Riassunto Basi Laboratorio

Creazione di tabelle e domini in PostgreSQL

Creazione di una tabella: `CREATE TABLE`

La sintassi base è:

```
CREATE TABLE nome_tabella (
  nome_attributo dominio [DEFAULT valore] [vincoli],
  ...
  [vincolo_di_tabella]
);
```

Esempio:

```
CREATE TABLE Impiegato (
  Matricola CHAR(6) PRIMARY KEY,
  Nome VARCHAR(20) NOT NULL,
  Cognome VARCHAR(20) NOT NULL,
  Dipart VARCHAR(15),
  Stipendio NUMERIC(9) DEFAULT 0,
  FOREIGN KEY(Dipart) REFERENCES Dipartimento(NomeDip),
  UNIQUE (Cognome, Nome)
);
```

Domini (Tipi di Dato)

***Domini elementari predefiniti*:**

- **Testuali**:
  - `CHAR(n)`: stringa di lunghezza fissa (padded con spazi).
  - `VARCHAR(n)`: stringa di lunghezza variabile.
  - `TEXT`: stringa di lunghezza illimitata (estensione PostgreSQL).
- **Booleani**:
  - `BOOLEAN`: valori `TRUE`, `FALSE`, `NULL`.
- **Numerici esatti**:
  - `SMALLINT` (2 byte), `INTEGER` (4 byte)
  - `NUMERIC(p,s)` o `DECIMAL(p,s)`: numeri decimali precisi con *p* cifre totali, *s* decimali.
- **Numerici approssimati**:
  - `REAL`, `DOUBLE PRECISION`: virgola mobile, ma non precisi (non usare per soldi!).
- **Temporali**:

- `DATE`, `TIME`, `TIMESTAMP [WITH TIME ZONE]`
- `INTERVAL`: intervallo temporale (`INTERVAL YEAR TO MONTH`, `DAY TO SECOND`)

Domini definiti dall’utente

Per riusabilità, si possono creare nuovi domini a partire da tipi predefiniti:

```
CREATE DOMAIN Voto AS SMALLINT
  DEFAULT NULL
  CHECK (VALUE >= 18 AND VALUE <= 30);
```

Vincoli intrarelazionali

- `NOT NULL`: il valore deve essere presente.
- `DEFAULT`: assegna un valore di default se non specificato.
- `UNIQUE`: garantisce l’unicità (può essere su uno o più attributi).
- `PRIMARY KEY`: unica chiave primaria per tabella (implica `NOT NULL`).
- `CHECK`: vincoli generici (es. `stipendio >= 0`).

Vincoli interrelazionali (Foreign Key)

Usati per imporre relazioni tra tabelle:

Su singolo attributo:

```
Dipart VARCHAR(15) REFERENCES Dipartimento(NomeDip)
```

Su più attributi:

```
FOREIGN KEY (Nome, Cognome) REFERENCES Anagrafica(Nome, Cognome)
```

Modifiche a posteriori (ALTER)

Modificare la struttura dopo la creazione:

- Aggiungere colonna:

  ```
  ALTER TABLE Impiegato ADD COLUMN Stipendio NUMERIC(8,2);
  ```

- Eliminare colonna:

  ```
  ALTER TABLE Impiegato DROP COLUMN Stipendio;
  ```

- Cambiare valore default:

```
ALTER TABLE Impiegato ALTER COLUMN Stipendio SET DEFAULT 1000.00;
```

Eliminazione dati

- Cancellare tutte le righe:

  ```
  DELETE FROM Impiegato;
  ```

- Cancellare una tabella:

  ```
  DROP TABLE Impiegato;
  ```

Interrogazioni SQL: SELECT

Obiettivo del comando `SELECT`

Recuperare dati da una o più tabelle, con possibilità di:

- filtrare (`WHERE`)
- ordinare (`ORDER BY`)
- raggruppare (`GROUP BY`)
- aggregare (es. `AVG`, `COUNT`)
- unire (`JOIN`)

Sintassi generale semplificata

```
SELECT [DISTINCT]
       [ * | espressione [AS alias], ... ]
FROM   tabella [, ...]
WHERE  condizione
GROUP BY attributi
HAVING condizione_raggruppamento
ORDER BY attributi [ASC|DESC];
```

Clausole principali

`SELECT`

- `*`: tutti gli attributi
- Espressioni con alias:

  ```
  SELECT UPPER(cognome) || ' ' || nome AS NomeCompleto
  ```

`FROM`

- Lista di tabelle (anche con alias):

```
FROM Impiegato AS I, Reparto AS R
```

`WHERE`

- Filtra le tuple con espressioni booleane:

  ```
  WHERE città = 'Verona' AND media > 25
  ```

- Operatori: `=`, `<`, `>`, `<=`, `>=`, `<>`, `AND`, `OR`, `NOT`

Operatori speciali

Operatore	Funzione	Esempio
`LIKE`	Confronto con pattern (%, _)	`WHERE nome LIKE 'A%'`
`SIMILAR TO`	Pattern tipo regex SQL	`SIMILAR TO '[ABDN]%a'`
`BETWEEN`	Intervallo inclusivo	`WHERE età BETWEEN 18 AND 25`
`IN`	Appartenenza a un insieme	`WHERE città IN ('Verona', 'Padova')`
`IS NULL`	Valore nullo	`WHERE email IS NULL`

Operatori di aggregazione

Funzione	Descrizione
`COUNT(*)`	Conta tutte le righe
`COUNT(attr)`	Conta righe con valore non nullo
`AVG(attr)`	Media dei valori
`SUM(attr)`	Somma dei valori
`MIN(attr)`	Valore minimo
`MAX(attr)`	Valore massimo

Clausola `GROUP BY` e `HAVING`

- `GROUP BY`: raggruppa righe con stessi valori su uno o più attributi.
- `HAVING`: filtra i gruppi risultanti.

Esempio:

```
SELECT città, AVG(media)
FROM Studente
GROUP BY città
HAVING AVG(media) > 25;
```

Ordinamento con `ORDER BY`

```
ORDER BY cognome DESC, nome ASC;
```

Join e prodotto cartesiano

- Più tabelle in `FROM` generano un **CROSS JOIN**
- I `JOIN` espliciti (da lezione 4) sono trattati più avanti

Esempi utili

Raggruppamento e filtro:

```
SELECT LOWER(città) AS città, COUNT(*) AS NumStudenti
FROM Studente
GROUP BY LOWER(città)
HAVING COUNT(*) > 1;
```

Ordinamento con espressioni:

```
SELECT nome, UPPER(cognome) AS cognome_maiuscolo
FROM Studente
ORDER BY cognome_maiuscolo;
```

Raggruppamenti e JOIN in SQL

Raggruppamenti (`GROUP BY`, `HAVING`)

`GROUP BY`

Permette di dividere il risultato in gruppi di tuple con stessi valori su uno o più attributi.

```
SELECT città, COUNT(*)
FROM Studente
GROUP BY città;
```

`HAVING`

Filtra i gruppi risultanti (dopo il `GROUP BY`).

```
SELECT città, COUNT(*)
FROM Studente
GROUP BY città
HAVING COUNT(*) > 1;
```

> Nella clausola `SELECT`, puoi usare **solo** gli attributi raggruppati o **funzioni di aggregazione**.

JOIN tra tabelle

Tipi di JOIN

Tipo	Descrizione
`INNER JOIN`	Seleziona solo le righe combinate che soddisfano la condizione di join
`LEFT OUTER JOIN`	Mantiene tutte le righe della **prima tabella** e completa con NULL se non

Tipo	Descrizione
	combacia
`RIGHT OUTER JOIN`	Viceversa del precedente
`FULL OUTER JOIN`	Unisce tutto (match e non match) da entrambe le tabelle

Esempio:

```
SELECT I.cognome, R.nomeRep
FROM Impiegato I
INNER JOIN Reparto R ON I.nomerep = R.nomerep;
```

Interrogazioni Nidificate (Subquery)

Cos'è una subquery (interrogazione nidificata)?

È una query **annidata** all'interno di un'altra query, tipicamente nelle clausole `FROM`, `WHERE` o `HAVING`.

---

1. Subquery nella clausola `FROM`

La subquery genera una **tabella temporanea** su cui si lavora.

```
SELECT titolo, prezzoIntero
FROM Mostra,
     (SELECT MAX(prezzoIntero) AS prezzoMax FROM Mostra) AS T
WHERE prezzoIntero = prezzoMax;
```

---

2. Subquery nella clausola `WHERE`

Utilizzano **operatori speciali** per confrontare un valore con l'output della subquery.

Operatori per subquery

- **`EXISTS` / `NOT EXISTS`**

Controlla se la subquery restituisce almeno una riga.

```
SELECT P.nome, P.cognome
FROM Persona P
WHERE EXISTS (
  SELECT 1
  FROM Docenza D JOIN InsErogato IE ON D.id_inserogato = IE.id
  WHERE IE.crediti > 24
    AND IE.annoaccademico = '2010/2011'
    AND D.coordinatore = '0'
    AND D.id_persona = P.id
  GROUP BY D.id_persona
  HAVING COUNT(*) >= 2
);
```

> **Nota:** quando nella subquery si usano attributi della query esterna si parla di **correlated subquery** (data binding).

- **`IN` / `NOT IN`**

Controlla se un valore è presente tra quelli restituiti dalla subquery.

```
SELECT I.nome, I.cognome
FROM Impiegato I
WHERE ROW(I.nome, I.cognome) IN (
  SELECT nome, cognome
  FROM ImpiegatoAltraAzienda
);
```

- **`ANY` / `SOME` e `ALL`**

Confrontano un valore con **qualsiasi** (`ANY`) o **tutti** (`ALL`) i valori restituiti dalla subquery.

```
-- Qualsiasi media annuale
SELECT DISTINCT I.nomeins, IE.crediti
FROM Insegn I JOIN InsErogato IE ON I.id = IE.id_insegn
WHERE IE.crediti < ANY (
  SELECT AVG(crediti)
  FROM InsErogato
  WHERE modulo = 0
  GROUP BY annoaccademico
);

-- Maggiore di tutti i crediti di un corso di studi
SELECT DISTINCT I.nomeins, IE.crediti
FROM Insegn I JOIN InsErogato IE ON I.id = IE.id_insegn
WHERE IE.modulo = 0
  AND IE.crediti > ALL (
    SELECT crediti
    FROM InsErogato
    WHERE id_corsostudi = 6 AND modulo = 0
);
```

---

3. Esempi avanzati: crediti e docenti

Obiettivo:

Trovare gli insegnamenti che:

- hanno **almeno due docenti**
- e hanno **crediti maggiori** di qualunque insegnamento del corso con `id=6`

Soluzione (con doppio JOIN)

```
SELECT DISTINCT I.nomeins
FROM Insegn I
JOIN InsErogato IE1 ON I.id = IE1.id_insegn
```

```
JOIN Docenza D1 ON IE1.id = D1.id_inserogato
JOIN InsErogato IE2 ON I.id = IE2.id_insegn
JOIN Docenza D2 ON IE2.id = D2.id_inserogato
WHERE IE1.modulo = 0 AND IE2.modulo = 0
  AND IE1.crediti > ALL (
    SELECT crediti FROM InsErogato
    WHERE id_corsostudi = 6 AND modulo = 0
  )
  AND IE2.crediti > ALL (
    SELECT crediti FROM InsErogato
    WHERE id_corsostudi = 6 AND modulo = 0
  )
  AND D1.id_persona <> D2.id_persona;
```

Alternativa più efficiente (usando `GROUP BY` e `HAVING`):

```
SELECT nomeInsegnamento
FROM (
  SELECT DISTINCT I.nomeins AS nomeInsegnamento, D.id_persona
  FROM Insegn I
  JOIN InsErogato IE ON I.id = IE.id_insegn
  JOIN Docenza D ON IE.id = D.id_inserogato
  WHERE IE.modulo = 0
    AND IE.crediti > ALL (
      SELECT crediti FROM InsErogato
      WHERE id_corsostudi = 6 AND modulo = 0
    )
) AS Risultato
GROUP BY nomeInsegnamento
HAVING COUNT(*) >= 2;
```

Interrogazioni di tipo insiemistico e Viste

---

1. Operatori insiemistici in SQL

Gli operatori insiemistici permettono di **combinare i risultati di più query**. Devono avere:

- stesso numero di colonne
- tipi compatibili

Sintassi generale

```
query1
{ UNION | INTERSECT | EXCEPT } [ALL]
query2
```

Significato operatori

Operatore	Significato	Duplichi ammessi con `ALL`?
`UNION`	Unione delle righe di entrambe le query	Sì (con `ALL`)
`INTERSECT`	Solo le righe comuni	Sì (con `ALL`)
`EXCEPT`	Righe della prima query escluse le comuni	Sì (con `ALL`)

Esempi pratici

`UNION ALL` – mantenere duplicati

```
SELECT nomeins FROM Insegn WHERE NOT nomeins LIKE 'A%'
UNION ALL
SELECT nome FROM CorsoStudi WHERE NOT nome LIKE 'A%';
```

`INTERSECT ALL` – insegnamenti che sono anche corsi di laurea

```
SELECT nomeins FROM Insegn
INTERSECT ALL
SELECT nome FROM CorsoStudi;
```

`EXCEPT` – insegnamenti che non sono corsi di laurea

```
SELECT nomeins FROM Insegn
EXCEPT
SELECT nome FROM CorsoStudi;
```

---

2. Le VISTE (VIEW)

Cos’è una vista

È una **tabella virtuale** che mostra il risultato di una query salvata con un nome. Ogni volta che si interroga una vista, viene **eseguita la query sottostante**.

Sintassi

```
CREATE [TEMP] VIEW nome [(col1, col2, ...)] AS
SELECT ...
```

- `TEMP`: la vista è temporanea e scompare alla disconnessione.
- I nomi delle colonne sono opzionali.

Esempi pratici

Vista con join tra `InsErogato` e `Insegn`

```
CREATE TEMP VIEW InsErogatiCompleti AS
SELECT I.nomeins, I.codiceins, IE.*
FROM InsErogato IE
JOIN Insegn I ON IE.id_insegn = I.id;
```

Vista aggregata con `GROUP BY` e `MAX`

Obiettivo: trovare il corso con più insegnamenti distinti.

```
CREATE TEMP VIEW InsCorsoStudi (Nome, NumIns) AS
```

```
SELECT CS.nome, COUNT(DISTINCT I.nomeins)
FROM CorsoStudi CS
JOIN InsErogato IE ON CS.id = IE.id_corsostudi
JOIN Insegn I ON IE.id_insegn = I.id
GROUP BY CS.nome;
```

Ora si può interrogare la vista:

```
SELECT Nome, NumIns
FROM InsCorsoStudi
WHERE NumIns = (
  SELECT MAX(NumIns) FROM InsCorsoStudi
);
```

Indici e Ottimizzazione in PostgreSQL

---

1. Obiettivo degli indici

Gli **indici** migliorano la velocità di accesso ai dati su grandi tabelle. PostgreSQL li usa per:

- velocizzare le ricerche (`WHERE`, `JOIN`, `ORDER BY`)
- ridurre il numero di pagine da leggere

  Gli indici **non** sono gratuiti: aumentano il costo di inserimento, aggiornamento e occupano memoria.

---

2. Creazione e tipi di indici

Sintassi base

```
CREATE INDEX nome_indice ON nome_tabella (colonna [, ...]);
```

Tipi di indici principali

Tipo	Caratteristiche principali
`B-TREE`	Predefinito. Ottimo per confronti di ordine (=, <, >, ...)
`HASH`	Ottimo solo per = (uguale). Più veloce ma meno flessibile
`GIN`	Per ricerche su array, JSON, testo
`GiST`	Utilizzato per dati geometrici/spaziali

---

3. Quando usare indici

- Colonne molto usate nei filtri `WHERE`
- Colonne usate nei `JOIN`

- Colonne in `ORDER BY` e `DISTINCT`
- Su **grandi tabelle**: su tabelle piccole spesso non servono

  Evitare indici su colonne con pochi valori distinti (es. booleani)

---

4. Indici multi-attributo

```
CREATE INDEX idx_nome_cognome ON Persona(nome, cognome);
```

- Utili quando le query filtrano in base a più colonne.
- L'ordine è importante: l'indice è utile solo se le query usano almeno **il primo attributo**.

---

5. Indici su espressioni

PostgreSQL consente anche indici su trasformazioni:

```
CREATE INDEX idx_lower_nome ON Persona(LOWER(nome));
```

Ottimo per ricerche case-insensitive.

---

6. Uso di `EXPLAIN` per analizzare una query

```
EXPLAIN ANALYZE SELECT * FROM Persona WHERE cognome = 'Rossi';
```

- Mostra il **piano di esecuzione**:
  - **Seq Scan** = scansione sequenziale
  - **Index Scan** = uso di indice
  - **Bitmap Index Scan** = strategia ibrida

Interpretare l'output:

- Tempo totale
- Numero righe lette
- Tipologia di scansione
- Costo stimato in funzione I/O e CPU

---

7. Ottimizzazione pratica: esempi

Query lenta (senza indice):

```
SELECT * FROM Persona WHERE LOWER(cognome) = 'rossi';
```

Ottimizzata:

1. Creare indice su `LOWER(cognome)`
2. Query sfrutterà `Index Scan`

---

8. Gestione indici

- Eliminare:

  ```
  DROP INDEX nome_indice;
  ```

- Analisi periodica utile:

  ```
  ANALYZE nome_tabella;
  ```

Transazioni e concorrenza

---

1. Comandi principali per gestire transazioni

```
BEGIN;          -- Inizio transazione
...             -- Comandi SQL
COMMIT;         -- Conferma modifiche
ROLLBACK;       -- Annulla tutto
```

PostgreSQL esegue ogni istruzione come **transazione implicita** se non si usa `BEGIN`.

---

2. Livelli di isolamento in PostgreSQL

Livello di Isolamento	Lettura sporca	Lettura inconsistente	Perdita di aggiornamento	Aggiornamento fantasma	Inserimento fantasma
READ UNCOMMITTED	❌ No	❌ No	❌ No	❌ No	❌ No
READ COMMITTED	✅ Sì	❌ No	❌ No	❌ No	❌ No
REPEATABLE READ	✅ Sì	✅ Sì	❌ No	❌ No	❌ No
SERIALIZABLE	✅ Sì	✅ Sì	✅ Sì	✅ Sì	✅ Sì

Esempio:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN;
...
COMMIT;
```

---

3. Esempi pratici

Esempio 1: uso base di transazione

```
BEGIN;
UPDATE Studente SET media = media + 1 WHERE matricola = 'IN0003';
COMMIT;
```

Esempio 2: ROLLBACK su errore

```
BEGIN;
INSERT INTO Impiegato VALUES (...);
-- errore!
ROLLBACK;
```

1. Lettura sporca (dirty read)

Cos'è:

Una transazione legge dati **modificati ma non ancora confermati (committed)** da un'altra transazione. Se l'altra transazione fa `ROLLBACK`, la lettura è "sporca".

Accade con: `READ UNCOMMITTED`

Esempio:

```
-- Transazione T1
BEGIN;
UPDATE Conto SET saldo = saldo - 100 WHERE id = 1;

-- Transazione T2 (nel frattempo)
SELECT saldo FROM Conto WHERE id = 1;  -- legge valore già modificato da T1 (sporco)

-- T1 annulla le modifiche
ROLLBACK;
```

T2 ha letto un **valore che non è mai esistito realmente** nel DB confermato.

---

2. Lettura inconsistente

Cos'è:

Una transazione legge la **stessa riga due volte**, ma ottiene valori diversi perché un’altra transazione la ha **modificata** nel frattempo.

Accade con: `READ COMMITTED`

Esempio:

```
-- Transazione T1
BEGIN;
SELECT saldo FROM Conto WHERE id = 1;  -- restituisce 500

-- Transazione T2
BEGIN;
UPDATE Conto SET saldo = 400 WHERE id = 1;
COMMIT;

-- T1 (di nuovo)
SELECT saldo FROM Conto WHERE id = 1;  -- ora ottiene 400
COMMIT;
```

> T1 vede **valori incoerenti** all’interno della stessa transazione.

---

3. Inserimento fantasma

Cos'è:

Una transazione esegue una **query condizionata**, ma tra due esecuzioni un’altra transazione **inserisce una riga** che **soddisfa quella condizione**, e appare “come un fantasma”.

Accade con: `REPEATABLE READ`

Esempio:

```
-- T1
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT COUNT(*) FROM Clienti WHERE età > 30;  -- 5 risultati

-- T2
BEGIN;
INSERT INTO Clienti VALUES ('Mario', 45);
COMMIT;

-- T1 (di nuovo)
SELECT COUNT(*) FROM Clienti WHERE età > 30;  -- ora 6 risultati (anomalia!)
COMMIT;
```

> **Una riga “fantasma” è apparsa** nella stessa query durante la transazione.

---

4. Aggiornamento fantasma (phantom update)

Cos'è:

Come l’inserimento fantasma, ma riguarda **modifiche di righe che entrano/escono dal range** della query.

Accade con: `REPEATABLE READ`

Esempio:

```
-- T1
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT * FROM Ordini WHERE quantità > 10;  -- include l'ordine #42

-- T2
BEGIN;
UPDATE Ordini SET quantità = 5 WHERE id = 42;
COMMIT;

-- T1 (di nuovo)
SELECT * FROM Ordini WHERE quantità > 10;  -- ora #42 non c'è più (anomalia)
COMMIT;
```

> L’insieme di righe **cambia** anche senza nuove righe → **“aggiornamento fantasma”**.

---

5. Perdita di aggiornamento

Cos'è:

Due transazioni leggono lo stesso valore e **scrivono modifiche concorrenti**, una sovrascrive l’altra **senza sapere**.

Accade con: `REPEATABLE READ`

Esempio:

```
-- T1
BEGIN;
SELECT saldo FROM Conto WHERE id = 1;  -- legge 100
-- intende fare saldo + 50

-- T2
BEGIN;
SELECT saldo FROM Conto WHERE id = 1;  -- legge 100
-- intende fare saldo - 20

-- T1
UPDATE Conto SET saldo = 150 WHERE id = 1;
COMMIT;

-- T2
UPDATE Conto SET saldo = 80 WHERE id = 1;
COMMIT;
```

Il saldo finale è 80, ma **l’aggiornamento di T1 è perso**, perché T2 ha sovrascritto **basandosi su un valore obsoleto**.

Accesso a PostgreSQL con Python (`psycopg2`)

---

1. Collegamento al database

```
import psycopg2

conn = psycopg2.connect(
    dbname="nome_db",
    user="utente",
    password="pwd",
    host="host"
)
cur = conn.cursor()
```

---

2. Esecuzione di query

`execute` + `fetchone` / `fetchall`

```
cur.execute("SELECT nome FROM Studente")
risultati = cur.fetchall()
for riga in risultati:
    print(riga)
```

---

3. Parametrizzazione sicura (contro SQL Injection)

```
cur.execute("SELECT * FROM Studente WHERE matricola = %s", ("IN0001",))
```

---

4. Transazioni in Python

- In automatico, `psycopg2` gestisce una transazione per ogni connessione.

Commit e rollback

```
conn.commit()   # salva modifiche
conn.rollback() # annulla modifiche
```

Modalità context manager

```
with conn:
    with conn.cursor() as cur:
        cur.execute(...)
```

---

5. Esecuzioni multiple (`executemany`)

```
data = [("IN001", "Marco"), ("IN002", "Luca")]
cur.executemany("INSERT INTO Studente VALUES (%s, %s)", data)
```

MongoDB

1. Struttura dei dati

- Ogni **documento** è un oggetto JSON-like:

```
{
  "_id": ObjectId("..."),
  "nome": "Arena",
  "città": "Verona",
  "prezzo": 20
}
```

- Le collezioni possono contenere documenti con **strutture diverse**.

---

2. Strumenti MongoDB

Strumento	Scopo
`mongosh`	Shell a riga di comando
Compass	Interfaccia grafica ufficiale
Atlas	Servizio MongoDB cloud
`mongodump`	Backup dati
`mongostat`, `mongotop`	Monitoraggio prestazioni

---

3. Connessione

Via `mongosh`:

```
mongosh "mongodb+srv://cluster_url/" --username user
```

In Compass:

- Inserire URI completo.
- Esplora graficamente database, collezioni, indici, schema.

---

4. Operazioni CRUD

Create

```
db.museo.insertOne({
  nome: "Arena", città: "Verona", prezzo: 20
});
```

- Per più documenti:

```
db.museo.insertMany([{...}, {...}]);
```

Read

```
db.museo.find({ città: "Verona" });
db.museo.find({});
```

Operatori:

- Confronto: `$eq`, `$ne`, `$gt`, `$lt`, `$in`, `$or`
- Regex: `{ nome: { $regex: "^A" } }`
- Valori nulli: `{ campo: null }`, `{ campo: { $ne: null } }`

Proiezione (selezione campi):

```
db.museo.find({}, { nome: 1, prezzo: 1, _id: 0 });
```

Update

```
db.museo.updateOne(
  { nome: "Arena" },
  { $set: { prezzo: 25 } }
);
```

- `updateMany()`, `replaceOne()` disponibili

Delete

```
db.museo.deleteOne({ nome: "Arena" });
db.museo.deleteMany({ città: "Verona" });
```

---

5. Incapsulamento e riferimenti

- I documenti possono contenere **altri oggetti** (sub-documenti)
- Alternativa: uso di **riferimenti** (ObjectId) tra collezioni
- Possibile **denormalizzazione** per aumentare prestazioni

---

6. Aggregazioni: `aggregate()`

Pipeline di trasformazioni sui documenti:

```
db.student_grades.aggregate([
  { $match: { semestre: "Fall 2023" } },
  { $group: { _id: "$student_id", media: { $avg: "$grade" } } }
]);
```

Operator mapping SQL → MongoDB

SQL	MongoDB
WHERE	$match
GROUP BY	$group
SELECT	$project
HAVING	$match
JOIN	$lookup
ORDER BY	$sort

---

7. Join con `$lookup`

```
db.clienti.aggregate([
  {
    $lookup: {
      from: "ordini",
      localField: "_id",
      foreignField: "clienteId",
      as: "ordini"
    }
  }
]);
```