NoiStudiamo

DISPENSA DI LABORATORIO

Basi di Dati
Laboratorio MySQL

DDL e DML in MySQL, query ricorrenti, viste, trigger, stored procedure, indici, backup e guida completa al progetto di una base di dati.

A CURA DI

Damiano La Pusata

Ottobre 2026

INDICE

Indice dei capitoli

I capitoli della dispensa, con il numero della pagina di inizio.

1. MySQL e gli strumenti di lavoro . . . 3
2. Tipi di dato e creazione delle tabelle . . . 6
3. Inserire e modificare i dati, transazioni . . . 9
4. Interrogazioni in MySQL: il ricettario . . . 11
5. Viste . . . 15
6. Trigger in MySQL . . . 16
7. Stored procedure, funzioni ed eventi . . . 20
8. Indici e prestazioni . . . 24
9. Importare, esportare e fare il backup . . . 26
10. Il progetto di una base di dati . . . 27

CAPITOLO 1

MySQL e gli strumenti di lavoro

Che cos'è MySQL, come è organizzato, quali motori di memorizzazione offre e come ci si lavora dalla riga di comando e da MySQL Workbench, compresa la gestione di utenti e privilegi.

Questa dispensa raccoglie la parte pratica di un corso di basi di dati: tutto ciò che serve per scrivere SQL funzionante in MySQL e per realizzare un **progetto completo**, con DDL, dati, operazioni, trigger e stored procedure. I concetti teorici (algebra, normalizzazione, transazioni) sono nella dispensa di teoria; qui li si vede all'opera.

CONVENZIONI DELLA DISPENSA

Nei blocchi di codice i commenti MySQL sono scritti dopo il simbolo #, che MySQL riconosce come inizio di un commento fino a fine riga. Il segno meno è stampato come −, anche davanti alle opzioni dei comandi da terminale: sulla tastiera si digita sempre con il tasto meno. Le date sono scritte nel formato compatto 'AAAAMMGG' (per esempio '20261005') e i valori data e ora come 'AAAAMMGGhhmmss', che MySQL accetta come letterali validi.

Che cos'è MySQL

MySQL è un DBMS relazionale open source, oggi di Oracle (che lo ha acquisito nel 2010 insieme a Sun, che a sua volta lo aveva comprato nel 2008). È composto da un **server** (mysqld), che gestisce i dati, e da **client** che gli inviano comandi: il client testuale mysql, programmi grafici come MySQL Workbench e phpMyAdmin, applicazioni scritte in qualsiasi linguaggio tramite connettori (ODBC, Java, Python, PHP). Esiste un derivato compatibile, MariaDB, nato dopo l'acquisizione da parte di Oracle. Le funzioni più recenti usate in questa dispensa (CHECK applicati, window function, CTE) richiedono MySQL 8.

I motori di memorizzazione

MySQL ha un'architettura a **storage engine**: lo stesso server può memorizzare ogni tabella con un motore diverso, scelto con l'opzione ENGINE di CREATE TABLE. I motori si dividono in **transazionali**, più sicuri perché permettono il rollback e il recupero dopo un crash, e **non transazionali**, più leggeri.

CARATTERISTICA	INNODB	MYISAM
Ruolo	motore predefinito da MySQL 5.5	motore predefinito fino alla 5.4
Transazioni ACID	sì, con COMMIT, ROLLBACK e savepoint	no: ogni modifica è subito definitiva
Chiavi esterne	sì, con integrità referenziale	no: vengono accettate ma ignorate
Lock	a livello di riga	a livello di tabella
Recupero dopo crash	automatico, tramite log	limitato
File su disco	tablespace condiviso o un file .ibd per tabella	tre file: definizione, dati (.MYD), indici (.MYI)

Altri motori sono MEMORY (tabelle in RAM, utili per dati temporanei), CSV, ARCHIVE e BLACKHOLE. Il comando SHOW ENGINES elenca quelli disponibili. Per il progetto si usa sempre **InnoDB**: senza di esso chiavi esterne e transazioni non funzionano.

Connettersi al server

Dal terminale ci si collega con il client mysql indicando host, utente e richiesta della password; se il server è sulla stessa macchina l'host si può omettere. Ogni comando termina con il punto e virgola.

```
mysql -h localhost -u root -p          # -p fa chiedere la password
Enter password: ********
mysql> SHOW DATABASES;                 # elenca i database
mysql> USE cinema;                     # sceglie il database corrente
mysql> SHOW TABLES;                    # tabelle del database corrente
mysql> DESCRIBE Proiezione;            # colonne, tipi, chiavi, default
mysql> SHOW CREATE TABLE Proiezione;   # il CREATE TABLE completo
mysql> SHOW INDEX FROM Proiezione;     # gli indici della tabella
mysql> exit
```

Oltre a mysql, la distribuzione include altri programmi da riga di comando: **mysqladmin** per amministrare il server, **mysqldump** per i backup, **mysqlimport** per importare file di testo, **mysqlcheck** per la manutenzione delle tabelle e **mysqlshow** per esplorare database e tabelle. Il comando SHOW ha molte altre forme utili: SHOW COLUMNS FROM t, SHOW TRIGGERS, SHOW PROCEDURE STATUS, SHOW WARNINGS, SHOW ERRORS, SHOW VARIABLES LIKE 'modello', SHOW PROCESSLIST.

MySQL Workbench

Workbench è lo strumento grafico ufficiale. Le funzioni da conoscere per il progetto sono:

- **connessioni**: dalla schermata iniziale si crea una connessione al server (host, porta 3306, utente) e la si apre;
- **pannello Schemas**: mostra database, tabelle, viste, stored procedure e funzioni; con il tasto destro si creano e modificano gli oggetti con finestre guidate, che generano il codice SQL;
- **editor SQL**: si scrivono ed eseguono script; il fulmine esegue tutto o la selezione, quello con il cursore solo l'istruzione corrente; i risultati compaiono nella griglia sotto;
- **execution plan**: mostra in forma grafica il piano di esecuzione di una query (equivalente a EXPLAIN);
- **modelli EER**: il menu Database, Reverse Engineer genera il diagramma delle tabelle da un database esistente, utile per la relazione del progetto;
- **Administration**: stato del server, client connessi, utenti e privilegi, Data Export e Data Import per backup e ripristino.

ATTENZIONE: SAFE UPDATES

Per evitare disastri, Workbench attiva di default la modalità **safe updates**, che rifiuta UPDATE e DELETE senza una condizione sulla chiave nel WHERE (errore 1175). Si disattiva nelle preferenze dell'editor SQL oppure, per la sessione, con SET SQL_SAFE_UPDATES = 0.

Installazione e database di esempio

MySQL Community Server e MySQL Workbench si scaricano gratuitamente da dev.mysql.com. Su Windows l'installer configura server, Workbench e la password di root in un'unica procedura; su macOS c'è un pacchetto con un pannello nelle Impostazioni di sistema; su Linux si installano i pacchetti della distribuzione (per esempio mysql server, oppure MariaDB) e Workbench a parte. In alternativa, pacchetti come XAMPP installano insieme server web, PHP, un DBMS compatibile e phpMyAdmin, un'interfaccia web per gestire i database dal browser.

Per esercitarsi conviene caricare i database di esempio ufficiali, disponibili nella pagina della documentazione di MySQL: **world** (paesi, città e lingue, poche tabelle, ideale per le prime query) e **sakila** (un noleggio di film con 16 tabelle, viste, trigger e procedure già pronti, utile per vedere un progetto completo). Si importano eseguendo i

rispettivi script, come descritto nel capitolo 9.

PRIMI PASSI CONSIGLIATI

- installa server e Workbench e crea una connessione locale con l'utente root;
- crea un utente personale con GRANT ALL su un database di prova, per non lavorare sempre come root;
- importa world e sakila ed esplorali con SHOW TABLES, DESCRIBE e qualche SELECT;
- riscrivi in MySQL le query degli esercizi della dispensa di teoria e verifica i risultati.

Utenti e privilegi

In MySQL un account è la coppia **'utente'@'host'**: lo stesso nome con host diversi è un account diverso. Gli account si creano con CREATE USER e i privilegi si concedono con GRANT su tutto (*.*), su un database (db.*) o su una tabella (db.tabella). I privilegi più usati sono ALL, SELECT, INSERT, UPDATE, DELETE, CREATE, ALTER, DROP, INDEX, CREATE VIEW, TRIGGER, EXECUTE (per le procedure); USAGE significa "nessun privilegio".

```
CREATE USER 'segreteria'@'localhost' IDENTIFIED BY 'Password1!';
GRANT SELECT, INSERT, UPDATE ON cinema.* TO 'segreteria'@'localhost';
GRANT SELECT ON cinema.Proiezione TO 'segreteria'@'%';  # % = qualsiasi host
SHOW GRANTS FOR 'segreteria'@'localhost';
REVOKE UPDATE ON cinema.* FROM 'segreteria'@'localhost';
DROP USER 'segreteria'@'localhost';
```

CAPITOLO 2

Tipi di dato e creazione delle tabelle

I tipi di MySQL, la sintassi completa di CREATE TABLE con chiavi, vincoli e politiche referenziali, e le modifiche dello schema con ALTER TABLE.

Tipi numerici

TIPO	BYTE	INTERVALLO CON SEGNO	SENZA SEGNO (UNSIGNED)
TINYINT	1	da −128 a 127	da 0 a 255
SMALLINT	2	da −32.768 a 32.767	da 0 a 65.535
MEDIUMINT	3	da −8.388.608 a 8.388.607	da 0 a 16.777.215
INT (INTEGER)	4	circa ±2,1 miliardi	da 0 a 4.294.967.295
BIGINT	8	circa $\pm 9{,}2 \cdot 10^{18}$	da 0 a circa $1{,}8 \cdot 10^{19}$
DECIMAL(M,D)	variabile	numero esatto di M cifre, D decimali	per importi e prezzi
FLOAT / DOUBLE	4 / 8	virgola mobile approssimata	per misure, mai per denaro
BOOLEAN	1	sinonimo di TINYINT(1): 0 falso, 1 vero	TRUE e FALSE valgono 1 e 0

Date, ore e testo

TIPO	BYTE	VALORI
DATE	3	date dall'anno 1000 al 9999
TIME	3	durate o orari da −838:59:59 a 838:59:59
DATETIME	5	data e ora dall'anno 1000 al 9999, senza fuso orario
TIMESTAMP	4	data e ora dal 1 gennaio 1970 al 19 gennaio 2038, convertita in UTC; si aggiorna da sola se definita con ON UPDATE CURRENT_TIMESTAMP
YEAR	1	anni dal 1901 al 2155
CHAR(M)	M	stringa a lunghezza fissa, completata con spazi (M fino a 255)
VARCHAR(M)	L + 1 o L + 2	stringa a lunghezza variabile: occupa la lunghezza effettiva L più 1 byte (2 se oltre 255)
TEXT, BLOB	L + 2	testi e dati binari fino a 64 KB; esistono le varianti TINY, MEDIUM e LONG
ENUM('a', 'b', …)	1 o 2	un valore scelto da un elenco fisso
SET('a', 'b', …)	da 1 a 8	un sottoinsieme di un elenco fisso, al massimo 64 elementi
JSON	variabile	documenti JSON validati, con funzioni dedicate

Scelte consigliate per il progetto: INT AUTO_INCREMENT per le chiavi artificiali, VARCHAR per nomi e descrizioni brevi, TEXT per descrizioni lunghe, DECIMAL per importi, DATE o DATETIME per le date, ENUM o un CHECK per gli attributi con pochi valori ammessi.

CREATE TABLE in MySQL

La sintassi è quella standard vista in teoria con alcune aggiunte. Come esempio, usato in tutta la dispensa, costruiamo il database di un **cinema multisala**: clienti, dipendenti, sale, film, proiezioni e prenotazioni. Ogni prenotazione corrisponde a un posto, e un cliente prenota al più un posto per proiezione.

```
CREATE DATABASE IF NOT EXISTS cinema;
USE cinema;

CREATE TABLE Cliente (
    tessera      INT AUTO_INCREMENT PRIMARY KEY,   # chiave generata dal DBMS
    nome         VARCHAR(100) NOT NULL,
    data_nascita DATE NOT NULL,
    categoria    ENUM('standard', 'studente', 'senior') DEFAULT 'standard'
) ENGINE = InnoDB;

CREATE TABLE Dipendente (
    matricola   INT AUTO_INCREMENT PRIMARY KEY,
    nome        VARCHAR(100) NOT NULL,
    paga_oraria DECIMAL(6,2) NOT NULL CHECK (paga_oraria > 0),
    mansione    VARCHAR(50)
) ENGINE = InnoDB;

CREATE TABLE Sala (
    numero INT PRIMARY KEY,
    piano  TINYINT NOT NULL,
    posti  SMALLINT NOT NULL CHECK (posti > 0),
    ha_3d  BOOLEAN NOT NULL DEFAULT FALSE
) ENGINE = InnoDB;
```

```
CREATE TABLE Film (
    id_film     INT AUTO_INCREMENT PRIMARY KEY,
    titolo      VARCHAR(150) NOT NULL,
    genere      VARCHAR(50),
    durata      SMALLINT NOT NULL CHECK (durata > 0),       # minuti
    eta_minima  TINYINT NOT NULL DEFAULT 0                 # 0, 14 o 18
) ENGINE = InnoDB;

CREATE TABLE Proiezione (
    id_proiezione INT AUTO_INCREMENT PRIMARY KEY,
    film          INT NOT NULL,
    proiezionista INT NOT NULL,
    sala          INT NOT NULL,
    data_ora      DATETIME NOT NULL,
    prezzo        DECIMAL(5,2) NOT NULL,
    posti_venduti INT NOT NULL DEFAULT 0,     # attributo ridondante
    FOREIGN KEY (film) REFERENCES Film(id_film),
    FOREIGN KEY (proiezionista) REFERENCES Dipendente(matricola),
    FOREIGN KEY (sala) REFERENCES Sala(numero)
) ENGINE = InnoDB;
```

```
CREATE TABLE Prenotazione (
    cliente           INT NOT NULL,
    proiezione        INT NOT NULL,
    data_prenotazione DATE NOT NULL,
    pagata            BOOLEAN NOT NULL DEFAULT FALSE,
    PRIMARY KEY (cliente, proiezione),            # chiave composta: un posto per cliente
    FOREIGN KEY (cliente) REFERENCES Cliente(tessera) ON DELETE CASCADE,
    FOREIGN KEY (proiezione) REFERENCES Proiezione(id_proiezione) ON DELETE CASCADE
) ENGINE = InnoDB;
```

CHE COSA NOTARE

- **AUTO_INCREMENT**: il DBMS assegna il valore successivo a ogni inserimento; deve essere una chiave e ce n'è al più uno per tabella. LAST_INSERT_ID() restituisce l'ultimo valore generato nella sessione.
- **CHECK**: da MySQL 8.0.16 i vincoli CHECK sono verificati davvero (prima venivano accettati e ignorati). Non possono contenere sottointerrogazioni né funzioni non deterministiche come CURDATE(): questi vincoli si realizzano con i trigger.
- **FOREIGN KEY**: funzionano solo con InnoDB e richiedono che le colonne collegate abbiano lo stesso tipo; MySQL crea automaticamente un indice sulla chiave esterna se non esiste.
- **ON DELETE / ON UPDATE**: le politiche sono RESTRICT o NO ACTION (rifiuto, predefinite), CASCADE e SET NULL; SET DEFAULT non è supportato da InnoDB.
- **CONSTRAINT nome**: dare un nome ai vincoli (CONSTRAINT fk_proiezione_film FOREIGN KEY …) rende leggibili i messaggi di errore e permette di eliminarli con ALTER TABLE … DROP.

Modificare ed eliminare gli schemi

```
ALTER TABLE Cliente ADD COLUMN email VARCHAR(100) UNIQUE;         # nuova colonna
ALTER TABLE Cliente MODIFY COLUMN nome VARCHAR(150) NOT NULL;     # cambia il tipo
ALTER TABLE Cliente RENAME COLUMN email TO posta;                 # rinomina
ALTER TABLE Cliente DROP COLUMN posta;                            # elimina
ALTER TABLE Proiezione ADD CONSTRAINT ck_prezzo CHECK (prezzo BETWEEN 3 AND 25);
ALTER TABLE Proiezione DROP CHECK ck_prezzo;
ALTER TABLE Proiezione DROP FOREIGN KEY proiezione_ibfk_1;        # nome generato da MySQL
DROP TABLE IF EXISTS Prenotazione;
DROP DATABASE IF EXISTS cinema;
```

L'ordine di creazione e di eliminazione conta: una tabella va creata dopo quelle che riferisce e cancellata prima di esse. Negli script di reinstallazione si può disattivare temporaneamente il controllo con SET FOREIGN_KEY_CHECKS = 0 e riattivarlo alla fine con SET FOREIGN_KEY_CHECKS = 1.

CAPITOLO 3

Inserire e modificare i dati, transazioni

INSERT, UPDATE e DELETE nelle loro forme MySQL, il trattamento dei nulli e la gestione delle transazioni con commit, rollback, savepoint, autocommit e livelli di isolamento.

INSERT

```
# una riga, elencando le colonne (le altre prendono il DEFAULT o NULL)
INSERT INTO Film (titolo, genere, durata, eta_minima)
VALUES ('La lunga notte', 'thriller', 118, 14);

# piu righe in un solo comando
INSERT INTO Sala (numero, piano, posti, ha_3d) VALUES
    (1, 0, 180, TRUE), (2, 1, 90, FALSE), (3, 1, 120, TRUE);

# righe prodotte da una query
INSERT INTO ArchivioProiezioni SELECT * FROM Proiezione WHERE data_ora < '20260101';

# se la chiave esiste gia, aggiorna invece di dare errore
INSERT INTO Prenotazione (cliente, proiezione, data_prenotazione)
VALUES (7, 12, CURDATE())
    ON DUPLICATE KEY UPDATE pagata = TRUE;
```

UPDATE e DELETE

```
UPDATE Dipendente SET paga_oraria = paga_oraria * 1.05
WHERE mansione = 'proiezionista';              # aumento del 5 per cento

DELETE FROM Prenotazione WHERE pagata = FALSE
    AND data_prenotazione < DATE_SUB(CURDATE(), INTERVAL 30 DAY);

# in MySQL UPDATE e DELETE accettano anche ORDER BY e LIMIT
DELETE FROM Proiezione ORDER BY data_ora LIMIT 10;   # le 10 proiezioni piu vecchie
```

Senza WHERE, UPDATE e DELETE agiscono su tutta la tabella. Con le chiavi esterne in RESTRICT, cancellare una riga riferita da altre fallisce (errore 1451); con CASCADE si cancellano anche le righe che la riferiscono.

I valori NULL

NULL significa "mancante o sconosciuto" e non si confronta con =: si usano IS NULL e IS NOT NULL, oppure l'operatore **<=>** (uguaglianza sicura rispetto ai nulli, vera anche se entrambi sono NULL). Le funzioni IFNULL(x, valore) e COALESCE(x, y, …) sostituiscono i nulli; in ORDER BY i NULL vengono per primi in ordine crescente e per ultimi in ordine decrescente. Un'espressione aritmetica o una concatenazione con un NULL vale NULL.

Transazioni in MySQL

Per default MySQL è in modalità **autocommit**: ogni istruzione è una transazione a sé, confermata subito. Per raggruppare più istruzioni si apre esplicitamente una transazione; le tabelle devono essere InnoDB, perché con MyISAM il rollback non è possibile.

```
START TRANSACTION;
    INSERT INTO Prenotazione (cliente, proiezione, data_prenotazione)
    VALUES (7, 12, CURDATE());
    UPDATE Proiezione SET posti_venduti = posti_venduti + 1 WHERE id_proiezione = 12;
    SAVEPOINT dopo_prenotazione;                # punto intermedio
    UPDATE Cliente SET categoria = 'senior' WHERE tessera = 7;
    ROLLBACK TO SAVEPOINT dopo_prenotazione;      # annulla solo l'ultimo UPDATE
COMMIT;                                         # conferma il resto
```

In alternativa SET autocommit = 0 disattiva la conferma automatica: da quel momento tutte le modifiche restano sospese fino a COMMIT o ROLLBACK. COMMIT RELEASE e ROLLBACK RELEASE chiudono anche la connessione. Alcuni comandi, come CREATE TABLE e ALTER TABLE, provocano un commit implicito.

LIVELLI DI ISOLAMENTO E LOCK ESPLICITI

```
SELECT @@transaction_isolation;                          # livello corrente
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;  # per tutta la connessione
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;            # solo per la prossima transazione

START TRANSACTION;
    SELECT posti FROM Sala WHERE numero = 2 FOR UPDATE;           # lock esclusivo
    SELECT * FROM Proiezione WHERE sala = 2 FOR SHARE;            # lock condiviso
COMMIT;
```

Il livello predefinito di InnoDB è **REPEATABLE READ**. Con GLOBAL al posto di SESSION si cambia il valore per tutte le connessioni future (serve un privilegio amministrativo); senza nessuno dei due vale solo per la transazione successiva. **SELECT … FOR UPDATE** blocca in modo esclusivo le righe lette, da usare quando le si legge per poi modificarle; **FOR SHARE** (nelle versioni precedenti LOCK IN SHARE MODE) le blocca in modo condiviso, impedendo agli altri di modificarle fino alla fine della transazione.

CAPITOLO 4

Interrogazioni in MySQL: il ricettario

Funzioni utili e soluzioni pronte per i tipi di query che ricorrono più spesso, scritte e commentate sullo schema del cinema.

Funzioni più usate

CATEGORIA	FUNZIONI	ESEMPIO
Date	CURDATE(), NOW(), YEAR(d), MONTH(d), DAY(d), DATE(dt), DATEDIFF(a, b), DATE_ADD / DATE_SUB(d, INTERVAL n DAY), DATE_FORMAT(d, formato)	DATE_SUB(CURDATE(), INTERVAL 7 DAY) = una settimana fa
Stringhe	CONCAT(a, b), CONCAT_WS(sep, a, b), UPPER, LOWER, LENGTH, CHAR_LENGTH, SUBSTRING(s, pos, n), REPLACE, TRIM, LEFT, RIGHT, LOCATE	CONCAT(nome, ' ', cognome)
Numeri	ROUND(x, d), FLOOR, CEILING, ABS, MOD(a, b), POWER, LOG(b, x)	ROUND(AVG(prezzo), 2)
Condizionali	IF(cond, a, b), IFNULL(x, y), COALESCE(…), NULLIF(a, b), CASE WHEN … THEN … ELSE … END	IF(pagata, 'sì', 'no')
Confronto	BETWEEN, IN, LIKE (% e _), REGEXP, IS NULL, <=>	nome LIKE 'P%'

In MySQL gli operatori logici hanno anche una forma simbolica (! per NOT, && per AND, || per OR) e i confronti restituiscono 1 o 0. Le stringhe si confrontano senza distinguere maiuscole e minuscole con le collation predefinite. LIMIT n (oppure LIMIT inizio, n) restituisce solo le prime righe del risultato ordinato.

Ricetta 1: conteggi per gruppo, compresi gli zeri

"Per ogni film, il numero di proiezioni". Con un JOIN normale i film mai proiettati sparirebbero; il LEFT JOIN li conserva, e COUNT di una colonna della tabella destra conta zero per loro (COUNT(*) conterebbe 1).

```
SELECT f.id_film, f.titolo, COUNT(p.id_proiezione) AS numero_proiezioni
FROM Film f
     LEFT JOIN Proiezione p ON p.film = f.id_film
GROUP BY f.id_film, f.titolo
ORDER BY numero_proiezioni DESC;
```

Ricetta 2: sopra la media

"I dipendenti pagati più della media". La media si calcola in una sottointerrogazione non correlata, valutata una volta sola. Se la media è di un conteggio (proiezioni per film) serve un livello in più: una tabella derivata nel FROM oppure una vista.

```
SELECT nome, paga_oraria
FROM Dipendente
WHERE paga_oraria > (SELECT AVG(paga_oraria) FROM Dipendente);

# film con piu proiezioni della media delle proiezioni per film
SELECT film, COUNT(*) AS n
FROM Proiezione
GROUP BY film
HAVING COUNT(*) > (SELECT AVG(conteggio)
                   FROM (SELECT COUNT(*) AS conteggio
                         FROM Proiezione GROUP BY film) AS t);   # alias obbligatorio
```

Ricetta 3: il massimo con tutti i pari merito

ORDER BY … LIMIT 1 restituisce una sola riga anche in caso di parità, quindi può perdere dei risultati. Le forme corrette confrontano con il massimo o usano ALL:

```
# la sala (o le sale) con piu posti
SELECT numero, posti FROM Sala
WHERE posti = (SELECT MAX(posti) FROM Sala);

# il film con piu proiezioni (anche a pari merito)
SELECT film, COUNT(*) AS n FROM Proiezione
GROUP BY film
HAVING COUNT(*) >= ALL (SELECT COUNT(*) FROM Proiezione GROUP BY film);
```

Ricetta 4: "tutti" (la divisione)

"I film proiettati in tutte le sale attrezzate per il 3D". Due soluzioni equivalenti: la doppia negazione, fedele alla logica, e il confronto dei conteggi, spesso più semplice da scrivere.

```
# non esiste una sala 3D in cui il film non sia stato proiettato
SELECT f.id_film, f.titolo FROM Film f
WHERE NOT EXISTS (SELECT * FROM Sala s
                  WHERE s.ha_3d = TRUE
                    AND NOT EXISTS (SELECT * FROM Proiezione p
                                    WHERE p.film = f.id_film AND p.sala = s.numero));

# conteggio: sale 3D distinte usate = numero di sale 3D
SELECT p.film
FROM Proiezione p JOIN Sala s ON p.sala = s.numero
WHERE s.ha_3d = TRUE
GROUP BY p.film
HAVING COUNT(DISTINCT p.sala) = (SELECT COUNT(*) FROM Sala WHERE ha_3d = TRUE);
```

Ricetta 5: "mai" e "solo"

```
# clienti che non hanno mai prenotato: NOT EXISTS ...
SELECT c.tessera, c.nome FROM Cliente c
WHERE NOT EXISTS (SELECT * FROM Prenotazione p WHERE p.cliente = c.tessera);

# ... oppure LEFT JOIN e controllo del NULL
SELECT c.tessera, c.nome
FROM Cliente c LEFT JOIN Prenotazione p ON p.cliente = c.tessera
WHERE p.cliente IS NULL;

# proiezionisti che hanno lavorato SOLO in sala 3
SELECT proiezionista FROM Proiezione
GROUP BY proiezionista
HAVING SUM(sala <> 3) = 0;           # nessuna proiezione in una sala diversa
```

L'ultimo esempio sfrutta il fatto che in MySQL un confronto vale 1 o 0: SUM(condizione) conta le righe che la soddisfano. È comodo anche per i conteggi condizionati, come SUM(pagata) per le prenotazioni pagate.

Ricetta 6: i primi N per gruppo

"Per ogni genere, i tre clienti con più prenotazioni". La soluzione classica, valida in ogni DBMS, usa una sottointerrogazione correlata che conta quanti hanno un valore maggiore; in MySQL 8 è più semplice con una **window function**.

```
WITH conteggi AS (                    # CTE: una vista temporanea per questa query
    SELECT f.genere, pr.cliente, COUNT(*) AS n
    FROM Prenotazione pr
         JOIN Proiezione p ON pr.proiezione = p.id_proiezione
         JOIN Film f ON p.film = f.id_film
    GROUP BY f.genere, pr.cliente
)
SELECT genere, cliente, n
FROM (SELECT genere, cliente, n,
             RANK() OVER (PARTITION BY genere ORDER BY n DESC) AS posizione
      FROM conteggi) AS t
WHERE posizione <= 3;
```

Window function

Una **window function** calcola, per ogni riga, un valore su un insieme di righe collegate (la finestra), senza raggruppare: a differenza di GROUP BY, ogni riga resta nel risultato. La finestra si definisce con **OVER**: OVER() è tutta la tabella, OVER(PARTITION BY x) le righe con lo stesso x, e ORDER BY dentro OVER stabilisce l'ordine. Si possono usare le aggregate (SUM, AVG, COUNT, MAX, MIN) e funzioni specifiche: ROW_NUMBER() (numero progressivo), RANK() (posizione con salti a parità), DENSE_RANK() (senza salti), FIRST_VALUE, LAST_VALUE, NTH_VALUE, LAG e LEAD (riga precedente e successiva), CUME_DIST. Le window function si calcolano dopo FROM, WHERE, GROUP BY e HAVING e possono comparire solo nella SELECT e nell'ORDER BY.

```
SELECT nome, mansione, paga_oraria,
       AVG(paga_oraria) OVER (PARTITION BY mansione) AS media_mansione,
       ROW_NUMBER() OVER (ORDER BY paga_oraria DESC) AS posizione
FROM Dipendente;

# totale progressivo delle prenotazioni per data
SELECT data_prenotazione, COUNT(*) AS giorno,
       SUM(COUNT(*)) OVER (ORDER BY data_prenotazione
                           ROWS UNBOUNDED PRECEDING) AS cumulato
FROM Prenotazione GROUP BY data_prenotazione;
```

Il **frame** restringe la finestra rispetto alla riga corrente: ROWS UNBOUNDED PRECEDING va dall'inizio della partizione alla riga corrente, ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING prende la riga precedente, la corrente e la successiva. La finestra si può anche definire una volta in una clausola WINDOW w AS (…) e richiamare con OVER w.

CTE e query ricorsive

La clausola **WITH** (common table expression) definisce tabelle temporanee con nome, valide solo per la query che segue: rende leggibili le query a più passi, come una vista usa e getta. Con WITH RECURSIVE si scrivono le viste ricorsive viste in teoria, per esempio una gerarchia di responsabili:

```
WITH RECURSIVE Superiori (impiegato, superiore) AS (
    SELECT matricola, capo FROM Impiegato WHERE capo IS NOT NULL      # passo base
    UNION ALL
    SELECT s.impiegato, i.capo                                        # passo ricorsivo
    FROM Superiori s JOIN Impiegato i ON s.superiore = i.matricola
    WHERE i.capo IS NOT NULL
)
SELECT * FROM Superiori ORDER BY impiegato;
```

Operatori insiemistici e join esterni in MySQL

UNION e UNION ALL sono sempre disponibili; INTERSECT ed EXCEPT esistono solo da MySQL 8.0.31, e nelle versioni precedenti si sostituiscono con IN e NOT IN (o EXISTS e NOT EXISTS). MySQL **non** supporta FULL OUTER JOIN: lo si ottiene come unione di un LEFT JOIN e di un RIGHT JOIN.

```
SELECT p.figlio, p.padre, m.madre
FROM Paternita p LEFT JOIN Maternita m ON p.figlio = m.figlio
UNION                                    # UNION elimina i duplicati
SELECT m.figlio, p.padre, m.madre
FROM Paternita p RIGHT JOIN Maternita m ON p.figlio = m.figlio;
```

CAPITOLO 5

Viste

Come si creano e si usano le viste in MySQL, quando sono aggiornabili e come servono a spezzare le query complesse e a proteggere i dati.

```
CREATE OR REPLACE VIEW Programmazione AS
SELECT p.id_proiezione, f.titolo, d.nome AS proiezionista, s.numero AS sala,
       p.data_ora, p.posti_venduti, s.posti
FROM Proiezione p
     JOIN Film f       ON p.film = f.id_film
     JOIN Dipendente d ON p.proiezionista = d.matricola
     JOIN Sala s       ON p.sala = s.numero;

SELECT * FROM Programmazione WHERE posti_venduti >= posti;   # proiezioni esaurite
DROP VIEW IF EXISTS Programmazione;
```

Una vista non memorizza dati: ogni volta che la si interroga MySQL esegue la query che la definisce, di solito fondendola con la query esterna (algoritmo MERGE) oppure calcolandola in una tabella temporanea (TEMPTABLE, necessario con aggregati e DISTINCT). Le viste servono a:

- **semplificare**: nascondere join ripetuti e dare un nome a un concetto, come nella vista sopra;
- **spezzare** le query a più livelli: la media dei conteggi diventa una media su una vista di conteggi;
- **proteggere**: si concede SELECT sulla vista e non sulla tabella, nascondendo colonne come la paga oraria;
- **rendere stabile** l'interfaccia verso le applicazioni quando lo schema cambia.

```
# la media dei conteggi con una vista
CREATE VIEW ProiezioniPerFilm (film, n) AS
    SELECT film, COUNT(*) FROM Proiezione GROUP BY film;

SELECT film, n FROM ProiezioniPerFilm
WHERE n > (SELECT AVG(n) FROM ProiezioniPerFilm);
```

Viste aggiornabili

Si può fare INSERT, UPDATE o DELETE su una vista quando ogni sua riga corrisponde a una sola riga di una tabella di base: niente aggregati, DISTINCT, GROUP BY, HAVING, UNION o sottointerrogazioni nella SELECT. Sulle viste con join si può aggiornare una sola delle tabelle per volta. L'opzione WITH CHECK OPTION rifiuta le modifiche che produrrebbero righe non più visibili attraverso la vista.

```
CREATE VIEW SalePrimoPiano AS
    SELECT numero, piano, posti FROM Sala WHERE piano = 1
    WITH CHECK OPTION;

INSERT INTO SalePrimoPiano VALUES (9, 1, 60);   # accettato
INSERT INTO SalePrimoPiano VALUES (10, 2, 60);  # rifiutato: piano diverso da 1
```

CAPITOLO 6

Trigger in MySQL

La sintassi dei trigger di MySQL, le loro regole e limitazioni, il comando DELIMITER e una serie di trigger completi: vincoli, ridondanze, registri di controllo e la traduzione delle asserzioni.

Sintassi

```
DELIMITER $$                      # il ; non chiude piu il comando: serve per il corpo
CREATE TRIGGER nome_trigger
{ BEFORE | AFTER } { INSERT | UPDATE | DELETE } ON tabella
FOR EACH ROW
[{ FOLLOWS | PRECEDES } altro_trigger]
BEGIN
    # istruzioni separate da ;
END $$
DELIMITER ;                       # si ripristina il delimitatore normale
```

Rispetto allo standard SQL:1999 visto in teoria, in MySQL:

- esistono solo trigger **row level** (FOR EACH ROW è obbligatorio): non ci sono trigger statement level né tabelle di transizione;
- non esiste la clausola WHEN: la condizione si scrive con IF … THEN … END IF dentro il corpo;
- le righe si leggono con **NEW.colonna** e **OLD.colonna**: OLD esiste in UPDATE e DELETE ed è in sola lettura; NEW esiste in INSERT e UPDATE e solo in un trigger BEFORE si può modificare con SET NEW.colonna = …;
- un trigger **non può modificare la tabella su cui è definito** (errore 1442): per correggere la riga in arrivo si usa un BEFORE con SET NEW, non un UPDATE sulla stessa tabella;
- per lo stesso evento e momento si possono definire più trigger (da MySQL 5.7), ordinati con FOLLOWS e PRECEDES; senza indicazioni vale l'ordine di creazione;
- per bloccare un'operazione si solleva un errore con **SIGNAL SQLSTATE '45000'**, che annulla anche la modifica che ha attivato il trigger.

Il comando **DELIMITER** serve perché il corpo del trigger contiene dei punto e virgola: senza cambiare delimitatore il client manderebbe al server il comando a metà, al primo ;. È necessario sia nel client mysql sia quando si esegue uno script in Workbench; non serve se si crea il trigger con la finestra guidata di Workbench, che lo gestisce da sola. Un trigger si elimina con DROP TRIGGER [IF EXISTS] nome e si elencano con SHOW TRIGGERS.

BEFORE o AFTER?

OBIETTIVO	MOMENTO	COME
Rifiutare una riga che viola una regola	BEFORE	SELECT … INTO variabili, IF regola violata THEN SIGNAL
Correggere o completare la riga in arrivo	BEFORE	SET NEW.colonna = valore
Aggiornare un dato ridondante in un'altra tabella	AFTER	UPDATE sull'altra tabella usando NEW o OLD
Registrare la modifica in un log	AFTER	INSERT nella tabella di log con OLD, NEW, utente e ora

Trigger 1: vincolo con sottointerrogazione

Un CHECK non può contenere query, quindi "non si possono prenotare più posti di quelli della sala" richiede un trigger. Si leggono il numero di prenotazioni e i posti della sala, e si blocca l'inserimento se la proiezione è esaurita.

```
DELIMITER $$
CREATE TRIGGER NoOverbooking
BEFORE INSERT ON Prenotazione
FOR EACH ROW
BEGIN
    DECLARE prenotati INT;
    DECLARE capienza INT;
    # prenotazioni gia presenti per la proiezione
    SELECT COUNT(*) INTO prenotati FROM Prenotazione WHERE proiezione = NEW.proiezione;
    # posti della sala in cui si svolge la proiezione
    SELECT s.posti INTO capienza
    FROM Proiezione p JOIN Sala s ON p.sala = s.numero
    WHERE p.id_proiezione = NEW.proiezione;
    IF prenotati >= capienza THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Proiezione esaurita';
    END IF;
END $$
DELIMITER ;
```

Trigger 2: regola sulle date

CURDATE() non è ammessa nei CHECK, ma nei trigger sì. Si rifiuta la prenotazione se il cliente, alla data della proiezione, non ha l'età minima richiesta dal film; intanto si completa la riga con la data odierna.

```
DELIMITER $$
CREATE TRIGGER EtaMinima
BEFORE INSERT ON Prenotazione
FOR EACH ROW
BEGIN
    DECLARE nascita DATE;
    DECLARE quando DATETIME;
    DECLARE minima INT;
    SELECT data_nascita INTO nascita FROM Cliente WHERE tessera = NEW.cliente;
    SELECT p.data_ora, f.eta_minima INTO quando, minima
    FROM Proiezione p JOIN Film f ON p.film = f.id_film
    WHERE p.id_proiezione = NEW.proiezione;
    # eta compiuta alla data della proiezione
    IF TIMESTAMPDIFF(YEAR, nascita, DATE(quando)) < minima THEN
        SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'Eta minima non raggiunta per questo film';
    END IF;
    SET NEW.data_prenotazione = CURDATE();   # completa la riga in arrivo
END $$
DELIMITER ;
```

Trigger 3: mantenere un attributo ridondante

Se l'analisi delle ridondanze ha deciso di tenere posti_venduti in Proiezione, il valore va aggiornato a ogni inserimento, cancellazione e spostamento di prenotazione; dimenticare uno dei tre casi rende il dato incoerente.

```
CREATE TRIGGER PrenotazioneInserita AFTER INSERT ON Prenotazione FOR EACH ROW
    UPDATE Proiezione SET posti_venduti = posti_venduti + 1
    WHERE id_proiezione = NEW.proiezione;

CREATE TRIGGER PrenotazioneCancellata AFTER DELETE ON Prenotazione FOR EACH ROW
    UPDATE Proiezione SET posti_venduti = posti_venduti - 1
    WHERE id_proiezione = OLD.proiezione;

DELIMITER $$
CREATE TRIGGER PrenotazioneSpostata AFTER UPDATE ON Prenotazione FOR EACH ROW
BEGIN
    IF NEW.proiezione <> OLD.proiezione THEN      # solo se cambia la proiezione
        UPDATE Proiezione SET posti_venduti = posti_venduti - 1
        WHERE id_proiezione = OLD.proiezione;
        UPDATE Proiezione SET posti_venduti = posti_venduti + 1
        WHERE id_proiezione = NEW.proiezione;
    END IF;
END $$
DELIMITER ;
```

I primi due trigger hanno una sola istruzione, quindi non serve BEGIN … END né DELIMITER. Notare che il trigger è su Prenotazione e modifica Proiezione: è permesso, perché non tocca la tabella che lo ha attivato.

Trigger 4: registro delle modifiche (audit)

Molte applicazioni richiedono la tracciabilità delle modifiche. Si crea una tabella di log e un trigger AFTER che vi scrive i valori vecchi e nuovi, l'utente e l'istante.

```
CREATE TABLE LogPaghe (
    id         INT AUTO_INCREMENT PRIMARY KEY,
    dipendente INT NOT NULL,
    vecchia    DECIMAL(6,2),
    nuova      DECIMAL(6,2),
    utente     VARCHAR(100),
    istante    DATETIME DEFAULT CURRENT_TIMESTAMP
);

DELIMITER $$
CREATE TRIGGER TracciaPaghe AFTER UPDATE ON Dipendente FOR EACH ROW
BEGIN
    IF NEW.paga_oraria <> OLD.paga_oraria THEN
        INSERT INTO LogPaghe (dipendente, vecchia, nuova, utente)
        VALUES (NEW.matricola, OLD.paga_oraria, NEW.paga_oraria, CURRENT_USER());
    END IF;
END $$
DELIMITER ;
```

Trigger 5: tradurre un'asserzione

MySQL non ha CREATE ASSERTION. Un'asserzione si traduce individuando le operazioni che possono violarla e scrivendo per ciascuna un trigger BEFORE che rifà il controllo. Per "una paga non può superare il doppio della media" le operazioni a rischio sono l'inserimento di un dipendente e la modifica di una paga:

```
DELIMITER $$
CREATE TRIGGER LimitePagaIns BEFORE INSERT ON Dipendente FOR EACH ROW
BEGIN
    DECLARE media DECIMAL(8,2);
    SELECT AVG(paga_oraria) INTO media FROM Dipendente;
    IF media IS NOT NULL AND NEW.paga_oraria > 2 * media THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Paga oltre il doppio della media';
    END IF;
END $$
DELIMITER ;
# un trigger gemello BEFORE UPDATE ON Dipendente fa lo stesso controllo sulle modifiche
```

ERRORI TIPICI CON I TRIGGER

- dimenticare DELIMITER negli script: il server riceve un CREATE TRIGGER troncato e segnala un errore di sintassi;
- fare UPDATE sulla stessa tabella del trigger (errore 1442): usare BEFORE e SET NEW;
- una SELECT … INTO che restituisce più righe (errore 1172): la query deve dare un solo valore;
- dimenticare i casi UPDATE e DELETE quando si mantiene una ridondanza;
- confondere NEW e OLD: in un DELETE esiste solo OLD, in un INSERT solo NEW.

CAPITOLO 7

Stored procedure, funzioni ed eventi

Il linguaggio procedurale di MySQL: parametri, variabili, controllo del flusso, cursori, gestione degli errori, funzioni richiamabili nelle query ed eventi pianificati.

Una **stored procedure** è un programma memorizzato nel database ed eseguito dal server su richiesta, con CALL. SQL è dichiarativo; le procedure aggiungono costrutti procedurali: variabili, IF, cicli, cursori. Ogni DBMS ha il proprio linguaggio (PL/SQL in Oracle, PL/pgSQL in PostgreSQL, Transact SQL in SQL Server); MySQL usa una sintassi vicina allo standard SQL/PSM e ammette solo procedure scritte in SQL.

VANTAGGI	SVANTAGGI
Riusabili da qualsiasi applicazione, perché girano sul server	Linguaggi diversi da un DBMS all'altro
Astrazione: chi le chiama ignora i dettagli	Il carico applicativo si sposta sul server
Sicurezza: si concede EXECUTE sulla procedura e non l'accesso diretto alle tabelle	Debug meno comodo
Prestazioni: un solo scambio con il server invece di molte query	

Procedure e parametri

```
DELIMITER $$
CREATE PROCEDURE ProiezioniDiFilm (IN p_film INT, OUT p_totale INT)
BEGIN
    # elenco delle proiezioni: diventa un result set per il chiamante
    SELECT id_proiezione, data_ora, sala FROM Proiezione
    WHERE film = p_film ORDER BY data_ora;
    # conteggio restituito nel parametro di uscita
    SELECT COUNT(*) INTO p_totale FROM Proiezione WHERE film = p_film;
END $$
DELIMITER ;

CALL ProiezioniDiFilm(3, @tot);       # @tot riceve il valore di OUT
SELECT @tot;
```

- **IN** (predefinito): il valore entra nella procedura; le modifiche interne non si vedono fuori;
- **OUT**: la procedura assegna il valore, che il chiamante legge in una variabile di sessione;
- **INOUT**: entra un valore e ne esce uno modificato.

Variabili

Le variabili **locali** si dichiarano all'inizio di un blocco BEGIN … END con DECLARE nome tipo [DEFAULT valore] (senza DEFAULT valgono NULL) e sono visibili solo nel blocco. Le variabili di **sessione** iniziano con @, non si dichiarano e durano fino alla chiusura della connessione. Si assegnano con SET oppure con SELECT … INTO.

```
DECLARE n INT DEFAULT 0;
SET n = n + 1;
SELECT COUNT(*) INTO n FROM Cliente;
SET @soglia = 25;                       # variabile di sessione, anche fuori dalle procedure
```

Controllo del flusso

```
IF condizione THEN istruzioni;
ELSEIF altra_condizione THEN istruzioni;
ELSE istruzioni;
END IF;

CASE WHEN x < 10 THEN SET esito = 'basso';
     WHEN x < 20 THEN SET esito = 'medio';
     ELSE SET esito = 'alto';
END CASE;

WHILE i <= 5 DO SET i = i + 1; END WHILE;              # controlla prima
REPEAT SET i = i + 1; UNTIL i > 5 END REPEAT;         # esegue almeno una volta
ciclo: LOOP
    SET i = i + 1;
    IF i > 10 THEN LEAVE ciclo; END IF;               # LEAVE = break
    IF MOD(i, 2) = 1 THEN ITERATE ciclo; END IF;      # ITERATE = continue
END LOOP ciclo;
```

Cursori e handler

Un **cursore** scorre una per una le righe di una query. In MySQL i cursori sono in sola lettura, non scorrevoli (solo in avanti, senza salti) e asensitive (non bisogna modificare le tabelle su cui sono aperti). Si dichiarano dopo le variabili, si aprono con OPEN, si leggono con FETCH … INTO e si chiudono con CLOSE. Quando le righe finiscono, FETCH solleva la condizione NOT FOUND: per non interrompere la procedura si dichiara un **handler** che imposta una variabile di fine.

```
DELIMITER $$
CREATE PROCEDURE PremioProiezionisti ()
BEGIN
    DECLARE fine INT DEFAULT 0;
    DECLARE cod INT;
    DECLARE n INT;
    DECLARE cur CURSOR FOR
        SELECT proiezionista, COUNT(*) FROM Proiezione GROUP BY proiezionista;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET fine = 1;   # righe finite

    OPEN cur;
    leggi: LOOP
        FETCH cur INTO cod, n;
        IF fine = 1 THEN LEAVE leggi; END IF;
        IF n > 100 THEN                                    # piu di 100 proiezioni
            UPDATE Dipendente SET paga_oraria = paga_oraria * 1.03
            WHERE matricola = cod;
        END IF;
    END LOOP leggi;
    CLOSE cur;
END $$
DELIMITER ;
```

L'ordine delle dichiarazioni è obbligatorio: prima le variabili, poi i cursori, poi gli handler. Un handler **CONTINUE** fa proseguire l'esecuzione, uno **EXIT** esce dal blocco. Lo stesso aumento si poteva fare con un solo UPDATE con sottointerrogazione: i cursori servono quando la logica riga per riga non si esprime in SQL.

Procedure transazionali e gestione degli errori

Le operazioni complesse del progetto (per esempio spostare la prenotazione di un cliente su un'altra proiezione) vanno eseguite come una transazione che si annulla in caso di errore. Un handler EXIT su SQLEXCEPTION fa il rollback e rilancia l'errore con RESIGNAL, così il chiamante sa che cosa è successo.

```
DELIMITER $$
CREATE PROCEDURE CambiaProiezione (IN p_cliente INT, IN p_da INT, IN p_a INT)
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN
        ROLLBACK;          # annulla tutto cio che la transazione ha fatto
        RESIGNAL;          # ripropone l'errore originale al chiamante
    END;
    START TRANSACTION;
        # le due operazioni devono riuscire insieme o fallire insieme
        DELETE FROM Prenotazione WHERE cliente = p_cliente AND proiezione = p_da;
        INSERT INTO Prenotazione (cliente, proiezione, data_prenotazione)
        VALUES (p_cliente, p_a, CURDATE());    # i trigger controllano posti ed eta
    COMMIT;
    SELECT 'Cambio registrato' AS esito;
END $$
DELIMITER ;
```

Se la nuova proiezione è esaurita, il trigger NoOverbooking solleva un errore durante l'INSERT: l'handler annulla anche la DELETE già eseguita, e il cliente conserva la prenotazione originale.

Funzioni

Una **stored function** restituisce un valore con RETURN e si usa dentro le query, come le funzioni predefinite. Accetta solo parametri di ingresso e non può restituire result set. Se il server ha il log binario attivo, come di default in MySQL 8, bisogna dichiarare la natura della funzione: DETERMINISTIC (stesso risultato a parità di argomenti), READS SQL DATA (legge tabelle) o NO SQL; altrimenti la creazione fallisce con l'errore 1418.

```
DELIMITER $$
CREATE FUNCTION PostiLiberi (p_proiezione INT) RETURNS INT
READS SQL DATA
BEGIN
    DECLARE liberi INT;
    SELECT s.posti - p.posti_venduti INTO liberi
    FROM Proiezione p JOIN Sala s ON p.sala = s.numero
    WHERE p.id_proiezione = p_proiezione;
    RETURN liberi;
END $$
DELIMITER ;

SELECT id_proiezione, data_ora, PostiLiberi(id_proiezione) AS liberi FROM Proiezione;
```

Procedure e funzioni si elencano con SHOW PROCEDURE STATUS e SHOW FUNCTION STATUS, se ne vede il codice con SHOW CREATE PROCEDURE nome e si eliminano con DROP PROCEDURE / DROP FUNCTION [IF EXISTS] nome. Per farle eseguire a un altro utente serve il privilegio EXECUTE.

Eventi pianificati

Un **evento** è un blocco di SQL eseguito dal server a un orario o a intervalli regolari, come un cron interno. Richiede che lo scheduler sia attivo (SET GLOBAL event_scheduler = ON, attivo di default in MySQL 8).

```
CREATE EVENT PuliziaPrenotazioni
ON SCHEDULE EVERY 1 DAY STARTS '20261006030000'     # ogni notte alle 3
DO
    DELETE FROM Prenotazione
    WHERE pagata = FALSE
      AND data_prenotazione < DATE_SUB(CURDATE(), INTERVAL 30 DAY);
```

CAPITOLO 8

Indici e prestazioni

Come si creano gli indici in MySQL, come si legge il piano di esecuzione con EXPLAIN e quali condizioni impediscono di usare un indice.

Creare gli indici

InnoDB crea automaticamente un indice per la chiave primaria (la tabella stessa è organizzata come albero B+ su di essa), per ogni vincolo UNIQUE e per ogni chiave esterna. Gli altri indici si aggiungono per le colonne usate spesso in WHERE, JOIN e ORDER BY.

```
CREATE INDEX idx_proiezione_data ON Proiezione(data_ora);
CREATE UNIQUE INDEX idx_cliente_email ON Cliente(email);
CREATE INDEX idx_proiezione_sala_data ON Proiezione(sala, data_ora);   # indice composto
SHOW INDEX FROM Proiezione;
DROP INDEX idx_proiezione_data ON Proiezione;
```

Un **indice composto** su (sala, data_ora) è ordinato prima per sala e, a parità di sala, per data: serve alle ricerche su sala, su sala e data, e su sala con un intervallo di date, ma non a quelle sulla sola data. È la regola del **prefisso sinistro**, la stessa dell'elenco telefonico ordinato per cognome e nome, inutile per cercare solo per nome. Ogni indice accelera le letture ma rallenta INSERT, UPDATE e DELETE e occupa spazio: si indicizza ciò che le operazioni frequenti usano davvero.

EXPLAIN

Anteponendo EXPLAIN a una SELECT, MySQL mostra il piano scelto dall'ottimizzatore invece di eseguirla. In Workbench lo stesso piano si vede in forma grafica con il pulsante Execution Plan. EXPLAIN ANALYZE (da MySQL 8.0.18) esegue davvero la query e riporta anche i tempi reali.

```
EXPLAIN SELECT * FROM Proiezione WHERE sala = 2 AND data_ora >= '20261001';
```

COLONNA	SIGNIFICATO
type	tipo di accesso, dal peggiore al migliore: ALL (scansione completa), index (scansione di tutto l'indice), range (intervallo su un indice), ref (uguaglianza su un indice non unico), eq_ref (una riga per join su chiave), const (al più una riga)
possible_keys	indici che si potrebbero usare
key	indice effettivamente scelto; NULL se nessuno
rows	stima del numero di righe da esaminare: è il numero da ridurre
Extra	note: Using where (filtro dopo la lettura), Using index (basta l'indice, senza leggere la tabella), Using filesort e Using temporary (ordinamento o tabella temporanea, spesso costosi)

Condizioni che impediscono l'uso di un indice

Un indice su una colonna si usa solo se la condizione confronta la **colonna da sola** con un valore. Se la colonna è dentro una funzione o un'espressione, l'indice non serve e si torna alla scansione completa:

```
# NON usa l'indice su data_ora: la colonna e dentro una funzione
SELECT * FROM Proiezione WHERE YEAR(data_ora) = 2026;
# usa l'indice: stessa condizione riscritta come intervallo
SELECT * FROM Proiezione WHERE data_ora >= '20260101' AND data_ora < '20270101';

# NON usa l'indice: LIKE che inizia con %
SELECT * FROM Cliente WHERE nome LIKE '%rossi';
# usa l'indice: prefisso fisso
SELECT * FROM Cliente WHERE nome LIKE 'Rossi%';
```

Altri casi frequenti: confronti fra tipi diversi (una colonna VARCHAR confrontata con un numero), condizioni in OR su colonne diverse senza un indice per ciascuna, e indici compositi interrogati senza la prima colonna.

Esempio: ottimizzare una query lenta

L'operazione "proiezioni della sala 2 in un dato giorno" viene eseguita spesso. Si guarda il piano prima e dopo la creazione di un indice composto; i valori di rows sono le stime per una tabella di 10.000 proiezioni.

```
EXPLAIN SELECT id_proiezione, data_ora FROM Proiezione
WHERE sala = 2 AND data_ora >= '20261005' AND data_ora < '20261006';

CREATE INDEX idx_proiezione_sala_data ON Proiezione(sala, data_ora);  # poi si ripete EXPLAIN
```

MOMENTO	TYPE	KEY	ROWS	EXTRA
Prima	ref	chiave esterna su sala	circa 2.000	Using where: legge tutte le proiezioni della sala e filtra la data
Dopo	range	idx_proiezione_sala_data	circa 8	Using where; Using index: intervallo sull'indice composto

Prima dell'indice MySQL usa l'indice creato automaticamente per la chiave esterna sala, ma deve esaminare tutte le proiezioni di quella sala; con l'indice composto scende direttamente all'intervallo di date della sala 2. Poiché la query chiede solo colonne contenute nell'indice (la chiave primaria è inclusa in ogni indice secondario di InnoDB), non serve nemmeno leggere la tabella: è un **indice coprente**. Questi confronti, con le stime di rows, sono un ottimo materiale per la sezione di progettazione fisica del progetto.

CAPITOLO 9

Importare, esportare e fare il backup

Come eseguire script SQL, caricare dati da file di testo e CSV, esportare un database con mysqldump e con Workbench: tutto ciò che serve per consegnare il dump del progetto.

Eseguire uno script SQL

```
mysql -u root -p cinema < creazione.sql          # dal terminale, nel database cinema
mysql -u root -p < creazione.sql                 # se lo script contiene gia USE cinema
mysql> SOURCE /percorso/creazione.sql;          # dalla shell di mysql
```

In Workbench si apre lo script con File, Open SQL Script e lo si esegue con il fulmine. Se lo script contiene trigger o procedure, ogni definizione deve essere racchiusa fra DELIMITER $$ e DELIMITER ;.

Caricare dati da file: LOAD DATA

```
LOAD DATA LOCAL INFILE '/percorso/clienti.csv'
INTO TABLE Cliente
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 LINES                                   # salta la riga di intestazione del CSV
(nome, data_nascita, categoria);                 # colonne nell'ordine del file
```

LOAD DATA carica righe da un file di testo molto più velocemente di tanti INSERT. Senza **LOCAL** il file viene letto dal server, nel percorso indicato e solo nella cartella permessa dalla variabile secure_file_priv; con LOCAL il file viene letto dal client e inviato al server, ma l'opzione local_infile deve essere attiva sia sul server (SET GLOBAL local_infile = 1) sia sul client. REPLACE o IGNORE stabiliscono che cosa fare con le righe che duplicano una chiave; LINES STARTING BY permette di saltare un prefisso su ogni riga. Dal terminale, il programma mysqlimport fa la stessa cosa usando il nome del file come nome della tabella.

Backup con mysqldump

```
mysqldump -u root -p cinema > cinema_dump.sql                # un database
mysqldump -u root -p --routines cinema > cinema_dump.sql     # con procedure e funzioni
mysqldump -u root -p --databases cinema altro_db > due.sql     # piu database
mysqldump -u root -p --all-databases > tutto.sql                 # tutti i database
mysql -u root -p cinema < cinema_dump.sql                    # ripristino
```

Il **dump** è un file SQL con i CREATE TABLE e gli INSERT necessari a ricostruire il database: è ciò che il progetto chiede di consegnare ("dump del database popolato"). I trigger sono inclusi di default, le procedure e le funzioni solo con l'opzione routines, gli eventi con events. In Workbench si ottiene lo stesso risultato da Server, Data Export, scegliendo lo schema e spuntando "Include Create Schema" e le voci per stored procedure, funzioni, eventi e trigger; Data Import ripristina un dump.

CAPITOLO 10

Il progetto di una base di dati

Che cosa contiene un progetto completo, come impostare la relazione passo per passo, come calcolare i costi con e senza indici e una lista di controllo prima della consegna.

Che cosa produrre

Un progetto di basi di dati mette insieme tutto ciò che si è studiato: parte da un dominio descritto a parole (un cinema, una rete di officine, un archivio museale) e arriva a un database MySQL funzionante, accompagnato da una **relazione** che documenta e motiva ogni scelta. Il risultato si compone di due parti.

NELLA RELAZIONE	DA PRODURRE COME FILE
Progettazione concettuale: schema ER con entità, associazioni, attributi e vincoli	Schema delle tabelle in SQL (DDL completo)
Progettazione logica: traduzione nello schema relazionale	Dump del database popolato con dati significativi
Progettazione fisica: scelta motivata degli indici, costo delle operazioni con e senza indici	Query SQL delle operazioni principali
Normalizzazione: verifica che le relazioni siano in 3NF o BCNF	Trigger per i vincoli di integrità e stored procedure per le operazioni complesse

La struttura della relazione

La relazione segue l'ordine della metodologia vista in teoria; ogni capitolo produce il materiale che serve al successivo. Un indice consigliato, con che cosa deve contenere ogni parte:

1. **Descrizione del progetto**: il dominio in poche righe e gli obiettivi del sistema.
2. **Specifiche sui dati**: i requisiti riscritti in frasi standardizzate, raggruppate per concetto (dati sui manufatti, dati sui committenti, …), senza ambiguità.
3. **Glossario dei termini**: tabella con termine, descrizione, sinonimi e collegamenti.
4. **Definizione delle operazioni**: un elenco numerato di 8 o 10 operazioni significative (inserimenti, ricerche, statistiche), che torneranno in tutte le fasi successive.
5. **Progettazione concettuale**: strategia scelta (di solito top down o mista) e perché; schema scheletro; raffinamenti successivi, ciascuno con il suo frammento di schema; schema ER finale.
6. **Vincoli non esprimibili**: domini ristretti (valori ammessi di uno stato), vincoli fra attributi (data di fine non precedente a quella di inizio), regole aziendali (un intervento richiede un'autorizzazione approvata), con l'indicazione di come verranno realizzati: CHECK, ENUM o trigger.
7. **Dizionario dei dati**: tabella delle entità (descrizione, attributi, identificatore) e delle associazioni (descrizione, entità coinvolte con cardinalità, attributi).
8. **Tavola dei volumi e tavola delle frequenze**: numeri plausibili e coerenti con le cardinalità; per ogni operazione tipo (interattiva o batch) e frequenza.
9. **Schemi delle operazioni e tavole degli accessi**: per ogni operazione lo schema di navigazione e il conteggio degli accessi in lettura e scrittura.
10. **Ristrutturazione**: analisi delle ridondanze con i calcoli, eliminazione delle generalizzazioni, partizionamenti o accorpamenti, scelta degli identificatori primari.

11. **Traduzione nel modello relazionale**: come è stata tradotta ogni associazione e lo schema logico finale con chiavi primarie e chiavi esterne evidenziate.
12. **Normalizzazione**: per ogni relazione le dipendenze funzionali e la verifica della BCNF (o della 3NF).
13. **Progettazione fisica**: indici scelti con la motivazione e confronto dei costi delle operazioni con e senza indici.
14. **Implementazione SQL**: DDL, trigger, stored procedure, viste e le query delle operazioni, ciascuna con una breve spiegazione e, per i trigger, un esempio di inserimento rifiutato.

CONSIGLI PRATICI

- Scegli le operazioni presto e tienile fisse: guidano volumi, accessi, indici e query; cambiare un'operazione a fine lavoro costringe a rifare metà relazione.
- Le cardinalità dello schema ER, i NOT NULL e le chiavi esterne del DDL devono raccontare la stessa storia: un (1,1) nello schema diventa una chiave esterna NOT NULL nella tabella.
- Ogni vincolo non esprimibile deve comparire nel codice come CHECK, ENUM o trigger: è il collegamento fra progetto e implementazione che va sempre verificato.
- Per i dati di prova servono abbastanza righe da rendere sensate le query (almeno qualche decina per le tabelle principali) e casi limite che mostrino i trigger in azione.
- Prova lo script di creazione da zero su un database vuoto prima di consegnarlo, con DROP DATABASE e riesecuzione: è il modo più sicuro per scoprire DELIMITER mancanti, ordini di creazione sbagliati e chiavi esterne non soddisfatte.

Il calcolo dei costi con e senza indici

La progettazione fisica chiede di giustificare gli indici confrontando il costo delle operazioni, misurato come in teoria in **accessi a blocco**. Si fissano prima i parametri (dimensione del blocco, dimensione media delle righe, quindi righe per blocco) e si ricava dalla tavola dei volumi il numero di blocchi di ogni tabella. Poi, per ogni operazione:

- **senza indice**, una ricerca per uguaglianza su una colonna non chiave legge tutta la tabella: b blocchi; una ricerca sulla chiave primaria in InnoDB usa comunque l'albero B+ della chiave;
- **con indice**, il costo è l'altezza dell'albero (di solito 2 o 3 accessi) più un accesso per ogni riga trovata, perché righe diverse stanno in generale in blocchi diversi;
- per le **scritture** l'indice costa: ogni inserimento aggiunge l'aggiornamento di ciascun indice della tabella, circa altezza + 1 accessi per indice;
- moltiplicando per le frequenze si ottiene il costo giornaliero, e si tiene l'indice se il risparmio sulle letture supera il costo aggiuntivo sulle scritture.

ESEMPIO SVOLTO

Proiezione ha 10.000 righe di circa 200 byte; blocchi da 4 KB, quindi 20 righe per blocco e 500 blocchi. Operazione A: "proiezioni di una sala in un giorno", 50 volte al giorno, restituisce in media 8 righe. Operazione B: inserimento di una proiezione, 20 volte al giorno. Indice candidato: (sala, data_ora), albero di altezza 2.

- Senza indice: A costa 500 accessi, per 50 = 25.000; B costa circa 2 accessi (lettura e scrittura del blocco), per 20 = 40. Totale 25.040.
- Con indice: A costa 2 + 8 = 10 accessi, per 50 = 500; B costa 2 + 3 (aggiornamento dell'indice) = 5, per 20 = 100. Totale 600.

L'indice riduce il costo giornaliero di oltre 40 volte: si mantiene. Un indice su una colonna poco selettiva (uno stato con tre valori) o su una tabella con molte scritture e poche letture può invece non convenire, e va scritto con lo stesso ragionamento.

Lista di controllo prima della consegna

CONTROLLO	FATTO QUANDO
Schema ER	ogni entità ha un identificatore, ogni associazione le cardinalità su entrambi i rami
Coerenza	volumi, cardinalità e frequenze sono plausibili fra loro; le operazioni della relazione coincidono con le query del file SQL
Ridondanze	ogni attributo derivato ha il suo calcolo di convenienza e, se mantenuto, i trigger che lo aggiornano
Schema logico	chiavi primarie ed esterne indicate; le associazioni molti a molti hanno la loro tabella
Normalizzazione	ogni relazione ha le sue dipendenze e la verifica della forma normale
DDL	ENGINE InnoDB, chiavi esterne con politiche ON DELETE scelte consapevolmente, CHECK per i domini
Trigger e procedure	racchiusi fra DELIMITER, provati con un caso che li attiva
Dump	generato con mysqldump o Data Export includendo trigger e routine; reimportato con successo in un database vuoto
Relazione	indice, numerazione delle pagine, figure leggibili, autore e data in copertina

La presentazione

Quando si presenta un progetto bisogna saper motivare le scelte: perché un'entità e non un attributo, perché quella cardinalità, perché mantenere o eliminare una ridondanza, perché quell'indice, che cosa succede se si cancella una riga riferita. Conviene preparare una breve dimostrazione dal vivo: una query significativa, un trigger che rifiuta un inserimento con il suo messaggio, una procedura che esegue un'operazione complessa. Se per il dominio o per la sintassi sono stati usati strumenti di supporto, va dichiarato nella relazione, essendo pronti a spiegare ogni componente in modo autonomo.