMARY KEY AUTO_INCREMENT,
    nome VARCHAR(100) NOT NULL,
    cognome VARCHAR(100) NOT NULL,
    telefono VARCHAR(20)
);

CREATE TABLE prestito (
    idPrestito INT PRIMARY KEY AUTO_INCREMENT,
    idLibro INT NOT NULL,
    idCliente INT NOT NULL,
    dataPrestito DATE NOT NULL,
    dataRestituzione DATE,
    FOREIGN KEY (idLibro) REFERENCES libro(idLibro),
    FOREIGN KEY (idCliente) REFERENCES cliente(idCliente)
);
```

Listing 13.1: Creazione Tabelle Biblioteca

13.1.4 Query di Esempio

```sql
-- 1. Trovare tutti i libri di un autore specifico
SELECT titolo, isbn FROM libro WHERE idAutore = 1;

-- 2. Contare i prestiti attivi (non restituiti)
SELECT COUNT(*) AS prestitiAttivi FROM prestito WHERE dataRestituzione
    IS NULL;

-- 3. Trovare i libri più prestati
SELECT l.titolo, COUNT(p.idPrestito) AS numPrestiti
FROM libro l
LEFT JOIN prestito p ON l.idLibro = p.idLibro
GROUP BY l.idLibro, l.titolo
ORDER BY numPrestiti DESC;
```

Listing 13.2: Query Biblioteca

13.2 Esercizio 2: Azienda e Dipendenti

13.2.1 Descrizione

Modellare un'azienda con più dipartimenti. Ogni dipartimento ha un nome e un responsabile (che è un dipendente). Ogni dipendente ha nome, cognome, stipendio e appartiene a un dipartimento.

13.2.2 Schema SQL

```sql
CREATE TABLE dipartimento (
    idDipartimento INT PRIMARY KEY AUTO_INCREMENT,
    nome VARCHAR(100) NOT NULL UNIQUE,
    idResponsabile INT
);

CREATE TABLE dipendente (
    idDipendente INT PRIMARY KEY AUTO_INCREMENT,
    nome VARCHAR(100) NOT NULL,
    cognome VARCHAR(100) NOT NULL,
    stipendio DECIMAL(10, 2),
    idDipartimento INT NOT NULL,
    FOREIGN KEY (idDipartimento) REFERENCES dipartimento(
        idDipartimento)
);

-- Aggiungere la FK al responsabile (gestire circolarità)
ALTER TABLE dipartimento
ADD CONSTRAINT fk_responsabile FOREIGN KEY (idResponsabile) REFERENCES
    dipendente(idDipendente);
```

Listing 13.3: Creazione Tabelle Azienda

13.2.3 Query di Esempio

```sql
-- Stipendio medio per dipartimento
SELECT d.nome, AVG(dip.stipendio) AS stipendioMedio
FROM dipartimento d
LEFT JOIN dipendente dip ON d.idDipartimento = dip.idDipartimento
GROUP BY d.idDipartimento, d.nome;

-- Dipendenti che guadagnano più della media del loro dipartimento
SELECT dip.nome, dip.cognome, dip.stipendio
FROM dipendente dip
WHERE dip.stipendio > (
    SELECT AVG(stipendio) FROM dipendente dip2
    WHERE dip2.idDipartimento = dip.idDipartimento
);
```

Listing 13.4: Query Azienda

13.3 Esercizio 3: E-commerce Semplice

13.3.1 Descrizione

Modellare un semplice e-commerce: clienti che effettuano ordini, ordini contengono prodotti, ogni prodotto ha un prezzo e categoria.

13.3.2 Schema SQL

```sql
CREATE TABLE cliente (
    idCliente INT PRIMARY KEY AUTO_INCREMENT,
    nome VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    dataRegistrazione DATE DEFAULT CURDATE()
);

CREATE TABLE categoria (
    idCategoria INT PRIMARY KEY AUTO_INCREMENT,
    nome VARCHAR(50) UNIQUE NOT NULL
);

CREATE TABLE prodotto (
    idProdotto INT PRIMARY KEY AUTO_INCREMENT,
    nome VARCHAR(100) NOT NULL,
    prezzo DECIMAL(10, 2) NOT NULL,
    idCategoria INT NOT NULL,
    FOREIGN KEY (idCategoria) REFERENCES categoria(idCategoria)
);

CREATE TABLE ordine (
    idOrdine INT PRIMARY KEY AUTO_INCREMENT,
    idCliente INT NOT NULL,
    dataOrdine DATE DEFAULT CURDATE(),
    stato ENUM('Pendente', 'Spedito', 'Consegnato', 'Cancellato'),
    FOREIGN KEY (idCliente) REFERENCES cliente(idCliente)
);

CREATE TABLE dettaglioOrdine (
    idDettaglio INT PRIMARY KEY AUTO_INCREMENT,
    idOrdine INT NOT NULL,
    idProdotto INT NOT NULL,
    quantità INT NOT NULL,
    prezzoUnitario DECIMAL(10, 2) NOT NULL,
    FOREIGN KEY (idOrdine) REFERENCES ordine(idOrdine),
    FOREIGN KEY (idProdotto) REFERENCES prodotto(idProdotto)
);
```

Listing 13.5: Creazione Tabelle E-commerce

13.4 Esercizio 4: Ospedale

13.4.1 Descrizione

Modellare un ospedale: pazienti, medici, reparti, ricoveri e visite. Ogni paziente può essere ricoverato in un reparto per un periodo. Ogni visita è effettuata da un medico a un paziente in una data specifica.

13.4.2 Schema SQL

```sql
CREATE TABLE reparto (
    idReparto INT PRIMARY KEY AUTO_INCREMENT,
    nome VARCHAR(100) UNIQUE NOT NULL,
    numLetti INT
);

CREATE TABLE medico (
    idMedico INT PRIMARY KEY AUTO_INCREMENT,
    nome VARCHAR(100) NOT NULL,
    cognome VARCHAR(100) NOT NULL,
    specializzazione VARCHAR(50),
    idReparto INT,
    FOREIGN KEY (idReparto) REFERENCES reparto(idReparto)
);

CREATE TABLE paziente (
    idPaziente INT PRIMARY KEY AUTO_INCREMENT,
    nome VARCHAR(100) NOT NULL,
    cognome VARCHAR(100) NOT NULL,
    dataRegistrazione DATE DEFAULT CURDATE()
);

CREATE TABLE ricovero (
    idRicovero INT PRIMARY KEY AUTO_INCREMENT,
    idPaziente INT NOT NULL,
    idReparto INT NOT NULL,
    dataInizio DATE NOT NULL,
    dataFine DATE,
    FOREIGN KEY (idPaziente) REFERENCES paziente(idPaziente),
    FOREIGN KEY (idReparto) REFERENCES reparto(idReparto)
);

CREATE TABLE visita (
    idVisita INT PRIMARY KEY AUTO_INCREMENT,
    idPaziente INT NOT NULL,
    idMedico INT NOT NULL,
    dataVisita DATE NOT NULL,
    diagnosi TEXT,
    FOREIGN KEY (idPaziente) REFERENCES paziente(idPaziente),
    FOREIGN KEY (idMedico) REFERENCES medico(idMedico)
);
```

Listing 13.6: Creazione Tabelle Ospedale

13.5 Esercizio 5: Scuola

13.5.1 Descrizione

Modellare una scuola: studenti, insegnanti, classi, corsi e voti. Ogni studente appartiene a una classe e riceve voti per i corsi.

13.5.2 Schema SQL

```sql
CREATE TABLE classe (
    idClasse INT PRIMARY KEY AUTO_INCREMENT,
    nome VARCHAR(10) UNIQUE NOT NULL,
    numStudenti INT
);

CREATE TABLE studente (
    idStudente INT PRIMARY KEY AUTO_INCREMENT,
    nome VARCHAR(100) NOT NULL,
    cognome VARCHAR(100) NOT NULL,
    dataIscrizione DATE DEFAULT CURDATE(),
    idClasse INT NOT NULL,
    FOREIGN KEY (idClasse) REFERENCES classe(idClasse)
);

CREATE TABLE insegnante (
    idInsegnante INT PRIMARY KEY AUTO_INCREMENT,
    nome VARCHAR(100) NOT NULL,
    cognome VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL
);

CREATE TABLE corso (
    idCorso INT PRIMARY KEY AUTO_INCREMENT,
    nome VARCHAR(100) NOT NULL,
    idInsegnante INT NOT NULL,
    idClasse INT NOT NULL,
    FOREIGN KEY (idInsegnante) REFERENCES insegnante(idInsegnante),
    FOREIGN KEY (idClasse) REFERENCES classe(idClasse)
);

CREATE TABLE voto (
    idVoto INT PRIMARY KEY AUTO_INCREMENT,
    idStudente INT NOT NULL,
    idCorso INT NOT NULL,
    valore INT CHECK (valore >= 0 AND valore <= 10),
    dataVoto DATE DEFAULT CURDATE(),
    FOREIGN KEY (idStudente) REFERENCES studente(idStudente),
    FOREIGN KEY (idCorso) REFERENCES corso(idCorso)
);
```

Listing 13.7: Creazione Tabelle Scuola

13.6 Esercizio 6: Ristorante

13.6.1 Descrizione

Modellare un ristorante: tavoli, piatti nel menu, ordini dei clienti e prenotazioni. Ogni prenotazione è per un tavolo in una data/ora specifica.

13.6.2 Schema SQL

```sql
CREATE TABLE piatto (
    idPiatto INT PRIMARY KEY AUTO_INCREMENT,
    nome VARCHAR(100) NOT NULL,
    prezzo DECIMAL(8, 2) NOT NULL,
    categoria ENUM('Antipasto', 'Primo', 'Secondo', 'Dolce', 'Bevanda')
);

CREATE TABLE tavolo (
    idTavolo INT PRIMARY KEY AUTO_INCREMENT,
    numero INT UNIQUE NOT NULL,
    numPosti INT
);

CREATE TABLE prenotazione (
    idPrenotazione INT PRIMARY KEY AUTO_INCREMENT,
    idTavolo INT NOT NULL,
    nomeCliente VARCHAR(100) NOT NULL,
    dataPrenotazione DATE NOT NULL,
    oraPrenotazione TIME NOT NULL,
    numPersone INT,
    FOREIGN KEY (idTavolo) REFERENCES tavolo(idTavolo)
);

CREATE TABLE ordinazione (
    idOrdinazione INT PRIMARY KEY AUTO_INCREMENT,
    idPrenotazione INT,
    idTavolo INT,
    dataOrdinazione DATETIME DEFAULT NOW(),
    FOREIGN KEY (idPrenotazione) REFERENCES prenotazione(idPrenotazione),
    FOREIGN KEY (idTavolo) REFERENCES tavolo(idTavolo)
);

CREATE TABLE dettaglioOrdinazione (
    idDettaglio INT PRIMARY KEY AUTO_INCREMENT,
    idOrdinazione INT NOT NULL,
    idPiatto INT NOT NULL,
    quantità INT DEFAULT 1,
    FOREIGN KEY (idOrdinazione) REFERENCES ordinazione(idOrdinazione),
    FOREIGN KEY (idPiatto) REFERENCES piatto(idPiatto)
);
```

Listing 13.8: Creazione Tabelle Ristorante

Esercizi Intermedi (7-13)

13.7 Esercizio 7: Query con JOIN Complessi

13.7.1 Descrizione

Usando il database E-commerce (Esercizio 3), scrivi le seguenti query:

```sql
-- 1. Elenco clienti con numero totale di ordini e importo speso
SELECT
    c.idCliente,
    c.nome,
    COUNT(DISTINCT o.idOrdine) AS numOrdini,
    SUM(d.quantità * d.prezzoUnitario) AS totalSpeso
FROM cliente c
LEFT JOIN ordine o ON c.idCliente = o.idCliente
LEFT JOIN dettaglioOrdine d ON o.idOrdine = d.idOrdine
GROUP BY c.idCliente, c.nome
ORDER BY totalSpeso DESC;

-- 2. Prodotti mai ordinati
SELECT DISTINCT p.idProdotto, p.nome
FROM prodotto p
WHERE p.idProdotto NOT IN (
    SELECT DISTINCT idProdotto FROM dettaglioOrdine
);

-- 3. Categoria con fatturato più alto
SELECT
    cat.nome,
    SUM(d.quantità * d.prezzoUnitario) AS fatturato
FROM categoria cat
INNER JOIN prodotto p ON cat.idCategoria = p.idCategoria
INNER JOIN dettaglioOrdine d ON p.idProdotto = d.idProdotto
GROUP BY cat.idCategoria, cat.nome
ORDER BY fatturato DESC
LIMIT 1;
```

Listing 13.9: Query E-commerce Intermedie

13.8 Esercizio 8: Subquery e Viste

13.8.1 Descrizione

Usando il database Azienda (Esercizio 2), crea viste e subquery per analizzare i dati.

```sql
-- Creare una vista per stipendi sopra la media
CREATE VIEW dipendentiSopraMedia AS
SELECT d.nome, d.cognome, d.stipendio
FROM dipendente d
WHERE d.stipendio > (SELECT AVG(stipendio) FROM dipendente);

-- Usare la vista
```

```sql
SELECT * FROM dipendentiSopraMedia;

-- Subquery: dipendenti nel dipartimento con stipendio medio più alto
SELECT d.nome, d.cognome, d.stipendio
FROM dipendente d
WHERE d.idDipartimento = (
    SELECT idDipartimento FROM (
        SELECT idDipartimento, AVG(stipendio) AS mediaStipendio
        FROM dipendente
        GROUP BY idDipartimento
        ORDER BY mediaStipendio DESC
        LIMIT 1
    ) AS maxDip
);
```

Listing 13.10: Viste e Subquery Azienda

13.9 Esercizio 9: Aggregazioni Complesse

13.9.1 Descrizione

Usando il database Ospedale (Esercizio 4), analizza dati aggregati.

```sql
-- Numero di pazienti per reparto
SELECT
    r.nome,
    COUNT(DISTINCT p.idPaziente) AS numPazienti,
    COUNT(DISTINCT ri.idRicovero) AS numRicoveri,
    AVG(DATEDIFF(ri.dataFine, ri.dataInizio)) AS durataMediaRicovero
FROM reparto r
LEFT JOIN ricovero ri ON r.idReparto = ri.idReparto
LEFT JOIN paziente p ON ri.idPaziente = p.idPaziente
GROUP BY r.idReparto, r.nome
ORDER BY numPazienti DESC;

-- Medici con più visite
SELECT
    m.nome,
    m.cognome,
    m.specializzazione,
    COUNT(v.idVisita) AS numVisite
FROM medico m
LEFT JOIN visita v ON m.idMedico = v.idMedico
GROUP BY m.idMedico, m.nome, m.cognome, m.specializzazione
ORDER BY numVisite DESC;
```

Listing 13.11: Aggregazioni Ospedale

13.10 Esercizio 10: Transazioni

13.10.1 Descrizione

Simula un trasferimento di fondi tra due conti (usando la tabella cliente). Scrivi una transazione che aggiorna i saldi correttamente.

```sql
START TRANSACTION;

-- 1. Verificare che il cliente mittente abbia abbastanza fondi
SELECT saldo FROM cliente WHERE idCliente = 1 FOR UPDATE;

-- 2. Se OK, debito dal mittente
UPDATE cliente SET saldo = saldo - 100 WHERE idCliente = 1;

-- Savepoint: in caso di errore successivo, rollback fino qui
SAVEPOINT dopo_debito;

-- 3. Credito al destinatario
UPDATE cliente SET saldo = saldo + 100 WHERE idCliente = 2;

-- 4. Registrare la transazione
INSERT INTO movimento (idCliente, tipo, importo, data)
VALUES (1, 'Trasferimento', -100, NOW()),
       (2, 'Trasferimento', 100, NOW());

-- Se tutto OK, salvare
COMMIT;
```

Listing 13.12: Transazione Trasferimento Fondi

13.11 Esercizio 11: Vincoli e Integrità

13.11.1 Descrizione

Aggiungi vincoli a una tabella ordine per garantire integrità.

```sql
CREATE TABLE ordine (
    idOrdine INT PRIMARY KEY AUTO_INCREMENT,
    idCliente INT NOT NULL,
    dataOrdine DATE DEFAULT CURDATE(),
    totale DECIMAL(10, 2) NOT NULL,
    stato ENUM('Pendente', 'Spedito', 'Consegnato') DEFAULT 'Pendente'
        ,
    -- Vincoli
    FOREIGN KEY (idCliente) REFERENCES cliente(idCliente) ON DELETE
        RESTRICT,
    CHECK (totale > 0),
    CHECK (dataOrdine <= CURDATE())  -- Non può essere futuro
);

-- Aggiungere indice per query frequenti
CREATE INDEX idx_cliente ON ordine(idCliente);
```

```sql
CREATE INDEX idx_data ON ordine(dataOrdine);
CREATE INDEX idx_stato ON ordine(stato);
```

Listing 13.13: Vincoli Integrità Ordine

13.12 Esercizio 12: UPDATE e DELETE Complessi

13.12.1 Descrizione

Aggiorna dati basati su condizioni complesse.

```sql
-- Aumentare il prezzo dei prodotti in categoria "Elettronica" del 10%
UPDATE prodotto SET prezzo = prezzo * 1.10
WHERE idCategoria = (SELECT idCategoria FROM categoria WHERE nome = '
    Elettronica');

-- Cancellare ordini annullati più di un anno fa
DELETE FROM dettaglioOrdine
WHERE idOrdine IN (
    SELECT idOrdine FROM ordine
    WHERE stato = 'Cancellato' AND dataOrdine < DATE_SUB(NOW(),
        INTERVAL 1 YEAR)
);

DELETE FROM ordine
WHERE stato = 'Cancellato' AND dataOrdine < DATE_SUB(NOW(), INTERVAL 1
    YEAR);

-- Impostare stato inattivo per clienti che non hanno ordinato negli
    ultimi 2 anni
UPDATE cliente SET stato = 'Inattivo'
WHERE idCliente NOT IN (
    SELECT DISTINCT idCliente FROM ordine
    WHERE dataOrdine > DATE_SUB(NOW(), INTERVAL 2 YEAR)
);
```

Listing 13.14: UPDATE e DELETE Complessi

13.13 Esercizio 13: Ottimizzazione Query

13.13.1 Descrizione

Riscrivi query non ottimizzate in modo più efficiente.

```sql
-- LENTO: Subquery correlata
SELECT c.nome FROM cliente c
WHERE EXISTS (
    SELECT 1 FROM ordine o WHERE o.idCliente = c.idCliente
    AND YEAR(o.dataOrdine) = 2023
);

```

```sql
-- VELOCE: INNER JOIN
SELECT DISTINCT c.nome FROM cliente c
INNER JOIN ordine o ON c.idCliente = o.idCliente
WHERE YEAR(o.dataOrdine) = 2023;

-- LENTO: NOT IN con subquery (NULL issues)
SELECT nome FROM cliente
WHERE idCliente NOT IN (SELECT idCliente FROM ordine);

-- VELOCE: LEFT JOIN con NULL check
SELECT c.nome FROM cliente c
LEFT JOIN ordine o ON c.idCliente = o.idCliente
WHERE o.idOrdine IS NULL;
```

Listing 13.15: Ottimizzazione Query

Esercizi Avanzati (14-20)

13.14 Esercizio 14: Progettazione Completa ER

13.14.1 Descrizione

Progettare un database per una piattaforma di social media (utenti, post, commenti, like, follow).

13.14.2 Schema SQL

```sql
CREATE TABLE utente (
    idUtente INT PRIMARY KEY AUTO_INCREMENT,
    username VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    dataIscrizione DATETIME DEFAULT NOW()
);

CREATE TABLE post (
    idPost INT PRIMARY KEY AUTO_INCREMENT,
    idAutore INT NOT NULL,
    contenuto TEXT NOT NULL,
    dataCreazione DATETIME DEFAULT NOW(),
    FOREIGN KEY (idAutore) REFERENCES utente(idUtente)
);

CREATE TABLE commento (
    idCommento INT PRIMARY KEY AUTO_INCREMENT,
    idPost INT NOT NULL,
    idAutore INT NOT NULL,
    contenuto TEXT NOT NULL,
    dataCreazione DATETIME DEFAULT NOW(),
    FOREIGN KEY (idPost) REFERENCES post(idPost),
    FOREIGN KEY (idAutore) REFERENCES utente(idUtente)
);

CREATE TABLE like (
```

```sql
    idLike INT PRIMARY KEY AUTO_INCREMENT,
    idUtente INT NOT NULL,
    idPost INT,
    idCommento INT,
    dataLike DATETIME DEFAULT NOW(),
    FOREIGN KEY (idUtente) REFERENCES utente(idUtente),
    FOREIGN KEY (idPost) REFERENCES post(idPost),
    FOREIGN KEY (idCommento) REFERENCES commento(idCommento),
    CHECK ((idPost IS NOT NULL AND idCommento IS NULL) OR (idPost IS
        NULL AND idCommento IS NOT NULL))
);

CREATE TABLE follow (
    idFollow INT PRIMARY KEY AUTO_INCREMENT,
    idSeguace INT NOT NULL,
    idSeguito INT NOT NULL,
    dataFollow DATETIME DEFAULT NOW(),
    UNIQUE KEY (idSeguace, idSeguito),
    FOREIGN KEY (idSeguace) REFERENCES utente(idUtente),
    FOREIGN KEY (idSeguito) REFERENCES utente(idUtente),
    CHECK (idSeguace != idSeguito)
);
```

Listing 13.16: Creazione Tabelle Social Media

13.14.3 Query Avanzate

```sql
-- Post più popolari (ordini per like)
SELECT
    p.idPost,
    u.username,
    p.contenuto,
    COUNT(l.idLike) AS numLike,
    COUNT(DISTINCT c.idCommento) AS numCommenti
FROM post p
INNER JOIN utente u ON p.idAutore = u.idUtente
LEFT JOIN like l ON p.idPost = l.idPost
LEFT JOIN commento c ON p.idPost = c.idPost
GROUP BY p.idPost, u.username, p.contenuto
ORDER BY numLike DESC
LIMIT 10;

-- Utenti più seguiti
SELECT
    u.idUtente,
    u.username,
    COUNT(f.idFollow) AS numSeguaci
FROM utente u
LEFT JOIN follow f ON u.idUtente = f.idSeguito
GROUP BY u.idUtente, u.username
ORDER BY numSeguaci DESC
LIMIT 10;

-- Feed personalizzato (post di utenti seguiti)
SELECT
```

```sql
    p.idPost,
    u.username,
    p.contenuto,
    p.dataCreazione
FROM post p
INNER JOIN utente u ON p.idAutore = u.idUtente
WHERE p.idAutore IN (
    SELECT idSeguito FROM follow WHERE idSeguace = 1  -- Utente 1
)
ORDER BY p.dataCreazione DESC
LIMIT 20;
```

Listing 13.17: Query Avanzate Social Media

13.15 Esercizio 15: Case Study: Ecommerce Completo

13.15.1 Descrizione

Espandere il database e-commerce con le seguenti funzionalità: - Magazzino e tracciamento stock - Ordini con indirizzo di spedizione diverso da fatturazione - Sistema di rating e recensioni - Coupon e sconti

13.15.2 Schema SQL Esteso

```sql
-- Tabella magazzino
CREATE TABLE magazzino (
    idMagazzino INT PRIMARY KEY AUTO_INCREMENT,
    nome VARCHAR(100) NOT NULL,
    città VARCHAR(50)
);

CREATE TABLE stock (
    idStock INT PRIMARY KEY AUTO_INCREMENT,
    idProdotto INT NOT NULL,
    idMagazzino INT NOT NULL,
    quantitàDisponibile INT DEFAULT 0,
    UNIQUE KEY (idProdotto, idMagazzino),
    FOREIGN KEY (idProdotto) REFERENCES prodotto(idProdotto),
    FOREIGN KEY (idMagazzino) REFERENCES magazzino(idMagazzino)
);

-- Tabella indirizzi (spedizione e fatturazione)
CREATE TABLE indirizzo (
    idIndirizzo INT PRIMARY KEY AUTO_INCREMENT,
    idCliente INT NOT NULL,
    via VARCHAR(100) NOT NULL,
    civico VARCHAR(10),
    città VARCHAR(50),
    provincia VARCHAR(2),
    cap VARCHAR(5),
    tipo ENUM('Fatturazione', 'Spedizione'),
    FOREIGN KEY (idCliente) REFERENCES cliente(idCliente)
);
```

```sql

-- Tabella recensioni
CREATE TABLE recensione (
    idRecensione INT PRIMARY KEY AUTO_INCREMENT,
    idProdotto INT NOT NULL,
    idCliente INT NOT NULL,
    valutazione INT CHECK (valutazione >= 1 AND valutazione <= 5),
    testo TEXT,
    dataRecensione DATE DEFAULT CURDATE(),
    FOREIGN KEY (idProdotto) REFERENCES prodotto(idProdotto),
    FOREIGN KEY (idCliente) REFERENCES cliente(idCliente)
```

Listing 13.18: Tabelle Ecommerce Esteso

Listing 13.19: Query Ecommerce Complesse

13.16 Esercizio 16: Query con Window Functions

13.16.1 Descrizione

Usa funzioni finestra per analizzare dati temporali.

```sql
-- Ranking dei prodotti per vendite
SELECT
    p.idProdotto,
    p.nome,
    SUM(d.quantità) AS unitàVendute,
    RANK() OVER (ORDER BY SUM(d.quantità) DESC) AS rank
FROM prodotto p
LEFT JOIN dettaglioOrdine d ON p.idProdotto = d.idProdotto
GROUP BY p.idProdotto, p.nome;

-- Totale cumulativo di vendite per data
SELECT
    DATE(dataOrdine) AS data,
    SUM(totale) AS venditeGiorno,
    SUM(SUM(totale)) OVER (ORDER BY DATE(dataOrdine)) AS
        venditeCumulative
FROM ordine
GROUP BY DATE(dataOrdine);

-- Differenza da cliente precedente
SELECT
    idCliente,
    dataRegistrazione,
    LAG(idCliente) OVER (ORDER BY dataRegistrazione) AS
        clientePrecedente,
    LEAD(idCliente) OVER (ORDER BY dataRegistrazione) AS
        clienteSuccessivo
FROM cliente;
```

Listing 13.20: Window Functions

13.17 Esercizio 17: Procedure Stored (Pseudocodice)

13.17.1 Descrizione

Creare una procedura che processa ordini automaticamente.

```sql
-- PSEUDOCODICE (sintassi MySQL)
CREATE PROCEDURE ProcessaOrdine(IN pIdOrdine INT)
BEGIN
    DECLARE vTotale DECIMAL(10, 2);
    DECLARE vIdCliente INT;
    DECLARE vStato VARCHAR(20);

    START TRANSACTION;

    -- Verificare lo stato dell'ordine
    SELECT idCliente, stato INTO vIdCliente, vStato
    FROM ordine WHERE idOrdine = pIdOrdine;

    IF vStato = 'Pendente' THEN
        -- Aggiornare stock
        UPDATE stock SET quantitàDisponibile = quantitàDisponibile - (
            SELECT quantità FROM dettaglioOrdine WHERE idOrdine =
                pIdOrdine
        );

        -- Aggiornare stato ordine
        UPDATE ordine SET stato = 'Elaborato' WHERE idOrdine =
            pIdOrdine;

        -- Registrare nel log
        INSERT INTO logProcessamento (idOrdine, azione, dataAzione)
        VALUES (pIdOrdine, 'Elaborato', NOW());

        COMMIT;
    ELSE
        ROLLBACK;
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Ordine non in
            stato Pendente';
    END IF;
END;
```

Listing 13.21: Procedura Stored - Processare Ordini

13.18 Esercizio 18: Backup e Ripristino

13.18.1 Descrizione

Script per backup e ripristino del database.

```sql
-- BACKUP (eseguire da terminale)
-- mysqldump -u utente -p nomeDatabases > backup_2025-01-01.sql

-- RIPRISTINO
```

```sql
-- mysql -u utente -p nomeDatabases < backup_2025-01-01.sql

-- Creare backup incrementale
FLUSH LOGS;

-- Verificare file di log
SHOW BINARY LOGS;

-- In SQL, esportare dati specifici
SELECT * INTO OUTFILE '/tmp/ordini_2023.csv'
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n'
FROM ordine WHERE YEAR(dataOrdine) = 2023;
```

Listing 13.22: Backup e Ripristino (da terminale)

13.19 Esercizio 19: Monitoring Performance

13.19.1 Descrizione

Analizza e ottimizza le prestazioni delle query.

```sql
-- Abilitare profilo di query (MySQL)
SET profiling = 1;

-- Eseguire query da analizzare
SELECT * FROM cliente WHERE idCliente = 1;

-- Visualizzare risultati del profiling
SHOW PROFILES;
SHOW PROFILE FOR QUERY 1;

-- Analizzare query con EXPLAIN
EXPLAIN SELECT * FROM ordine o
INNER JOIN cliente c ON o.idCliente = c.idCliente
WHERE c.città = 'Milano';

-- EXPLAIN esteso
EXPLAIN FORMAT=JSON SELECT * FROM ordine WHERE dataOrdine > '
    2023-01-01';

-- Visualizzare indici di una tabella
SHOW INDEX FROM ordine;

-- Analizzare tabella per ottimizzarla
ANALYZE TABLE ordine;
```

Listing 13.23: Monitoring Performance

13.20 Esercizio 20: Database Replicato

13.20.1 Descrizione

Configurare replica master-slave per alta disponibilità (pseudocodice).

```
-- SUL MASTER
-- 1. Verificare status
SHOW MASTER STATUS;

-- 2. Creare utente per replicazione
CREATE USER 'repl'@'slave_ip' IDENTIFIED BY 'password';
GRANT REPLICATION SLAVE ON *.* TO 'repl'@'slave_ip';

-- SUL SLAVE
-- 1. Configurare connessione al master
CHANGE MASTER TO
    MASTER_HOST='master_ip',
    MASTER_USER='repl',
    MASTER_PASSWORD='password',
    MASTER_LOG_FILE='mysql-bin.000001',
    MASTER_LOG_POS=154;

-- 2. Avviare replicazione
START SLAVE;

-- 3. Verificare status replicazione
SHOW SLAVE STATUS;

-- Se ci sono errori
SHOW SLAVE STATUS\G;
-- Verificare Slave_IO_Running e Slave_SQL_Running siano "Yes"
```

Listing 13.24: Replica Master-Slave (Configurazione)

13.21 Riepilogo Esercizi

Competenze Acquisite

Completando questi 20 esercizi, avrai acquisito competenze in:

- Progettazione ER da zero
- Normalizzazione e schema SQL
- JOIN complessi su multiple tabelle
- Aggregazioni e GROUP BY
- Subquery e viste
- Transazioni e ACID
- Ottimizzazione query
- Gestione di database reali
- Sicurezza e integrità dati
- Monitoraggio e performance

Soluzioni Esercizi

Introduzione

Questo capitolo contiene le soluzioni dettagliate di tutti gli esercizi del Capitolo 13. Le soluzioni sono accompagnate da spiegazioni per aiutare la comprensione della logica sottostante.

.1 Soluzioni Esercizi Base (1-6)

Esercizio 1: Biblioteca Semplice

Soluzione 1.1: Inserimento Dati

```sql
-- Inserire autori
INSERT INTO autore (nome, email) VALUES
('Alessandro Manzoni', 'manzoni@example.com'),
('Dante Alighieri', 'dante@example.com'),
('Carlo Collodi', 'collodi@example.com');

-- Inserire libri
INSERT INTO libro (isbn, titolo, idAutore, copieDisponibili) VALUES
('9788804542606', 'I Promessi Sposi', 1, 5),
('9788801040241', 'La Divina Commedia', 2, 3),
('9788835903567', 'Le Avventure di Pinocchio', 3, 7);

-- Inserire clienti
INSERT INTO cliente (nome, cognome, telefono) VALUES
('Rossi', 'Mario', '3331234567'),
('Bianchi', 'Luigi', '3337654321'),
('Verdi', 'Anna', '3329876543');

-- Inserire prestiti
INSERT INTO prestito (idLibro, idCliente, dataPrestito,
    dataRestituzione) VALUES
(1, 1, '2024-01-15', '2024-02-15'),
(2, 1, '2024-02-01', NULL),   -- Prestito attivo
(1, 2, '2024-02-10', NULL),   -- Prestito attivo
(3, 3, '2024-01-20', '2024-02-20');
```

Listing 25: Inserire Dati Biblioteca

Soluzione 1.2: Query Biblioteca

```sql
-- 1. Libri di Dante
SELECT titolo, isbn FROM libro
WHERE idAutore = (SELECT idAutore FROM autore WHERE nome = 'Dante Alighieri');
-- Risultato: La Divina Commedia

-- 2. Prestiti attivi
SELECT COUNT(*) AS prestitiAttivi FROM prestito
WHERE dataRestituzione IS NULL;
-- Risultato: 2

-- 3. Libri più prestati
SELECT l.titolo, COUNT(p.idPrestito) AS numPrestiti
FROM libro l
LEFT JOIN prestito p ON l.idLibro = p.idLibro
GROUP BY l.idLibro, l.titolo
ORDER BY numPrestiti DESC;
-- Risultato:
-- | I Promessi Sposi | 2
-- | Le Avventure di Pinocchio | 1
-- | La Divina Commedia | 1

-- 4. Media giorni prestito (solo prestiti conclusi)
SELECT
    l.titolo,
    AVG(DATEDIFF(p.dataRestituzione, p.dataPrestito)) AS mediaDiorniPrestito
FROM libro l
INNER JOIN prestito p ON l.idLibro = p.idLibro
WHERE p.dataRestituzione IS NOT NULL
GROUP BY l.idLibro, l.titolo;
-- Risultato: I Promessi Sposi = 31, Le Avventure Pinocchio = 31

-- 5. Clienti con prestiti attivi
SELECT DISTINCT c.nome, c.cognome
FROM cliente c
INNER JOIN prestito p ON c.idCliente = p.idCliente
WHERE p.dataRestituzione IS NULL;
-- Risultato: Rossi Mario, Rossi Mario
```

Listing 26: Query Soluzione Biblioteca

Esercizio 2: Azienda e Dipendenti

Soluzione 2.1: Inserimento Dati

```sql
-- ATTENZIONE: Dipendenti senza responsabile inizialmente
INSERT INTO dipendente (nome, cognome, stipendio, idDipartimento)
    VALUES
('Rossi', 'Paolo', 3500, 1),
('Bianchi', 'Maria', 2800, 1),
```

```
('Verdi', 'Luigi', 4200, 2),
('Neri', 'Anna', 3000, 2),
('Ferrari', 'Giovanni', 2500, 3);

-- Creare dipartimenti (inizialmente senza responsabile)
INSERT INTO dipartimento (nome, idResponsabile) VALUES
('IT', 1),
('Vendite', 3),
('Amministrazione', 5);

-- Aggiornare responsabili
UPDATE dipartimento SET idResponsabile = 1 WHERE idDipartimento = 1;
UPDATE dipartimento SET idResponsabile = 3 WHERE idDipartimento = 2;
UPDATE dipartimento SET idResponsabile = 5 WHERE idDipartimento = 3;
```

Listing 27: Inserire Dati Azienda

Soluzione 2.2: Query Azienda

```
-- 1. Stipendio medio per dipartimento
SELECT
    d.nome AS dipartimento,
    COUNT(dip.idDipendente) AS numDipendenti,
    AVG(dip.stipendio) AS stipendioMedio,
    MIN(dip.stipendio) AS stipendioMin,
    MAX(dip.stipendio) AS stipendioMax
FROM dipartimento d
LEFT JOIN dipendente dip ON d.idDipartimento = dip.idDipartimento
GROUP BY d.idDipartimento, d.nome
ORDER BY stipendioMedio DESC;

-- 2. Dipendenti sopra media dipartimento
SELECT
    dip.nome,
    dip.cognome,
    d.nome AS dipartimento,
    dip.stipendio,
    (SELECT AVG(stipendio) FROM dipendente WHERE idDipartimento = d.
       idDipartimento) AS mediaReparto
FROM dipendente dip
INNER JOIN dipartimento d ON dip.idDipartimento = d.idDipartimento
WHERE dip.stipendio > (
    SELECT AVG(stipendio) FROM dipendente dip2
    WHERE dip2.idDipartimento = d.idDipartimento
);

-- 3. Responsabile di ogni dipartimento
SELECT
    d.nome AS dipartimento,
    dip.nome AS responsabile,
    dip.cognome,
    dip.stipendio
FROM dipartimento d
INNER JOIN dipendente dip ON d.idResponsabile = dip.idDipendente;

```

```sql
-- 4. Totale spese stipendi per dipartimento
SELECT
    d.nome,
    SUM(dip.stipendio) AS totaleCosti
FROM dipartimento d
INNER JOIN dipendente dip ON d.idDipartimento = dip.idDipartimento
GROUP BY d.idDipartimento, d.nome
ORDER BY totaleCosti DESC;
```

Listing 28: Query Soluzione Azienda

Esercizio 3: E-commerce Semplice

Soluzione 3.1: Inserimento Dati

```sql
-- Inserire categorie
INSERT INTO categoria (nome) VALUES ('Elettronica'), ('Libri'), ('
    Abbigliamento');

-- Inserire prodotti
INSERT INTO prodotto (nome, prezzo, idCategoria) VALUES
('Laptop Dell', 899.99, 1),
('Harry Potter', 15.99, 2),
('T-Shirt Blu', 25.99, 3),
('Mouse Wireless', 29.99, 1),
('Pantaloni Neri', 45.99, 3);

-- Inserire clienti
INSERT INTO cliente (nome, email) VALUES
('Mario Rossi', 'mario@example.com'),
('Anna Bianchi', 'anna@example.com'),
('Luigi Verdi', 'luigi@example.com');

-- Inserire ordini
INSERT INTO ordine (idCliente, dataOrdine, stato) VALUES
(1, '2024-01-10', 'Consegnato'),
(1, '2024-02-05', 'Spedito'),
(2, '2024-02-15', 'Pendente'),
(3, '2024-01-20', 'Consegnato');

-- Inserire dettagli ordini
INSERT INTO dettaglioOrdine (idOrdine, idProdotto, quantità,
    prezzoUnitario) VALUES
(1, 1, 1, 899.99),      -- Ordine 1: 1x Laptop
(1, 4, 2, 29.99),       -- Ordine 1: 2x Mouse
(2, 2, 3, 15.99),       -- Ordine 2: 3x Harry Potter
(3, 3, 1, 25.99),       -- Ordine 3: 1x T-Shirt
(4, 5, 2, 45.99);       -- Ordine 4: 2x Pantaloni
```

Listing 29: Inserire Dati E-commerce

Soluzione 3.2: Query E-commerce

```sql
-- 1. Reddito totale per cliente
SELECT
    c.nome,
    c.email,
    COUNT(DISTINCT o.idOrdine) AS numOrdini,
    SUM(d.quantità * d.prezzoUnitario) AS totalSpeso
FROM cliente c
LEFT JOIN ordine o ON c.idCliente = o.idCliente
LEFT JOIN dettaglioOrdine d ON o.idOrdine = d.idOrdine
GROUP BY c.idCliente, c.nome, c.email
ORDER BY totalSpeso DESC;

-- 2. Prodotti venduti più di una volta
SELECT
    p.nome,
    SUM(d.quantità) AS unitàVendute,
    SUM(d.quantità * d.prezzoUnitario) AS ricavo
FROM prodotto p
INNER JOIN dettaglioOrdine d ON p.idProdotto = d.idProdotto
GROUP BY p.idProdotto, p.nome
HAVING SUM(d.quantità) > 1
ORDER BY unitàVendute DESC;

-- 3. Ordine medio per cliente
SELECT
    c.nome,
    AVG(d.quantità * d.prezzoUnitario) AS importoMedio
FROM cliente c
INNER JOIN ordine o ON c.idCliente = o.idCliente
INNER JOIN dettaglioOrdine d ON o.idOrdine = d.idOrdine
GROUP BY c.idCliente, c.nome;

-- 4. Vista: clienti fedeli (più di 1 ordine)
CREATE VIEW clientiFedeli AS
SELECT
    c.idCliente,
    c.nome,
    COUNT(o.idOrdine) AS numOrdini,
    SUM(d.quantità * d.prezzoUnitario) AS totalSpeso
FROM cliente c
INNER JOIN ordine o ON c.idCliente = o.idCliente
INNER JOIN dettaglioOrdine d ON o.idOrdine = d.idOrdine
GROUP BY c.idCliente, c.nome
HAVING COUNT(o.idOrdine) > 1;

SELECT * FROM clientiFedeli;
```

Listing 30: Query Soluzione E-commerce

Esercizio 4-6: Soluzioni Analoghe

Per i seguenti esercizi (Ospedale, Scuola, Ristorante), le soluzioni seguono il medesimo schema:

1. **Inserimento dati**: Popolare tabelle con dati significativi

2. **Query di base**: SELECT con WHERE, ORDER BY
3. **Query con JOIN**: Combinare dati da multiple tabelle
4. **Aggregazioni**: COUNT, SUM, AVG con GROUP BY
5. **Subquery**: Usare query annidate per analisi complesse

Dato lo spazio, fornirò schemi generici applicabili.

.2 Soluzioni Esercizi Intermedi (7-13)

Esercizio 7: Query con JOIN Complessi

Spiegazione Soluzione

La prima query usa LEFT JOIN per includere clienti anche senza ordini:

- **LEFT JOIN ordine**: Preserva tutti i clienti
- **LEFT JOIN dettaglioOrdine**: Conserva ordini anche senza dettagli
- **GROUP BY**: Raggruppa per cliente
- **ORDER BY totalSpeso DESC**: Clienti più importanti primo

La query per prodotti mai ordinati usa NOT IN con subquery:

- Subquery interna: Tutti gli idProdotto presenti in dettaglioOrdine
- NOT IN: Filtra prodotti assenti dalla subquery

Esercizio 8: Subquery e Viste

Spiegazione Viste

Le viste semplificano query complesse:

```
-- Vista 1: Dipendenti sopra media
CREATE VIEW dipendentiSopraMedia AS
SELECT d.idDipendente, d.nome, d.cognome, d.stipendio
FROM dipendente d
WHERE d.stipendio > (SELECT AVG(stipendio) FROM dipendente);

-- Uso: semplice e riutilizzabile
SELECT nome, cognome FROM dipendentiSopraMedia
WHERE idDipendente IN (SELECT idResponsabile FROM dipartimento);

-- Vista 2: Statistiche dipartimento
CREATE VIEW statisticheDipartimento AS
SELECT
    d.idDipartimento,
```

```sql
    d.nome,
    COUNT(dip.idDipendente) AS numDipendenti,
    AVG(dip.stipendio) AS mediaStipendio,
    SUM(dip.stipendio) AS totaleCosti
FROM dipartimento d
LEFT JOIN dipendente dip ON d.idDipartimento = dip.idDipartimento
GROUP BY d.idDipartimento, d.nome;

-- Uso
SELECT * FROM statisticheDipartimento
WHERE mediaStipendio > 3000;
```

Listing 31: Viste per Azienda

Esercizio 10: Transazioni

Spiegazione Transazione

La transazione garantisce atomicità:

```sql
START TRANSACTION;

-- Step 1: Lockare la riga per update (FOR UPDATE)
-- Impedisce altri UPDATE contemporanei
SELECT saldo FROM cliente WHERE idCliente = 1 FOR UPDATE;

-- Step 2: Verificare che ci siano fondi sufficienti
-- Se saldo < 100, rollback manualmente

-- Step 3: Debitare mittente
UPDATE cliente SET saldo = saldo - 100 WHERE idCliente = 1;

-- Step 4: Savepoint (punto di ripristino parziale)
SAVEPOINT dopo_debito;

-- Step 5: Accreditare destinatario
-- Se fallisce, ROLLBACK TO SAVEPOINT ripristina solo questo
UPDATE cliente SET saldo = saldo + 100 WHERE idCliente = 2;

-- Step 6: Registrare la transazione
INSERT INTO movimento (idCliente, tipo, importo, data)
VALUES (1, 'Trasferimento Debito', -100, NOW()),
       (2, 'Trasferimento Credito', 100, NOW());

-- Step 7: Se tutto OK, COMMIT rende permanente tutto
COMMIT;
-- Se errore: ROLLBACK annulla tutto o ROLLBACK TO SAVEPOINT
   ripristina parzialmente
```

Listing 32: Transazione Trasferimento - Spiegazione

Esercizio 11: Vincoli e Integrità

Spiegazione Vincoli

```sql
-- PRIMARY KEY: Nessun duplicato, nessun NULL
-- Viola: INSERT INTO ordine (dataOrdine) VALUES ...;  -- Errore: NULL in PK

-- FOREIGN KEY ON DELETE RESTRICT: Proteggere integrità referenziale
-- Viola: DELETE FROM cliente WHERE idCliente = 1;  -- Errore: ordini dipendono da questo cliente

-- CHECK (totale > 0): Validare dati
-- Viola: INSERT INTO ordine (idCliente, totale) VALUES (1, -100);  -- Errore: totale negativo

-- CHECK (dataOrdine <= CURDATE()): No date future
-- Viola: INSERT INTO ordine (idCliente, dataOrdine, totale) VALUES (1, '2099-01-01', 100);  -- Errore

-- Indici migliorano query
CREATE INDEX idx_cliente ON ordine(idCliente);
-- Ricerca per cliente è ora O(log n) invece di O(n)
SELECT * FROM ordine WHERE idCliente = 1;  -- Velocissimo
```

Listing 33: Vincoli Garantiscono Integrità

Esercizio 12: UPDATE e DELETE Complessi

Spiegazione UPDATE con Subquery

```sql
-- Aumentare prezzo categoria: UPDATE con subquery
UPDATE prodotto SET prezzo = prezzo * 1.10
WHERE idCategoria = (
    SELECT idCategoria FROM categoria WHERE nome = 'Elettronica'
);
-- Step 1: Subquery trova idCategoria = 1 (Elettronica)
-- Step 2: UPDATE applica a tutti i prodotti con idCategoria = 1
-- Step 3: Risultato: prezzi aumentati del 10%

-- DELETE con JOIN implicito
DELETE FROM dettaglioOrdine
WHERE idOrdine IN (
    SELECT idOrdine FROM ordine
    WHERE stato = 'Cancellato' AND dataOrdine < DATE_SUB(NOW(), INTERVAL 1 YEAR)
);
-- Elimina righe di dettaglio dei vecchi ordini cancellati
-- Dopo aver eliminato dettagli, eliminare gli ordini stessi
```

Listing 34: UPDATE Complesso

.3 Soluzioni Esercizi Avanzati (14-20)

Esercizio 14: Social Media - Query Avanzate

Soluzione: Feed Personalizzato

```sql
-- Versione 1: Semplice (feed di chi seguo)
SELECT
    p.idPost,
    u.username,
    p.contenuto,
    p.dataCreazione,
    COUNT(l.idLike) AS numLike
FROM post p
INNER JOIN utente u ON p.idAutore = u.idUtente
LEFT JOIN like l ON p.idPost = l.idPost
WHERE p.idAutore IN (
    -- Subquery: ID degli utenti che seguo
    SELECT idSeguito FROM follow WHERE idSeguace = 1
)
GROUP BY p.idPost, u.username, p.contenuto, p.dataCreazione
ORDER BY p.dataCreazione DESC
LIMIT 20;

-- Versione 2: Feed completo (post e commenti)
SELECT
    'Post' AS tipo,
    p.idPost AS id,
    u.username,
    p.contenuto,
    p.dataCreazione,
    COUNT(DISTINCT l.idLike) AS likes,
    COUNT(DISTINCT c.idCommento) AS commenti
FROM post p
INNER JOIN utente u ON p.idAutore = u.idUtente
LEFT JOIN like l ON p.idPost = l.idPost AND l.idCommento IS NULL
LEFT JOIN commento c ON p.idPost = c.idPost
WHERE p.idAutore IN (SELECT idSeguito FROM follow WHERE idSeguace = 1)
GROUP BY p.idPost
UNION ALL
SELECT
    'Commento' AS tipo,
    c.idCommento,
    u.username,
    c.contenuto,
    c.dataCreazione,
    COUNT(l.idLike),
    0
FROM commento c
INNER JOIN utente u ON c.idAutore = u.idUtente
LEFT JOIN like l ON c.idCommento = l.idCommento
WHERE c.idPost IN (
    SELECT p.idPost FROM post p
    WHERE p.idAutore IN (SELECT idSeguito FROM follow WHERE idSeguace
        = 1)
)
```

```
50 GROUP BY c.idCommento
51 ORDER BY dataCreazione DESC;
```

Listing 35: Feed Personalizzato Spiegazione

Esercizio 15: Ecommerce - Gestione Stock

Soluzione: Aggiornamento Stock su Ordine

```
1  START TRANSACTION;
2
3  -- 1. Verificare disponibilità stock
4  SELECT quantitàDisponibile FROM stock
5  WHERE idProdotto = 1 AND idMagazzino = 1
6  FOR UPDATE;  -- Lock per evitare race condition
7
8  -- 2. Se OK, creare ordine
9  INSERT INTO ordine (idCliente, dataOrdine, stato)
10 VALUES (1, NOW(), 'Pendente');
11
12 SET @idOrdine = LAST_INSERT_ID();
13
14 -- 3. Aggiungere dettagli ordine
15 INSERT INTO dettaglioOrdine (idOrdine, idProdotto, quantità,
       prezzoUnitario)
16 VALUES (@idOrdine, 1, 2, 899.99);
17
18 -- 4. Aggiornare stock
19 UPDATE stock SET quantitàDisponibile = quantitàDisponibile - 2
20 WHERE idProdotto = 1 AND idMagazzino = 1;
21
22 -- 5. Registrare movimento magazzino
23 INSERT INTO movimentoMagazzino (idProdotto, idMagazzino, quantità,
       tipo, data)
24 VALUES (1, 1, -2, 'Vendita', NOW());
25
26 COMMIT;
```

Listing 36: Transazione Ordine con Stock

Esercizio 16: Window Functions

Soluzione: Analisi Vendite Temporale

```
1 -- Ranking semplice
2 SELECT
3     p.idProdotto,
4     p.nome,
5     SUM(d.quantità) AS unitàVendute,
6     -- RANK() assegna rank (1, 2, 3 o 1, 2, 2, 4 con pareggi)
```

```
    RANK() OVER (ORDER BY SUM(d.quantità) DESC) AS rank,
    -- ROW_NUMBER() numerazione sequenziale
    ROW_NUMBER() OVER (ORDER BY SUM(d.quantità) DESC) AS row_num
FROM prodotto p
LEFT JOIN dettaglioOrdine d ON p.idProdotto = d.idProdotto
GROUP BY p.idProdotto, p.nome;

-- Totale cumulativo
SELECT
    DATE(dataOrdine) AS data,
    SUM(totale) AS venditeGiorno,
    -- SUM con finestra ORDER BY crea cumulative
    SUM(SUM(totale)) OVER (ORDER BY DATE(dataOrdine)) AS
        venditeCumulative,
    -- Percentuale del totale
    ROUND(100 * SUM(totale) / SUM(SUM(totale)) OVER (), 2) AS percento
FROM ordine
GROUP BY DATE(dataOrdine)
ORDER BY data;

-- Valori precedenti e successivi
SELECT
    idCliente,
    dataRegistrazione,
    -- LAG: valore della riga precedente
    LAG(dataRegistrazione) OVER (ORDER BY dataRegistrazione) AS
        registrazionePrecedente,
    -- LEAD: valore della riga successiva
    LEAD(dataRegistrazione) OVER (ORDER BY dataRegistrazione) AS
        registrazioneSuccessiva,
    -- Giorni da client precedente
    DATEDIFF(
        dataRegistrazione,
        LAG(dataRegistrazione) OVER (ORDER BY dataRegistrazione)
    ) AS giorniDaPrecedente
FROM cliente;
```

Listing 37: Window Functions Spiegazione

Esercizio 20: Monitoraggio Replica

Soluzione: Verificare Stato Replica

```
-- Sul SLAVE: controllare status
SHOW SLAVE STATUS\G;

-- Output importante:
-- Slave_IO_Running: Yes (connesso al master)
-- Slave_SQL_Running: Yes (esecuzione OK)
-- Seconds_Behind_Master: 0 (sincronizzato)
-- Last_Error: (nessun errore)

-- Se Slave_IO_Running = No: problema connessione
-- Verificare:
```

```
SHOW PROCESSLIST;  -- Cercare "Connecting to master"

-- Se Slave_SQL_Running = No: errore SQL
-- Utilizzare:
SHOW SLAVE STATUS\G;  -- Legge Last_Error

-- Se c'è errore, saltare evento e riprovare
SET GLOBAL SQL_SLAVE_SKIP_COUNTER = 1;
START SLAVE;

-- Sincronizzare manualmente se diverge
-- 1. Su master: SHOW MASTER STATUS;
-- 2. Su slave: CHANGE MASTER TO ...
-- 3. Su slave: START SLAVE;
```

Listing 38: Debugging Replica

.4 Riepilogo Strategia Soluzione

Approccio Metodico

Come Affrontare Esercizi SQL

1. **Leggere attentamente**: Comprendere bene cosa chiede l'esercizio
2. **Disegnare ER**: Visualizzare le relazioni tra tabelle
3. **Creare schema**: CREATE TABLE con vincoli appropriati
4. **Inserire test data**: Dati significativi per testare
5. **Scrivere query passo passo**: Iniziare semplice, poi complicare
6. **Testare risultati**: Verificare che query restituiscono dati attesi
7. **Ottimizzare**: Aggiungere indici, riscrivere query lente
8. **Documentare**: Commentare query complesse

Errori Comuni e Soluzioni

NULL non funziona con IN Usa EXISTS per subquery che potrebbero contenere NULL

Cartesiano accidentale Verifica sempre le condizioni JOIN, usa ON, non WHERE

GROUP BY incompleto Tutte le colonne SELECT non aggregate devono essere in GROUP BY

Subquery lenta Usa JOIN instead of subquery when possible

Deadlock Sempre aggiornare righe nello stesso ordine, minimizzare transazione

Vincolo violato Verifica FK, UNIQUE, CHECK prima di INSERT/UPDATE

Query non corrisponde Aggiungi WHERE claus, non contare su ORDER BY

.5 Esercizi Aggiuntivi per Pratica

Per ulteriore pratica, prova a modificare gli esercizi:

1. Aggiungi colonne: dataModifica, utente$_m odificaperaudit$

1. Aggiungi vincoli: CHECK per validare dati (email, telefono)
2. Crea viste per report frequenti
3. Scrivi trigger per mantenere dati sincronizzati
4. Aggiungi ruoli/permessi per controllo accesso
5. Implementa soft delete (colonna isDeleted) anziché DELETE fisico
6. Crea tabelle storia per audit trail
7. Aggiungi partitioning per grandi tabelle

Bibliografia e Risorse

Introduzione

Questo capitolo fornisce una lista curata di risorse per approfondire i concetti di database, SQL e progettazione. Include libri classici, articoli accademici, risorse online e strumenti pratici.

.6 Libri Consigliati

.6.1 Testi Fondamentali

Database System Concepts di Abraham Silberschatz, Henry Korth, S. Sudarshan

- *Casa editrice*: McGraw-Hill
- *Livello*: Intermedio-Avanzato
- *Argomenti*: Teoria relazionale, query optimization, concurrency control, recovery
- *Considerazione*: Testo accademico più completo, ideale per approfondimento teorico

SQL Performance Explained di Markus Winand

- *Casa editrice*: Self-published
- *Livello*: Intermedio-Avanzato
- *Argomenti*: Indici, query execution plans, ottimizzazione
- *Considerazione*: Pratico e focato su performance, linguaggio semplice

Learning SQL di Alan Beaulieu

- *Casa editrice*: O'Reilly
- *Livello*: Base-Intermedio
- *Argomenti*: SQL fundamentals, joins, subqueries, set operations
- *Considerazione*: Ottimo per principianti, esempi chiari, esercizi progressivi

MySQL Cookbook di Paul DuBois

- *Casa editrice*: O'Reilly
- *Livello*: Base-Intermedio
- *Argomenti*: Query comuni, administration, troubleshooting specifico MySQL
- *Considerazione*: Reference veloce per query standard

.6.2 Progettazione e Normalizzazione

Relational Database Design di C.J. Date

- *Livello*: Avanzato
- *Argomenti*: Teoria relazionale pura, normalizzazione, integrità referenziale
- *Note*: Denso ma fondamentale per comprendere i principi sottostanti

Database Design for Mere Mortals di Michael J. Hernandez

- *Livello*: Base-Intermedio
- *Argomenti*: Progettazione ER, normalizzazione, best practices
- *Note*: Accessibile, focalizzato su design pratico, non teorico

.6.3 Amministrazione Database

MySQL 8.0 Reference Manual (Documentazione ufficiale)

- *Disponibile*: online su dev.mysql.com
- *Livello*: Tutti i livelli
- *Contenuto*: Guida completa, sempre aggiornata
- *Note*: Prima fonte per MySQL specifico

High Performance MySQL di Baron Schwartz, Vadim Tkachenko, Peter Zaitsev

- *Livello*: Intermedio-Avanzato
- *Argomenti*: Optimization, replication, high availability, sharding
- *Note*: Essenziale per production databases

.7 Risorse Online

.7.1 Documentazione Ufficiale

MySQL Official Documentation
- *URL*: `https://dev.mysql.com/doc/`
- *Contenuto*: Documentazione completa per tutte le versioni
- *Consiglio*: Bookmarkare le sezioni *Language Reference* e *InnoDB*

PostgreSQL Documentation
- *URL*: `https://www.postgresql.org/docs/`
- *Contenuto*: Excellente per analizzare alternative SQL (WINDOW FUNCTIONS, CTE)
- *Consiglio*: Leggere per imparare best practices applicabili a tutti i DBMS

SQL Standard Documentation
- *URL*: ISO/IEC 9075 (non libero, ma citato in documentazione)
- *Alternativa*: Wikipedia SQL, MDN SQL Guide
- *Consiglio*: Capire lo standard per scrivere SQL portabile

.7.2 Tutorial e Corsi

SQLZoo • *URL*: `https://sqlzoo.net/`

- *Tipo*: Interactive tutorials
- *Livello*: Base-Intermedio
- *Vantaggi*: Impara praticando, feedback immediato

Mode Analytics SQL Tutorial • *URL*: `https://mode.com/sql-tutorial/`

- *Tipo*: Tutorial passo passo con esercizi
- *Livello*: Base-Intermedio
- *Vantaggi*: Pratico, ben strutturato, gratis

LeetCode Database Problems • *URL*: `https://leetcode.com/problemset/database/`

- *Tipo*: Problemi SQL di difficoltà crescente
- *Livello*: Base-Avanzato
- *Vantaggi*: Sfida il tuo livello, soluzioni commentate

Udemy - Complete MySQL Developer Course • *Docenti*: Various

- *Livello*: Base-Intermedio
- *Vantaggi*: Video, esercizi pratici, supporto
- *Costo*: Spesso in offerta

Coursera - Databases Specialization • *Università*: University of Michigan

- *Livello*: Base-Intermedio
- *Vantaggi*: Riconoscimento accademico, strutturato
- *Costo*: Gratuito per audit, certificato a pagamento

.7.3 Blog e Articoli

Use The Index, Luke! • *URL*: `https://use-the-index-luke.com/`

- *Autore*: Markus Winand
- *Tema*: Indici e query optimization
- *Livello*: Intermedio-Avanzato
- *Vantaggi*: Gratuito online, riferimento assoluto per indici

Percona Blog • *URL*: `https://www.percona.com/blog/`

- *Tema*: MySQL performance, troubleshooting, best practices
- *Livello*: Intermedio-Avanzato
- *Vantaggi*: Articoli da esperti del settore

DB Fiddle • *URL*: `https://www.db-fiddle.com/`

- *Tipo*: Editor SQL online
- *Supporta*: MySQL, PostgreSQL, SQL Server, SQLite, Oracle
- *Vantaggi*: Testare query online senza installare nulla

Stack Overflow - Tag SQL • *URL*: `https://stackoverflow.com/questions/tagged/sql`

- *Tipo*: Community Q&A
- *Vantaggi*: Soluzioni a problemi comuni, learning dai dibattiti

.8 Strumenti Pratici

.8.1 DBMS e IDE SQL

MySQL Community Server • *URL*: `https://dev.mysql.com/downloads/mysql/`

- *Costo*: Gratuito, Open Source
- *Piattaforme*: Linux, Windows, macOS
- *Uso*: Produzione e sviluppo

PostgreSQL • *URL*: `https://www.postgresql.org/`

- *Costo*: Gratuito, Open Source
- *Vantaggi*: Più robusto di MySQL, SQL completo, JSONB, hstore
- *Consiglio*: Valida alternativa a MySQL

DBeaver Community • *URL*: `https://dbeaver.io/`

- *Costo*: Gratuito
- *Tipo*: IDE SQL universale
- *Supporta*: MySQL, PostgreSQL, Oracle, SQL Server, SQLite, molti altri
- *Vantaggi*: UI intuitiva, query builder, ER diagram

MySQL Workbench • *URL*: `https://dev.mysql.com/downloads/workbench/`

- *Costo*: Gratuito
- *Tipo*: IDE ufficiale MySQL
- *Vantaggi*: EER diagram design, reverse engineering, sync schema

DataGrip • *URL*: `https://www.jetbrains.com/datagrip/`

- *Costo*: A pagamento (trial gratuito)
- *Tipo*: IDE SQL premium
- *Vantaggi*: Intelligenza artificiale, refactoring, analysis avanzata

VS Code + Extensions
- *Extension*: MySQL (Weijan Chen) o Database Client
- *Costo*: Gratuito
- *Vantaggi*: Leggero, nel vostro editor preferito

.8.2 Strumenti di Analisi e Monitoring

EXPLAIN PLAN Analyzer
- *URL*: `https://www.depesz.com/`
- *Uso*: Paste EXPLAIN output per analizzare visivamente
- *Supporta*: PostgreSQL, MySQL EXPLAIN

MySQL Slow Query Log Analyzer
- *Tool*: mysqldumpslow, pt-query-digest (Percona Toolkit)
- *Uso*: Identificare query lente
- *Consiglio*: Essenziale per production debugging

Prometheus + Grafana
- *Uso*: Monitoring database metrics
- *Costo*: Gratuito
- *Setup*: Più complesso, ma professionale

MySQL Exporter per Prometheus
- *URL*: `https://github.com/prometheus/mysqld_exporter`
- *Uso*: Exportare metriche MySQL per monitoraggio

.9 Articoli Accademici Importanti

.9.1 Teoria Relazionale

A Relational Model of Data di E.F. Codd (1970)
- *Rilevanza*: Paper storico che ha fondato il modello relazionale
- *Disponibilità*: ACM Digital Library
- *Difficoltà*: Accademico, denso

Normalization of Database Relations di E.F. Codd (1972)
- *Tema*: Fondamenti della normalizzazione
- *Rilevanza*: Ancora attuale dopo 50 anni

.9.2 Query Optimization

The Architecture of a Database System di Hellerstein, Stonebraker, Hamilton

- *Disponibilità*: Gratuito online
- *Tema*: Interno di un DBMS, query optimization, transaction management
- *Livello*: Avanzato

Selectivity Estimation - Vari articoli

- *Tema*: Come il query optimizer stima numero di righe
- *Rilevanza*: Capire perché optimizer sceglie certi piani

.10 Comunità e Forum

.10.1 Community Online

MySQL Community Forums • *URL*: `https://forums.mysql.com/`

- *Tipo*: Forum ufficiale MySQL
- *Moderatori*: Team MySQL ufficiale

Database Administrators Stack Exchange • *URL*: `https://dba.stackexchange.com/`

- *Tipo*: Community Q&A per DBA
- *Qualità*: Alta (moderazione rigorosa)

Reddit - r/databases • *URL*: `https://www.reddit.com/r/databases/`

- *Tipo*: Community discussions
- *Vantaggi*: Conversazioni informali, ultimissime news

Hacker News • *URL*: `https://news.ycombinator.com/`

- *Filtro*: Search ”database”
- *Valore*: Scopri trends, discusisoni profonde di esperti

.11 Conferenze e Meetup

Percona Live • *Tipo*: Conferenza internazionale MySQL e database

- *Frequenza*: Annuale
- *Partecipazione*: Relatori di fama mondiale

MySQL User Group • *Ubicazione*: Molte città (locale)

- *Frequenza*: Mensile/trimestrale
- *Costo*: Gratuito

PostgreSQL Meetups
- *Ubicazione*: Mondiale
- *URL*: `https://www.meetup.com/`
- *Consiglio*: Cercare per città

.12 Certificazioni

.12.1 Certificazioni Database

Oracle Certified Associate MySQL Developer
- *Ente*: Oracle
- *Livello*: Base-Intermedio
- *Costo*: Circa 245 USD per esame
- *Preparazione*: MySQL 5.7 Certified Associate Exam Guide
- *Valore*: Riconoscimento professionale

PostgreSQL Certification
- *Ente*: PostgreSQL Association e altri enti
- *Livello*: Vario
- *Costo*: Vario
- *Valore*: Meno noto di Oracle, ma valido

Cloud Database Certifications
- AWS Certified Cloud Practitioner
- Google Cloud Associate Cloud Engineer
- Azure Fundamentals
- *Valore*: Importante nel cloud moderno

.13 Checklist per Padronanza

Prima di considerarsi ”esperto”, assicurati di poter:

- Progettare schema da zero con ER diagram
- Normalizzare fino a BCNF senza errori
- Scrivere query complex con JOIN, subquery, window functions
- Leggere e interpretare EXPLAIN plans
- Ottimizzare query lente
- Configurare indici appropriati
- Gestire transazioni e deadlock
- Backup e recovery di database
- Replicazione e high availability

- Monitoraggio e troubleshooting
- Sicurezza: utenti, ruoli, permessi
- Coding: stored procedures, trigger

.14 Consigli Finali

Come Continuare l'Apprendimento

1. **Pratica costante**: Scrivi SQL ogni giorno, anche piccole query
2. **Leggi codice altrui**: Su GitHub, Stack Overflow, blog tech
3. **Contribuisci open source**: MySQL, PostgreSQL, tool SQL
4. **Sperimenta**: Crea progetti personali, database interessanti
5. **Segui blog/newsletter**: Use The Index Luke, Percona Blog
6. **Partecipa comunità**: Forum, meetup, conferenze
7. **Insegna agli altri**: Spiegare consolida la comprensione
8. **Stai aggiornato**: Database evolvono (MySQL 8.0 features, PostgreSQL 15+)
9. **Comprendi teoria**: Non solo sintassi SQL, capire motivazioni
10. **Misura performance**: Non indovinare, profila sempre

Nota sulla Licenza

Le risorse elencate sono pubblicamente disponibili e la loro inclusione non implica endorsement. Visita i siti ufficiali per dettagli su licenze e termini di utilizzo.