Corso di Basi di Dati (Laboratorio)
Introduzione a PostgreSQL

Magdalena M. Solitro

5 luglio 2020

Indice

Capitolo 1

Introduzione

Structured Query Language (SQL) è il linguaggio più diffuso per l'interazione con database relazionali (RDBMS), è stato definito negli anni '70 per poi essere standardizzato nel corso degli anni '80 e '90 da ISO e IEC.
I costrutti di SQL possono essere distinti in quattro categorie:

- *Data Definition Language (DDL)*, linguaggio per la definizione delle strutture dati e dei vincoli di integrità
- *Data Manipulation Language (DML)*, linguaggio per la manipolazione dei dati, consente l'inserimento, l'aggiornamento e la cancellazione di dati.
- *Data Query Language (DQL)*, linguaggio di interrogazione del database, serve per ottenere i dati desiderati
- *Data Control Language (DCL)*, linguaggio per controllare la base di dati (non sarà trattato in questo corso)

Originariamente, SQL era nato come linguaggio *puramente dichiarativo*, ma nel corso degli anni si è evoluto ed è stata introdotta la possibilità di inserire costrutti procedurali, istruzioni per il controllo di flusso, tipi di dati definiti dall'utente e altre estensioni. A partire dalla definizione dello standard SQL:1999 molte di queste estensioni sono state formalmente adottate come parte integrante di SQL nella sezione SQL/PSM dello standard.[1]

1.1 Sintassi

Un comando SQL è una sequenza di token terminate da un punto e virgola (;).
I commenti sono preceduti da due trattini consecutivi (una singola riga di commento), oppure sono delimitati da `/* ... */` (commenti multilinea).

[1] fonte: https://it.wikipedia.org/wiki/Structured_Query_Language

È importante ricordare che **le keyword e gli identificatori che non siano racchiusi tra doppi apici sono casi insensitive**. Quindi:

```
SELECT * FROM user;
```

è equivalente a:

```
select * from user;
```

ed è equivale anche a:

```
sElEcT * FrOm uSer;
```

Gli operatori di base in SQL sono:

- aritmetici: addizione (`+`), sottrazione (`-`), moltiplicazione (`*`), divisione (`/`)
- di confronto: minore (`<`), minore o uguale (`<=`), uguale (`=`), maggiore (`>`), maggiore o uguale (`>=`), diverso (`<>`)
- logici: not (`NOT`), and (`AND`), or (`OR`)

Capitolo 2

Data Definition Language (DDL)

Il *Data Definition Language (DDL)* comprende un sottoinsieme di istruzioni che servono a creare nuove tabelle, modificare la struttura di quelle esistenti, aggiungere e rimuovere vincoli o eliminare intere tabelle.
I comandi principali che effettuano questo operazioni sono: `CREATE TABLE`, `DROP TABLE`, `ALTER TABLE`...

2.1 Creazione di tabelle

La creazione di una nuova tabella si effettua tramite un comando di questo tipo:

```
CREATE TABLE persona (
    nome TEXT,
    cognome VARCHAR(30),
    dataNascita DATE
);
```

Questa istruzione crea una tabella chiamata `persona` con tre colonne: la prima si chiama `nome` e contiene un tipo `TEXT`, la seconda si chiama `cognome` e contiene un tipo `VARCHAR(30)`, infine la terza si chiama `dataNascita` e contiene un tipo `DATE`.
Quando si definisce tabella, è obbligatorio specificare il dominio di ogni colonna e si possono anche inserire dei vincoli che la tabella o i suoi singoli attributi devono rispettare.
Se la tabella con questo nome esiste già, si incorre in un errore: può essere d'aiuto, quindi, specificare anche la clausola `IF NOT EXISTS` subito dopo `CREATE TABLE`:

```
CREATE TABLE IF NOT EXISTS persona(
    ...
)
```

In questo modo, se esiste già una tabella con lo stesso nome, il sistema semplicemente salta questa istruzione.

2.2 Eliminazione di una tabella

Quando una tabella non è più necessaria, si può rimuovere dal database con il seguente comando:

```
DROP TABLE persona;
```

Tuttavia, se la tabella non esiste, si incorre in un errore: per questo motivo, può essere d'aiuto specificare la clausola `IF EXISTS`:

```
DROP TABLE IF EXISTS persona;
```

2.3 Vincoli

Prima di introdurre la sezione in cui parliamo di come modificare una tabella, è bene conoscere quali vincoli si possono specificare.
PostgreSQL fornisce una serie di vincoli che permettono di controllare il tipo di dato che viene inserito in una certa colonna, sollevando un errore se questo non dovesse rispettare il vincolo, oppure di specificare che una certa colonna contiene un valore "speciale" per una determinata tabella, ovvero è una chiave primaria, una superchiave, una chiave esterna o una chiave esportata.

2.3.1 DEFAULT

A una colonna può essere assegnato un valore di default. Questo fa sì che, se una nuova riga viene inserita nella tabella ma non viene specificato il valore per quella colonna, il sistema riempie automaticamente quella colonna con il valore di default.
Se questo vicolo non viene specificato, invece, alla colonna si assegna `NULL`. Nel vincolo di default può anche essere inserita una espressione, che verrà valutata nel momento in cui è richiesto di inserire il valore di default.
Esempio:

```
CREATE TABLE prodotto(
    nome TEXT,
    identificatore CHAR(10),
    prezzo NUMERIC(4,2) DEFAULT 10.00
);
```

Se non viene specificato il prezzo del prodotto, questo viene messo automaticamente a 10.00 .

2.3.2 CHECK

Un vincolo `CHECK` permette di definire vincoli molto generici: in particolare, impone che una certa colonna soddisfi l'espressione booleana specificata. Ad esempio:

```
CREATE TABLE prodotto(
    nome TEXT,
    identificatore CHAR(10),
    prezzo NUMERIC(4,2) CHECK (prezzo > 0)
);
```

Nel momento in cui viene inserita una riga nella tabella `prodotto`, si controlla che il valore nella colonna `prezzo` sia superiore a 0. Se non soddisfa questa condizione, viene lanciato un errore.
Possibile definire anche dei `CHECK` a livello di tabella:

```
CREATE TABLE prodotto(
    nome TEXT,
    identificatore CHAR(10),
    prezzo NUMERIC(4,2) CHECK (prezzo > 0),
    prezzoScontato NUMERIC(4,2) CHECK(prezzoScontato > 0),
    CHECK (prezzo > prezzoScontato)
);
```

Si definisce `CHECK` di tabella un vincolo che riguarda più attributi all'interno della tabella. Questi si specificano **dopo** aver definito tutti gli attributi.
Un `CHECK` di attributo riguarda una condizione su un attributo singolo e viene dichiarato in-line con l'attributo cui si riferisce.

Osservazione: in realtà, è possibile specificare i `CHECK` di attributo e di tabella *ovunque*: i `CHECK` di tabella possono essere specificati in-line con altri attributi e i `CHECK` di attributo possono essere specificati in fondo alla definizione. Le convenzioni sopra specificate sono buone pratiche per rendere la specifica più chiara.

2.3.3 NOT NULL

Il vincolo `NOT NULL` richiede che l'attributo in questione sia sempre valorizzato.
Un vincolo di questo tipo **deve essere scritto in-line con l'attributo cui si riferisce**.

Esempio

Imponiamo che gli attributi `nome` e `identificatore` siano non nulli:

```
CREATE TABLE prodotto(
```

```
    nome TEXT NOT NULL,
    identificatore CHAR(10) NOT NULL,
    prezzo NUMERIC(4,2) CHECK (prezzo > 0),
    prezzoScontato NUMERIC(4,2) CHECK(prezzoScontato > 0),
    CHECK (prezzo > prezzoScontato)
);
```

Questo vincolo è semanticamente equivalente a scrivere `CHECK(attributo IS NOT NULL)`, ma in PostgreSQL specificare un vincolo `NOT NULL` è più efficiente.
Se si vuole aggiungere una nuova colonna a una tabella già esistente e si impone il vincolo `NOT NULL`, è bene specificare anche un valore di default: normalmente, infatti, quando si aggiunge una nuova colonna il sistema la riempie inizialmente con `NULL`, ma se si specifica `NOT NULL` bisogna anche fornire il valore con cui riempire le colonne!

2.3.4 UNIQUE

Vincolo usato per specificare *superchiavi*. Definire un vincolo `UNIQUE` per un attributo (o un insieme di attributi), assicura che il valore presente in quell'attributo (o insieme di attributi) sia unico all'interno della tabella. **Non deve essere utilizzato per definire chiavi primarie**, per le quali esiste una keyword apposita.

Esempio

Imponiamo che il nome del prodotto sia unico all'interno della tabella:

```
CREATE TABLE prodotto(
    nome TEXT UNIQUE,
    identificatore CHAR(10) NOT NULL,
    prezzo NUMERIC(4,2) CHECK (prezzo > 0),
    prezzoScontato NUMERIC(4,2) CHECK(prezzoScontato > 0),
    CHECK (prezzo > prezzoScontato)
);
```

La seguente scrittura è del tutto equivalente:

```
CREATE TABLE prodotto(
    nome TEXT NOT NULL,
    identificatore CHAR(10) NOT NULL,
    prezzo NUMERIC(4,2) CHECK (prezzo > 0),
    prezzoScontato NUMERIC(4,2) CHECK(prezzoScontato > 0),
    CHECK (prezzo > prezzoScontato),
```

```
    UNIQUE(nome)
);
```

Aggiungere un vincolo `UNIQUE` comporta la creazione automatica di un indice B-tree sulla colonna (o gruppo di colonne) specificate nel vincolo.

Osservazione: Se due righe della tabella presentano il valore `NULL` nella colonna soggetta a un vincolo `UNIQUE`, **il vincolo di unicità non è violato**, perchè in SQL due `NULL` sono sempre diversi tra loro! (si veda l'approfondimento sui valori `NULL` nella sezione 2.3).
Quindi, un attributo con vincolo `UNIQUE` può anche essere nullo, a differenza di un attributo con vincolo `PRIMARY KEY`.

2.3.5 PRIMARY KEY

È un vincolo utilizzato per denotare la *chiave primaria* di una tabella, ovvero quell'attributo che identifica una riga all'interno della base di dati.
Alcune caratteristiche importanti:

- si può usare **una sola volta** all'interno di una tabella
- definire un vincolo `PRIMARY KEY` **implica** un vincolo `NOT NULL`
- può essere specificato su un solo attributo o su una n-upla di attributi

Aggiungere un vincolo `PRIMARY KEY` comporta la creazione automatica di un indice B-tree sulla colonna (o gruppo di colonne) specificate nel vincolo.

Esempio

Dichiariamo l'attributo `identificatore` come chiave primaria per la tabella `prodotto`. Il vincolo `NOT NULL`, quindi, non è più necessario.

```
CREATE TABLE prodotto(
    nome TEXT NOT NULL,
    identificatore CHAR(10) PRIMARY KEY,
    prezzo NUMERIC(4,2) CHECK (prezzo > 0),
    prezzoScontato NUMERIC(4,2) CHECK(prezzoScontato > 0),
    CHECK (prezzo > prezzoScontato),
    UNIQUE(nome)
);
```

Attenzione

A livello teorico, il modello relazionale **impone** che ogni tabella abbia una chiave primaria associata. Tuttavia, nell'implementazione di PostgreSQL questo regola non è obbligatoria, quindi creare una tabella che non abbia una chiave primaria è accettato (ovvero, non viene dato errore), anche se fortemente sconsigliato.

2.3.6 Chiavi esportate

Una chiave esportata è una colonna (o un gruppo di colonne) il cui valore deve comparire anche in una certa colonna (o un gruppo di colonne) di un'altra tabella.
In PostgreSQL ci sono due modi per specificare la chiave esportata di una tabella:

- `attribute TYPE REFERENCES anotherTable(anotherAttribute)`
  Questo è un **vincolo di attributo** e si utilizza quando la chiave esportata è composta da una singola colonna.

- `FOREIGN KEY(a_1, ..., a_n) REFERENCES anotherTable(b_1,...,b_n)`
  Questo è un **vincolo di tabella**, si usa quando il vincolo di integrità referenziale coinvolge più attributi.
  `a_1,...,a_n` = chiave **esportata**
  `b_1,...,b_n` = chiave **esterna**

Osservazione: quando si specifica una chiave esportata, è possibile omettere gli attributi della chiave esterna: il sistema assume che gli attributi a cui ci si riferisce sono quelli che compongono la chiave primaria della tabella esterna.

Attenzione

Quando la tabella A referenzia la tabella B, bisogna assicurarsi che la tabella B sia già stata dichiarata, altrimenti si incorre in un errore!

Anche in questo caso, la sintassi per specificare una chiave esportata è abbastanza flessibile: nonostante venga considerata buona pratica adottare il formato presentato sopra, quando la chiave esportata è composta da un singolo attributo non è necessario specificarlo in-line! Si può anche scrivere:

```
CREATE TABLE thisTable(
    ...
    FOREIGN KEY(a) REFERENCES anotherTable
);
```

ovvero si può utilizzare la sintassi della chiave esterna multiattributo.

2.4 Modificare la struttura di una tabella

È possibile modificare la struttura di una tabella in molti modi diversi. Le azioni concesse sono:

- aggiungere, modificare, rimuovere colonne
- aggiungere, modificare, rimuovere vincoli
- modificare i valori di default
- modificare il dominio di una colonna
- rinominare colonne
- rinominare tabelle

Tutti i comandi che effettuano modifiche di questo tipo sono introdotte dalla keyword `ALTER TABLE`.

2.4.1 Aggiungere, modificare, rimuovere colonne

L'istruzione che permette di modificare una colonna a una tabella già esistente è:

```
ALTER TABLE table_name
[ADD [COLUMN] <column_name> <domain> [<constraints>] ]
[DROP [COLUMN] <column_name> [CASCADE]]
[ALTER [COLUMN] <column_name> ...]
```

Cominciamo dalla keyword `ADD COLUMN`, che permette di aggiungere una nuova colonna alla tabella: quando si aggiunge una nuova colonna, è necessario specificare almeno il nome e il dominio, mentre l'aggiunta di vincoli è opzionale e può comunque essere effettuata in seguito. Se non viene specificato un valore di default, la colonna viene riempita con `NULL`.

Osservazione: a partire da PostgreSQL 11, l'aggiunta di una colonna non comporta l'aggiornamento immediato di tutte le righe della tabella al momento dell'esecuzione di `ALTER TABLE`. Invece, il valore di default viene ritornato la prima volta che si accede alla riga dopo la modifica e verrà effettivamente memorizzato quando la tabella verrà riscritta. Questo comporta un notevole vantaggio in termini di efficienza. L'unica eccezione vale per i valori di default "volatili", ovvero quei valori che cambiano a seconda dell'istante in cui sono creati (es. `clock_timestamp()`.)

L'eliminazione di una colonna è leggermente più complicata nel caso in cui ci siano altre colonne che dipendono da essa (ad esempio, la colonna è una chiave esterna o esportata). In questo caso, bisogna unire all'istruzione di `DROP` anche la clausola `CASCADE`, che comporta l'eliminazione a cascata di tutti gli elementi o vincoli del database che dipendono da quello che si vuole rimuovere.

Una colonna può essere rinominata tramite la seguente istruzione

```
ALTER TABLE [ IF EXISTS ] <table_name>
RENAME [ COLUMN ] <column_name> TO <new_column_name>
```

È possibile anche modificare il dominio di una colonna, ma non sempre è possibile. Per essere precisi, modificare il dominio di una colonna richiede di fare un cast di tutti i valori contenuti nella colonna stessa: è sicuramente possibile convertire, ad esempio, un tipo `DATE` a `TEXT`, ma non posso di certo convertire un tipo `DATE` a `SMALLINT` o viceversa!
Per modificare il dominio di una colonna, si scrive:

```
ALTER TABLE <table_name> ALTER COLUMN <column_name> TYPE <new_type>
```

Se la conversione non è possibile, viene tornato un errore.
A volte è possibile cercare di "forzare" il cast aggiugendo la clausola `USING`:

```
ALTER TABLE <table_name> ALTER COLUMN <column_name> TYPE <new_type> USING
    <old_type>::<new_type>
```

ma non sempre questo funziona: riprendendo lo stesso esempio di prima, in nessun caso è possibile convertire un `DATE` a `SMALLINT`, nemmeno attraverso un cast forzato.

2.4.2 Aggiungere, modificare, rimuovere vincoli

PostgreSQL permette di aggiungere, modificare e rimuovere vincoli anche dopo la creazione della tabella. Tuttavia, bisogna tenere a mente che, mentre i vincoli di tabella sono sempre alterabili, **non sempre è possibile modificare i vincoli di una attributo!**
La sintassi generica per aggiugnere, eliminare o modificare un vincolo (di colonna o di tabella) è la seguente:

```
ALTER TABLE <table_name> [ADD | DROP | ALTER] <constraint>
```

Esempio

Aggiungo alla tabella `test` i seguenti vincoli:

```
ALTER TABLE test ADD PRIMARY KEY(id);
ALTER TABLE test ADD CONSTRAINT "unique_name_value" UNIQUE(name, value);
```

```
ALTER TABLE test ALTER COLUMN duration SET NOT NULL;
ALTER TABLE test ALTER COLUMN name SET DEFAULT 'unknown';
```

Si noti che:

- la sintassi che permette di aggiungere vincoli di colonna è identica a quella per aggiungere vincoli di tabella (a eccezione di `NOT NULL` e `DEFAULT`, si veda il terzo punto)
- è possibile assegnare un nome ai vincoli, ma bisogna aggiugnere la keyword `CONSTRAINT` (questo torna utile se in seguito si vuole poter alterare o eliminare quel vincolo)
- gli unici vincoli di colonna che richiedono una sintassi "speciale" sono `NOT NULL` e `DEFAULT`. Più precisamente, il comando corretto per impostare questi vincoli è:

```
ALTER TABLE <table_name>
ALTER COLUMN <column_name> [SET | DROP] [NOT NULL | DEFAULT]
    [<default_value>];
```

Assegnare un identificativo a un vincolo è necessario per poterlo eliminare o modificare in seguito:

```
ALTER TABLE test ADD CONSTRAINT "unique_name_value" UNIQUE(name, value);
...
ALTER TABLE test DROP CONSTRAINT "unique_name_value";
```

Questo nome può essere successivamente modificato tramite la seguente istruzione:

```
ALTER TABLE [ IF EXISTS ] <table_name>
RENAME CONSTRAINT <constraint_name> TO <new_constraint_name>
```

2.4.3 Rinominare tabelle

Rinominare una tabella è sempre possibile, a patto che nel database non esista già una tabella con lo stesso nome. Per effettuare questa operazione, si può eseguire il seguente comando:

```
ALTER TABLE [ IF EXISTS ] <table_name>
RENAME TO <new_table_name>
```

2.4.4 Rinominare colonne

Con una sintassi simile a quella vista nella sezione precedente, è possibile riassegnare il nome a una certa colonna attraverso il seguente comando:

```
ALTER TABLE test RENAME [ COLUMN ] <column_name> TO <new_column_name>;
```

2.5 Domini

L'implementazione di PostgreSQL mette a disposizione svariati tipi di dati. I principali sono:

- tipi numerici
- tipi booleani
- tipi carattere/stringa
- tipi temporali

2.5.1 Tipi numerici

PostgreSQL fornisce diversi tipi per rappresentare quantità numeriche intere o decimali.
I tipi `SMALLINT, INTEGER, BIGINT` sono utilizzati per memorizzare numeri interi, quindi numeri privi di parte decimale. La differenza tra questi tre tipi sta nel range di valori che possono rappresentare:

Tipo	Spazio in memoria	Range
`smallint`	2 byte	[-32768, +32767]
`integer`	4 byte	$[-2^{31}, +2^{31} - 1]$
`bigint`	8 byte	$[-2^{63}, +2^{63} - 1]$

I tipi `NUMERIC, DECIMAL, REAL, DOUBLE PRECISION` sono invece usati per rappresentare valori decimali.

Tipo	Spazio in memoria	Range
`decimal`	variabile	fino a 131072 cifre prima della virgola, fino a 16383 cifre dopo la virgola
`numeric`	variabile	come `decimal`
`real`	4 byte	6 cifre decimali dopo la virgola
`double precision`	8 byte	15 cifre decimali dopo la virgola

`NUMERIC` e `DECIMAL` **sono sinonimi e vanno utilizzati quando si vogliono rappresentare quantità numeriche esatte**, robuste rispetto a errori di approssimazione che possono incorrere sia nel momento della memorizzazione, sia durante manipolazioni aritmetiche.
Per dichiarare un tipo `numeric` (o `decimal`) si usa la seguente sintassi:

```
NUMERIC(precision, scale)
```

dove `precision` rappresenta il numero totale di cifre presenti nel numero (quindi sia prima che dopo la virgola), mentre `scale` indica il numero di cifre presenti dopo la virgola.

Esempio

Il numero 924.12 è di tipo `NUMERIC(5,2)`.

Altre sintassi accettate sono:

```
NUMERIC(precision)
```

se `scale` è 0 quindi se il numero da rappresentare **non** ha parte decimale.
È comunque possibile inserire un numero decimale, ma le cifre della parte intera devono essere *al massimo* `precision` e il numero verrà memorizzato come intero, quindi senza parte decimale.
Se il numero di cifre della parte intera supera quello dichiarato da `precision`, si incorre in un errore di questo tipo:

```
ERROR: numeric field overflow
```

Infine, la seguente sintassi

```
NUMERIC
```

si usa per indicare un valore che può avere qualunque valore per `scale` e `precision`.
Il vantaggio dell'ultima notazione è che viene scongiurato l'unico caso in cui un tipo `numeric` può essere approssimato, ovvero il caso in cui il numero di cifre dopo la virgola supera quello dichiarato come `scale`.

Esempio

Se tento di memorizzare il numero 100.999 in una colonna di tipo `NUMERIC(5,2)`, questo viene arrotondato a 101.00, perchè il numero di cifre dopo la virgola nel numero originale è 3, mentre la colonna può memorizzare solo un massimo di 2 cifre decimali!
Se invece tento di memorizzare 1000.01, viene restituito un errore, perché la parte intera dovrebbe contenere 3 cifre. Quindi c'è "flessibilità" (con approssimazione) nel numero di cifre della parte decimale, mentre **non** c'è flessibilità nel numero di cifre della parte intera.
Nel caso in cui il numero di cifre del numero che vogliamo rappresentare sia inferiore a

quello specificato in `precision`, il numero viene memorizzato fisicamente **senza l'aggiunta di zeri** (*zero-padding*). Questo significa che `precision` e `scale` **sono solo un limite superiore**!

In aggiunta ai consueti valori numerici, il tipo `NUMERIC` consente anche il valore speciale `NaN`, il cui significato è `Not-A-Number`.
Qualunque operazione aritmetica che usi il valore `NaN` risulta in un `NaN`. Se si vuole assegnare questo valore a una colonna di tipo `NUMERIC`, si usa la seguente sintassi:

```
UPDATE <table_name> SET <column_name> = 'NaN';
```

Si noti che `NaN` è specificato tra singoli apici, come se fosse una stringa!

Attenzione

La maggior parte delle volte che un sistema deve implementare `NaN`, si considera questo valore come diverso da qualunque altro valore `NaN` (esattamente come con `NULL`). Tuttavia, per permettere l'ordinamento di valori `NUMERIC` in indici B-tree, **PostgreSQL tratta i valori `NaN` come uguali tra loro e maggiori di qualunque altro valore diverso da `NaN`.**

I tipi `REAL, DOUBLE PRECISION` rappresentano valori in virgola mobile e sono implementati secondo lo standard IEEE Standard 754 for Binary Floating-Point Arithmetic.
Questi tipi **non dovrebbero essere utilizzati se si vuole avere la garanzia di una rappresentazione esatta, priva di errori di approssimazione**.
Un tipico errore di approssimazione si ottiene quando si tenta di rappresentare un valore troppo vicino a zero. In questo caso si incorre in un underflow aritmetico.

I tipi serial Oltre ai tipi appena visti, esistono anche i `SMALLSERIAL, SERIAL, BIGSERIAL`: questi *non sono veri e propri tipi*, ma costituiscono piuttosto una notazione per creare un identificatore univoco su una colonna (simile alla proprietà `AUTO_INCREMENT` supportata da alcuni database). La seguente istruzione:

```
CREATE TABLE <table_name>(
    <column_name> SERIAL,
    ...
)
```

crea una tabella in cui `<column_name>` contiene *valori interi*, i cui valori di default sono generati tramite un generatore di sequenza. In altre parole, la colonna dichiarata di tipo `SERIAL` costituisce un identificatore (chiave primaria o superchiave); se quando si inserisce una nuova riga non si specifica un valore per quella colonna, il sistema genera automaticamente un nuovo

valore, che risulta essere il successore di quello precedentemente generato.

Esempio

Si consideri la tabella `test_serial` generata dalla seguente istruzione:

```
CREATE TABLE test_serial(
    x SERIAL,
    y VARCHAR(20),
    z NUMERIC(5,2)
);
```

Posso effettuare degli inserimenti senza specificare il valore per `x`:

```
INSERT INTO test_serial(y, z) VALUES('hello', 5.2), ('world', 12.9375),
    ('joy', 9.12);
```

ottenendo le seguenti righe:

```
x |   y   | z
---------------------
1 | hello |  5.2
2 | world | 12.94
3 |  joy  | 9.12
```

Può essere utile specificare comunque un vincolo `UNIQUE` o `PRIMARY KEY` per evitare che valori duplicati siano inseriti per errore.
Ad esempio, in riferimento all'esempio appena fatto, si potrebbe inserire una riga specificando un valore `x` pari a 6. Se poi si inseriscono altre righe facendo generare automaticamente i valori per `x`, il sistema calcola il valore successivo a partire dall'*ultimo valore automaticamente generato*, in questo caso 3. Inserendo altre 3 righe, verrebbero generati per `x` i valori 4, 5 e 6, senza incorrere in alcun errore a causa del valore duplicato.

2.5.2 Tipi booleani

Un valore booleano è identificato dalla keyword `BOOLEAN` oppure, in modo equivalente, `BOOL`.
Può avere tre stati possibili: `true, false` oppure `null`, che è il modo in cui SQL rappresenta uno stato "sconosciuto".
Esistono vari modi per rappresentare questi stati:

- lo stato `vero` può essere codificato dalla *keyword* `true`, oppure dalle seguenti *stringhe*: `'true', 't', 'yes', 'on', '1'`

- lo stato `falso` può essere codificato dalla *keyword* `false`, oppure dalle seguenti *stringhe*: `'false', 'f', 'no', 'off', '0'`

Per evitare ambiguità, si consiglia di usare le keyword `true` e `false`, ma le stringhe sopra elencate vengono comunque riconosciute e, se immesse in una colonna di tipo `BOOLEAN`, vengono interpretate nel loro valore booleano.

Osservazione: benché sia possibile inserire diverse stringhe, una colonna di tipo boolean conterrà sempre o il valore `true`, o il valore `false` (oppure `null`). Quindi il sistema *interpreta* la stringa e poi memorizza il corrispondente valore booleano.

Esempio

Si consideri il seguente codice SQL:

```
CREATE TABLE test(
    boolval BOOL
);

INSERT INTO test VALUES (true), ('1'), ('f'), (false), ('true'), ('off'),
    ('yes');

SELECT * FROM test;
```

Restituirà il seguente risultato:

```
boolval
--------
true
true
false
false
true
false
true
```

Si tenga presente che il valore `null` non ha di per sé un tipo, indipendentemente dal tipo della colonna in cui viene memorizzato.

Espressioni booleane con valori nulli È necessario fare alcune precisazioni in merito alla valutazione di un'espressione booleana in presenza di valori nulli.

PostgreSQL adotta una strategia *lazy* per la valutazione dell'espressione (anche detta valutazione *short-circuit*), quindi restituisce il risultato non appena incontra un valore che assicura un certo esito di valutazione.
Ad esempio, la seguente espressione

```
true or false
```

restituisce `true` se almeno uno dei due operandi è `true`. Dato che il valore `true` si trova in prima posizione, il sistema non valuta l'espressione completa (ovvero, non arriva a valutare anche `false`) perché avere un solo valore `true` è già sufficiente per decidere il valore di verità dell'espressione complessiva.
L'unico caso in cui una sequenza di espressioni in `or` resituisce `true` è quando *almeno un* valore viene valutato `true`.

```
true or null or null
```

Se pensiamo che `null` in SQL rappresenta un valore "sconosciuto", questo esito ha senso dal punto di vista logico: non importa se non conosciamo il valore delle altre espressioni, perché avere un singolo valore `true` è sufficiente per restituire con sicurezza l'esito `true`.
Se invece la disgiunzione di espressioni contiene solo `false` e `null`, **il valore di verità dell'espressione complessiva sarà sempre** `null`.

`false or null or false`	viene valutata `null`
`null or null`	viene valutata `null`

Anche questo ha senso dal punto di vista logico: se gli unici valori che riusciamo a valutare restituiscono `false`, non abbiamo elementi sufficienti per stabilire il valore di verità della disgiunzione: `null` potrebbe essere `true` oppure `false`, ma non lo sappiamo! Quindi siamo costretti a valutare l'espressione a `null`, perché non ci sono elementi sufficienti per determinare un qualsiasi altro valore di verità.

Diverso è il caso delle espressioni in `and`. Si adotta sempre una valutazione *lazy*, ma se si incontra anche solo un valore `null` durante la valutazione, il risultato complessivo dell'espressione sarà `null`. Ad esempio:

```
true and null and true
```

restituisce `null`.
In presenza di un valore `null`, è sufficiente valutare almeno una espressione come `false` per rendere il risultato complessivo `false`:

`false and null`	viene valutata `false`
`null and false`	viene valutata `false`

perché avere un singolo valore `false` in un'espressione in `and` è sufficiente per affermare che, indipendentemente dal valore che potrebbero rappresentare i `null`, l'espressione risulterà sempre e comunque falsa.

2.5.3 Tipi carattere/stringa

Di seguito riportiamo la sintassi accettata da PostgreSQL per la dichiarazione di tipi testuali:

Tipo	Descrizione
`char, character`	carattere singolo
`varchar(n), character varying(n)`	lunghezza variabile, con `n` limite massimo
`char(n), character(n)`	lunghezza fissa, blank-padded
`text`	lunghezza illimitata e variabile

Il tipo `CHAR` contiene semplicemente un carattere singolo, racchiuso tra singoli apici. Ad esempio, `'a'` può essere memorizzato come tipo `char`.

Il tipo `CHAR(n)`, sinonimo di `CHARACTER(n)`, rappresenta una stringa lunghezza fissa, specificata dal paramentro `n`.
Se la lunghezza della stringa è m $<$ n, viene effettuato il cosiddetto *"blank-padding"*, ovvero si riempiono i caratteri vuoti con degli spazi, fino a raggiungere la lunghezza `n`: questo perché il tipo `CHAR(n)` deve *garantire* che verranno memorizzati esattamente n caratteri.
Se si calcola la lunghezza della stringa, però, vengono contati **solo i caratteri significativi**, ovvero quelli che non sono spazi.

Esempio

Si consideri il seguente codice:

```
CREATE TABLE test (
    a CHAR(8)
);

INSERT INTO test VALUES ('ok              ');
```

```
SELECT a, char_length(a)
FROM test;
```

(Si supponga che nell'istruzione `INSERT` siano stati specificati 20 spazi vuoti).
Il risultato della `SELECT` sarà:

```
a  | char_length
----------------
ok |    2
```

Se la lunghezza della stringa è m > n, ma i caratteri in eccesso sono spazi vuoti, la stringa viene troncata a `n` caratteri.
Se la lunghezza della stringa è m > n e i caratteri in eccesso sono significativi (ovvero, non sono spazi vuoti), viene sollevato un errore.

Il tipo `VARCHAR(n)`, sinonimo di `CHARACTER VARYING(n)`, può memorizzare una stringa di lunghezza variabile, con un massimo di n caratteri.
Se la lunghezza della stringa è m < n, si memorizzano solamente m caratteri, con un conseguente risparmio di memoria.
Se la lunghezza della stringa è m > n e i caratteri in eccesso sono significativi, viene sollevato un errore.
Se la lunghezza della stringa è m > n e i caratteri in eccesso **non sono significativi** (quindi sono spazi bianchi), vengono memorizzati solo n caratteri.
Se una stringa di lunghezza m > n viene esplicitamente convertita (tramite cast, `::`) in un `varchar(n)` o in un `char(n)`, essa viene semplicemente troncata a n caratteri, senza sollevamento di errori!

Approfondimento: blank characters

I *blanks* (caratteri non significativi) vengono trattati in modo diverso a seconda del tipo della stringa in cui compaiono.
Il tipo `CHAR(n)` tiene conto solo dei caratteri significativi, ignorando quindi ogni spazio bianco nel momento in cui deve valutare una stringa. Infatti, il seguente comando

```
SELECT 'OK  '::CHAR(6) = 'OK'::CHAR(6)
```

restituisce `true`, perché si confrontano tra loro solo i caratteri significativi.
Se proviamo a calcolare la lunghezza della stringa:

```
SELECT char_length('OK '::CHAR(6))
```

il risultato sarà 2.

Il tipo `VARCHAR(n)`, invece, considera significativi anche gli spazi vuoti. A dimostrazione di ciò, si può facilmente verificare che il seguente comando:

```
SELECT 'OK   '::VARCHAR(5) = 'OK   '::VARCHAR(3)
```

restituisce `false`.
L'ultimo caso che presentiamo è il seguente:

```
SELECT 'OK   '::CHAR(3) = 'OK '::VARCHAR(5)
```

che restituisce `true`. Il motivo è che **in nessun caso è possibile fare confronti tra valori di tipi diversi**. Di conseguenza, anche se è stato fatto un cast per entrambe le stringhe, il sistema deve decidere di convertire una delle due stringhe al tipo dell'altra per poter decidere l'uguaglianza. In questo, il sistema ha probabilmente convertito la seconda stringa da `VARCHAR(5)` a `CHAR(3)` e ha considerato solo i caratteri significativi per il confronto.
Non c'è però una regola generale per decidere come effettuare le conversioni, dipende dalla situazione.

2.5.4 Tipi temporali

I tipi temporali permettono di rappresentare date, istanti e intervalli di tempo.

Nome	Spazio in memoria	Descrizione	Risoluzione
`date`	4 byte	data senza specificazione dell'ora	1 giorno
`time`	8 byte	ora del giorno, senza data	1 microsecondo
`time with timezone`	12 byte	ora del giorno, senza data, con time zone	1 microsecondo
`timestamp`	8 byte	data e ora	1 microsecondo
`timestamp with time zone`	8 byte	data e ora con time zone	1 microsecondo
`interval`	16 byte	intervallo di tempo tra due tipi `time`	1 microsecondo

I tipi `time, timestamp` e `interval` possono accettare un parametro di precisione, `(p)`, che specifica il numero di cifre che si voglio memorizzare nei secondi. Il range di valori che può assumere `p` è [0,6]. Se non si specifica alcuna precisione, questa viene impostata di default alla

precisione dell'input, ma comunque non può andare oltre i microsecondi.

Il tipo `interval` contiene di default i seguenti campi: `YEAR, MONTH, DAY, HOUR, MINUTE, SECOND`. È possibile tuttavia limitare il numero di campi rappresentati e la precisione dei secondi specificando due parametri aggiuntivi:

- `fields`, in cui si specificano tutti e soli i campi che si vogliono memorizzare nel risultato (sulla documentazione viene descritta la sintassi accettata per il paramentro `fields`[1])
- `(p)`, in cui si specifica la precisione desiderata per i secondi. La precisione massima, che è anche quella di default, arriva fino ai microsecondi

Si noti che se viene specificato sia `fields` che `p`, il campo `fields` **deve includere** anche `SECONDS`, poiché la precisione specificata riguarda unicamente i secondi.
La sintassi completa per esprimere un tipo `interval` è quindi:

```
INTERVAL [fields] [(p)]
```

Attenzione! Solo la differenza tra due tipi `time` o `timestamp` o `interval` restituisce come risultato un `interval`. La differenza tra due tipi `date` **non** restituisce un `interval`, bensì un intero con segno che rappresenta il numero di giorni trascorsi tra le due date.

2.6 Date

È possibile passare in input una data e/o un'ora in molti formati diversi, inclusi ISO, 8601, e POSTGRES, ma sempre sotto forma di stringa. PostgreSQL si occuperà di fare il *parsing* della stringa e di interpretare il suo valore come data (si veda la documentazione per un elenco completo dei formati accettati).
È importante sottolineare che alcuni di questi formati sono **ambigui** (ad esempio, 5/6/2020 è interpretato come 5 giugno 2020 in formato DMY o come 6 maggio 2020 in formato MDY), quindi quando vengono utilizzati richiedono di specificare il parametro `DateStyle` per dichiarare il modo in cui vogliamo che siano interpretate le nostre stringhe, ad esempio:

```
SET DateStyle TO MDY;
SET DateStyle TO ISO, DMY;
```

Il formato standard non ambiguo di PostgreSQL per le date è `YYYY-MM-DD` e questo è anche il modo in cui il sistema rappresenta internamente qualunque data.

[1] https://www.postgresql.org/docs/12/datatype-datetime.html

Esempio

```
CREATE TABLE manytimes(
dateval DATE
);

INSERT INTO manytimes(dateval) VALUES ('1999-01-08'), ('January 8,
    1999'), ('19990108'), ('1/8/1999');

SELECT * FROM manytimes;
```

L'output della `SELECT` è:

```
dateval
----------
1999-01-08
1999-01-08
1999-01-08
1999-01-08
```

2.7 Time

Per quanto specificare i tipi `TIME` ci sono due opzioni:

- `TIME`, che è sinonimo della foram verbosa `TIME WITHOUT TIME ZONE`, in cui si specifica solo l'orario senza time zone
- `TIME WITH TIME ZONE`, in cui si specifica anche il fuso orario.

Se si specifica una time zone o una data per un tipo `TIME` semplice, non si incorre in un errore, però il sistema rappresenterà solo l'ora, ignorando sia data che fuso orario. Nella documentazione si trovano tutti i modi per esprimere una l'orario e le time zones.
Se invece si non si specifica il fuso orario per un tipo dichiarato `TIME WITH TIME ZONE`, allora si assume di trovarsi nella time zone indicata nel paramentro `TimeZone` e viene convertita in UTC usando l'offset adeguato.

2.8 Time stamp

Valori accettati per il tipo `TIMESTAMP` sono concatenazioni di data e ora, seguiti eventualmente da una time zone e/o da AD o BC.

Esempio

Tutti i seguenti input sono accettati per un tipo `TIME STAMP`:

```
1999-01-08 04:05:06
1999-01-08 04:05:06 -8:00
January 8 04:05:06 1999 PST
```

I primi due seguono lo standard ISO 8601.
Nel secondo caso, si noti che la time zone viene introdotta da segno '-' (o '+'), e l'offset rispetto al meridiano di Greenwich.

Se il tipo è specificato `TIMESTAMP WITHOUT TIME ZONE`, PostgreSQL ignora implicitamente qualunque indicazione di una time zone, senza sollevare alcun errore.

Input Per il tipo `TIMESTAMP WITH TIME ZONE`, il sistema memorizza internamente il valore sempre in formato UTC (*Universal Coordinated Time*, conosciuto tradizionalmente come GMT, *Greenwich Mean Time*).
Una stringa in cui viene specificata esplicitamente la time zone viene conversita in UTC utilizzando un offset appropriato per quella time zone.

Output Quando invece si deve fornire un `TIMESTAMP WITH TIME ZONE` in output, questa viene mostrata secondo il tempo locale rispetto alla time zone specificata.

2.9 Il non-valore NULL: approfondimento

Il valore `NULL` rappresenta un valore (o meglio, un non-valore) speciale in SQL, che ha bisogno di essere trattato a parte.
La presenza di `NULL` in corrispondenza di un attributo indica uno stato in cui non si conosce il valore effettivo andrebbe collocato in corrispondenza di quell'attributo. A differenza di altri linugaggi come C e Java, **in SQL non è possibile usare gli operatori standard di confronto per testare la non valorizzazione di un certo attributo**.
Con operatori standard di confronto intendiamo:

`<, >, <=, >=, =, <>, !=`

In aggiunta a questi operatori standard, esistono due costrutti che permettono di scrivere in maniera più sintetica la condizione di presenza in un *range* di valori (`a BETWEEN x AND y`) e la condizione di presenza in un *insieme* di valori (`a IN (x, y, z, ... )`).
Quando `NULL` viene confrontato con altri valori tramite questi operatori, il risultato del confronto sarà sempre `NULL`. Questo vale anche nel caso in cui si effettui il confronto `NULL = NULL`.

Per sottolineare il concetto, illustriamo di seguito una serie di confronti che danno come risultato `NULL`:

```
SELECT 5 = NULL;

SELECT NULL = NULL;

SELECT NULL > 5;
```

Tutte queste `SELECT` danno come risultato `NULL`.
Se si tenta di effettuare una selezione con la sintassi `attributo = NULL`, nessuna riga verrà selezionata nel risultato.

Esempio

La selezione non produce alcuna riga:

```
CREATE TABLE test(
x INTEGER
);

INSERT INTO test VALUES(5), (NULL);

SELECT x FROM test WHERE x = NULL;
```

Per testare se un certo attributo non è stato valorizzato, esiste un operatore apposito: `IS`.

```
SELECT 5 IS NULL
```

Risultato: `false`.

```
SELECT NULL IS NULL
```

Risultato: `true`.

```
SELECT x IS NULL FROM test WHERE x = 5
```

Risultato: `false`.

È ammessa anche la sintassi `IS NOT NULL` per verificare, al contrario, se un attributo è effettivamente stato valorizzato.

Capitolo 3

Data Manipulation Language (DML)

Il *Data Manipulation Language* è la parte di SQL che fornisce i costrutti per l'aggiornamento, l'inserimento e la cancellazione dei dati.
Importante sottolineare la differenza tra *data definition* e *data manipulation*: il primo consiste nella creazione delle *strutture* della base di dati, e per farlo si serve di keyword quali `CREATE, DROP, ALTER` etc.
Il secondo invece riguarda la manipolazione dei *dati* stessi, quindi agisce sui valori memorizzati ma non sulle strutture. Le keyword utilizzate nelle operazioni di manipolazione sono `INSERT, DELETE, UPDATE`.

3.1 Inserimento

Al momento della sua creazione, una tabella è vuota, non contiene dati.
L'operazione di *inserimento*, implementata con il comando `INSERT`, permette di inserire nella tabella una riga alla volta; la sua sintassi è:

```
INSERT INTO myTable(col_1, col_2, ..., col_n)
VALUES(val_1, val_2, ..., val_n);
```

Il significato è: inserisci nella tabella `myTable` i valori `val_1, val_2, ..., val_n` in corrispondenza delle colonne `col_1, col_2, ..., col_n`.
È ammessa anche una sintassi più sintetica, che permette di omettere i nome delle colonne della tabella:

```
INSERT INTO myTable VALUES(val_1, val_2, ..., val_n);
```

Se si sceglie di inserire i dati in questo modo, bisogna accertarsi che l'ordine in cui si specificano i valori `val_1, val_2, ..., val_n` rispetti l'ordine delle colonne: questa caratteristica rende la notazione meno "robusta" della precedente, perché espone al rischio di inserire i dati nell'ordine sbagliato! È considerata quindi buona pratica utilizzare sempre la sintassi estesa, anche per

motivi di maggiore leggibilità.
È possibile anche inserire molteplici righe con un singolo comando `INSERT`:

```
INSERT INTO myTable(col_1, col_2, ..., col_n)
VALUES(x_1, x_2, ..., x_n), (y_1, y_2, ..., y_n), (z_1, z_2, ..., z_n) ...;
```

Sono ammessi inoltre altri due tipi di inserimento:

```
INSERT INTO myTable DEFAULT VALUES;
```

oppure posso inserire nella tabella le righe che costituiscono il risultato di una query:

```
INSERT INTO table_1 (a, b, c)
SELECT a, b, c FROM table_2
WHERE <condition>;
```

Ovviamente, il vincolo per poter eseguire questo inserimento è che le due tabelle, `table_1` e `table_2`, abbiano almeno un sottoinsieme di colonne in comune.
In questo modo, inserisco tutte le righe che ho selezionato con la query dalla tabella `table_2` nella tabella `table_1`. Con questo comando si inseriscono, laddove specificati, i valori di default, altrimenti viene inserito `null`.

3.2 Cancellazione

La rimozione di un sottoinsieme di righe di una tabella è possibile mediante il comando

```
DELETE FROM <table_name> [ WHERE <condition> ]
```

Omettere la condizione non è sintatticamente sbagliato e porta all'eliminazione di tutte le righe dalla tabella. Se questo è l'effetto voluto, potrebbe avere più senso usare un `DROP TABLE`, cioè eliminare la tabella stessa dalla base di dati.

3.3 Aggiornamento

La modifica dei dati presenti in un database è detta `UPDATE` (aggiornamento).
L'esecuzione di un `UPDATE` richiede almeno tre informazioni:

- il *nome della tabella* in cui si trova la colonna da modificare
- il *nome della colonna* da modificare
- il *nuovo valore* da inserire

Eventualmente, si può specificare anche una condizione da soddisfare, che limita il sottoinsieme di righe che saranno modificate.
La sintassi per effettuare un aggiornamento è:

```
UPDATE <table_name> SET <column_name> = <new_value> [ WHERE <condition> ]
```

Capitolo 4

Data Query Language (DQL)

Il comando che permette di recuperare i dati da una base di dati è chiamato *query* (o *interrogazione*).
In SQL, esiste un solo comando per interrogare una base di dati: `SELECT`. La sintassi di una interrogazione generica è:

```
SELECT [DISTINCT] [ * | expression [AS] output_name [, ...]]
[ FROM from_item [, ...] ]
[ WHERE condition ]
[ GROUP BY grouping_attribute [, ...] ]
[ HAVING condition [, ...] ]
[ { UNION | INTERSECT | EXCEPT } [ DISTINCT ] other_select ]
[ ORDER BY expression [ ASC | DESC | USING operator]]
```

dove:

- `expression` è un attributo
- `from_item` determina la sorgente degli attributi
- `condition` è un'espressione booleana che deve essere soddisfatta dalle righe che faranno parte del risultato.
- `grouping_attribute` è un'espressione che permette di "raggruppare" tutte le righe che hanno lo stesso valore per `attribute`, solitamente per poi performare determinate operazioni su queste righe e mostrare solo la riga di risultato.

Le clausole racchiuse tra parentesi quadre, [...], sono da considerarsi opzionali.
L'esecuzione di una `SELECT` produce una relazione risultato che ha come schema tutti gli attributi elencati nella clausola `SELECT`, ha come contenuto tutte le tuple t ottenute proiettando sugli attributi dopo ha come contenuto tutte le tuple t ottenute proiettando sugli attributi dopo SELECT, le tuple t' appartenenti al prodotto cartesiano delle tabelle ottenute dopo il `FROM` che soddisfano l'eventuale condizione nella clausola `WHERE/HAVING/GROUP BY`.

4.1 SELECT

La sintassi per specificare una `SELECT` è la seguente:

```
SELECT [DISTINCT] [{* | expression [AS output_name]} [, ...]]
```

dove:

- * è una abbreviazione che serve a indicare tutti gli attributi della tabella sorgente
- `expression` è un'espressione che coinvolge gli attributi della tabella (può anche essere semplicemente il nome di un attributo)
- `output_name` è il nome assegnato all'attributo che conterrà il risultato della valutazione di `expression` nella relazione risultato
- `DISTINCT` è una keyword che richiede l'eliminazione di tutti i duplicati di una tupla

4.2 FROM

Nella clausola `FROM` si specificano le tabelle in cui si devono ricercare le righe che andranno a far parte del risultato.
Nel caso in cui vengano specificate più tabelle separate da una virgola, la tabella in cui si effettua la ricerca è quella che risulta dal *prodotto cartesiano* delle tabelle specificate.
Quindi la seguente sintassi:

```
FROM table_1, table_2, ... table_n
```

equivale a: `table_1` $\times$ `table_1` $\times$ ... $\times$ `table_n`.
La tabella sorgente può anche essere quella risultante da una subquery:

```
FROM ( SELECT ... FROM ...) AS nomeRisultato
```

Per approfondimenti sulle *interrogazioni nidificate*, si vada alla sezione 4.8.
Tuttavia, nella maggior parte dei casi, le n tabelle specificate nel `FROM` vengono "legate" tra loro tramite gli operatori di `JOIN`:

```
FROM table_1
JOIN table_2 [ON join_condition]
JOIN ...
```

Il `JOIN` permette di selezionare solo un certo sottoinsieme del prodotto cartesiano di n tabelle; vista la grande frequenza del suo utilizzo, merita una trattazione più approfondita.

4.2.1 Operatori di JOIN

Esistono diversi modi per effettuare il `JOIN` tra più tabelle.
Il primo operatore di questo tipo ad essere dichiarato nello standard è stato il `CROSS JOIN`, che equivale a un prodotto cartesiano; a partire da SQL-2 sono stati introdotti altri tipi di `JOIN`: `INNER JOIN`, `LEFT OUTER JOIN`, `RIGHT OUTER JOIN` e `FULL OUTER JOIN`.

La clausola `NATURAL JOIN` ha la semantica del join naturale dell'algebra relazionale, dunque genera una sorgente di dati che include le righe delle tabelle che presentano lo stesso valore per l'attributo in comune.

La clausola `[INNER] JOIN` corrisponde al θ-join dell'algebra relazionale, ovvero un operatore che effettua il `JOIN` di due tabelle sulla base di una condizione, espressa dopo la keyword `ON`:

```
SELECT a, b, x, y
FROM table1 t1
[INNER] JOIN table2 t2 ON t1.a = t2.x
```

Si noti che quando si esegue un `INNER JOIN`, alle tabelle viene solitamente dato un nome temporaneo. Nel nostro caso, `table1` diventa `t1` e `table2` diventa `t2`. Pur non essendo obbligatoria, questa notazione è spesso è utile per distinguere attributi con lo stesso nome che appartengono a tabelle diverse. Si deduce quindi che la clausola `INNER JOIN` non può essere usata per eseguire il join naturale.

La clausola `t1 LEFT [OUTER] JOIN t2` esegue prima un normale `INNER JOIN` su condizione. Dopodiché, *ogni riga di t1 che non soddisfa la condizione di join viene aggiunta al risultato*, inserendo `NULL` negli attributi della riga risultato che appartengono a `t2`. Notare che nel `LEFT OUTER JOIN` si aggiungono le righe della tabella specificata a *sinistra*.

La clausola `t1 RIGHT [OUTER] JOIN t2` è simmetrica al `LEFT JOIN`: al risultato dell'`INNER JOIN` si aggiungono tutte le righe della tabella specificata a *destra* del join che non rispettano la condizione, inserendo `NULL` in tutti gli attributi della riga risultato che appartengono alla tabella specificata a sinistra.

La clausola `FULL [OUTER] JOIN` è equivalente a performare `INNER JOIN + LEFT JOIN + RIGHT JOIN`. **Non** è equivalente a fare `CROSS JOIN`!

4.3 WHERE

La clausola WHERE esprime una condizione, specificata come espressione booleana, che le righe della tabella sorgente devono rispettare per essere inserite nel risultato.
La sintassi è la seguente:

```
WHERE condition
```

Se le colonne coinvolte nella condizione contengono valori numerici, si possono utilizzare tutti gli operatori di confronto conosciuti.
Se le colonne contengono invece caratteri o stringhe, è possibile utilizzare anche gli operatori `LIKE` e `SIMILAR TO`.

Attenzione! WHERE e operatori di aggregazione

Nel clausola `WHERE` non è ammesso l'utilizzo di operatori di aggregazione.
Per esempio, una query di questo tipo:

```
SELECT DISTINCT id_insegn
FROM inserogato
WHERE COUNT(DISTINCT annoaccademico) > 2;
```

verrebbe rifiutata con il seguente errore:

```
ERROR: aggregate functions are not allowed in WHERE
LINE 3: WHERE COUNT(DISTINCT annoaccademico) > 2
```

Il motivo è che la clausola WHERE è pensata per eseguire selezioni su *righe*, non su gruppi, quindi non ha senso imporre delle condizioni di selezioni che richiedono di fare operazioni su insiemi di righe.
Se c'è l'esigenza di selezionare solo certi insiemi di righe in base a una condizione data da un operatore di aggregazione, si può utilizzare la clausola `HAVING`, che verrà affrontata nella sezione 4.6.

4.3.1 LIKE e SIMILAR TO

PostgreSQL permette di effettuare verifiche di *pattern matching* su stringhe, il che vuol dire testare se una stringa presenta un cert "schema". Lo schema si vuole che sia rispettato viene specificato utilizzando uno strumento chiamato *espressioni regolari*.
In PostgreSQL esistono tre modi diversi fare test di *pattern matching*:

- il tradizionale operatore `[NOT] LIKE`
- il più recente operatore `[NOT] SIMILAR TO`

- le espressioni regolari in stile POSIX

Di seguito forniamo uno schema in cui si riassumono le varianti e i sinonimi dell'operatore `LIKE`:

Operatore	Sinonimo	Variazione di significato
`LIKE`	`~~`	-
`NOT LIKE`	`!~~`	è la negazione di `LIKE`
`ILIKE`	`~~*`	è la versione *case insensitive*
`NOT ILIKE`	`!~~*`	è la negazione di `ILIKE`

La sua sintassi di questo operatore è:

```
WHERE attributo [NOT] {LIKE | ILIKE} 'regexp';
```

La stringa contenuta in `attribute` viene confrontata con il *pattern* specificato nella stringa `'regexp'`, che rappresenta un'espressione regolare. L'operatore `LIKE` non impone quindi l'uguaglianza tra la stringa e `'regexp'`, ma richiede che la stringa rispetti il pattern specificato. Per la sintassi delle espressioni regolari di PostgreSQL si rimanda alla documentazione ufficiale.

La keyword `SIMILAR TO` opera in modo simile a LIKE, ma il pattern specificato in `'regexp'` viene interpretato secondo la definizione standard di SQL (**POSIX**) per le espressioni regolari, che sono sostanzialmente una via di mezzo tra la notazione accettata da `LIKE` e la notazione standard delle espressioni regolari. In ultima analisi `SIMILAR TO` risulta quindi essere più espressivo di `LIKE`, perché è possibile esprimere un sottoinsieme maggiore di espressioni regolari.

4.4 ORDER BY

La clausola `ORDER BY`, posta subito dopo il `WHERE`, permette di ordinare le tuple del risultato rispetto a uno o più attributi.

```
WHERE ...
ORDER BY attribute [{ASC | DESC}] [, ...]
```

Le keyword `ASC, DESC` specificano l'ordine crescente (default) o descrescente.
Se vengono specificati più attributo, ad esempio:

```
ORDER BY nome, cognome, matricola
```

Si ordina prima rispetto al nome, che è il primo attributo specificato. A parità di nome, si ordina rispetto al cognome, e a parità di nome e cognome si ordina per matricola.
È ammesso usare nella clausola `ORDER BY` anche espressioni ed è uno dei pochi casi in cui è possibile specificare a una colonna con il nome con cui è stata rinominata nella `SELECT`. Ad esempio, la seguente sintassi **è legale**:

```
SELECT a + b AS sum, c, d
FROM table
ORDER BY sum   -- corretto!
```

Ma non è concesso utilizzare il nuovo nome in una espressione!
Infatti, la seguente sintassi **non è legale**:

```
SELECT a + b AS sum, c, d
FROM table
ORDER BY sum + c  -- errore!
```

Infine, l'`ORDER BY` è utilizzabile anche per ordinare il risultato di una `UNION, INTERSECT` o `EXCEPT`, ma in questo caso si possono specificare solo nomi di colonne, non espressioni. Infatti la seguente query dà errore:

```
(SELECT cs.nome, cs.durataanni AS anni, i.nomeins
FROM corsostudi cs
    JOIN inserogato ie ON ie.id_corsostudi = cs.id
    JOIN insegn i ON ie.id_insegn = i.id
WHERE ie.annoaccademico = '2010/2011'

EXCEPT

SELECT cs.nome, cs.durataanni AS anni, i.nomeins
FROM corsostudi cs
    JOIN inserogato ie ON ie.id_corsostudi = cs.id
    JOIN insegn i ON ie.id_insegn = i.id
WHERE ie.annoaccademico = '2010/2011' AND
ie.hamoduli = '1')
ORDER BY anni + 2;    -- errore: uso il nome di una colonna rinominata in
                      -- un'espressione

ERROR: invalid UNION/INTERSECT/EXCEPT ORDER BY clause
LINE 15: order by anni + 2;
^
DETAIL: Only result column names can be used, not expressions or functions.
```

4.5 GROUP BY

La clausola `GROUP BY`, posta dopo il `WHERE`, ha l'effetto di raggruppare tra loro tutte le righe della tabella risultato che presentano lo stesso valore per la colonna specificata:

```
WHERE ...
GROUP BY column_name
```

Se la clausola `SELECT` non contiene operatori di aggregazione, allora nel `GROUP BY` deve menzionare *tutti* gli attributi che compaiono nella `SELECT`. Il motivo è che un raggruppamento ha lo scopo di "mettere insieme" tutte le righe che presentano un certo valore per determinati attributi, poi nella tabella risultante si mostra *solo* una riga "rappresentante" di quel gruppo. Ma se esistono colonne che non sono incluse nel raggruppamento, e che quindi potrebbero avere valori distinti, com'è possibile scegliere una riga rappresentante? Che valori dovrei mostrare nelle colonne degli attributi non coinvolti nel raggruppamento? Da qui, la necessità di menzionare nella clausola `GROUP BY` tutti gli attributi che verranno mostrati nel risultato.

Se invece la clausola `SELECT` contiene attributi semplici insieme a operatori di aggregazione, allora la clausola `GROUP BY` è *obbligatoria* e deve menzionare tutti gli attributi che non sono coinvolti nell'operatore di aggregazione (vedi esempio nella sezione 4.7).
La clausola `GROUP BY` può essere usata in modo da produrre un effetto simile al `DISTINCT` eliminando quindi la ridondanza del risultato.

Esempio

Supponiamo che `table` sia una tabella contenente un campo `x` di tipo `CHAR` e un campo `y` di tipo `INTEGER`:

```
SELECT *
FROM table;
```

produce:

```
x | y
------
a | 3
c | 2
b | 5
a | 1
```

mentre il seguente comando:

```
SELECT x
```

```
FROM table
GROUP BY x;
```

produce un output senza ridondanze rispetto all'attributo x:

```
x
---
a
c
b
```

4.6 HAVING

La clausola `HAVING` permette di selezionare i raggruppamenti formati da `GROUP BY` in base a un certo predicato.

```
GROUP BY ...
HAVING bool_exp
```

Dato che `HAVING` lavora sui raggruppamenti, ovviamente è *obbligatorio* avere una clausola `GROUP BY` prima di `HAVING`.
Osservazione: `HAVING` ha sui gruppi lo stesso effetto che `WHERE` ha sulle righe!

Esempio particolare

Con il seguente comando si vogliono visualizzare le città i cui studenti hanno una media aritmetica dei voti superiore a 23.47:

```
SELECT citta, AVG(media)::DECIMAL(5,2) AS "roundAvg"
FROM Studente
GROUP BY citta
HAVING AVG(media) > 23.47;
```

In output abbiamo la seguente tabella:

citta	roundAvg
Padova	28.67
Vicenza	25.39
Verona	23.47

Notiamo subito che c'è una riga che non ci aspettavamo, ovvero la terza: con la condizione che abbiamo imposto nella clausola `HAVING` vorremmo vedere solo le città la cui media è *strettamente maggiore di 23.47*, quindi 23.47 escluso.
Il motivo di questo strano risultato sta nel cast effettuato nella `SELECT`: avendo specificato `AVG(media)::DECIMAL(5,2)`, chiediamo si *visualizzare* solo due cifre decimali, ma il valore memorizzato in tabella rimane invariato! Il fatto che venga mostrata anche la riga di Verona ci suggerisce che il valore di media effettivamente memorizzato è leggermente maggiore di 23.47, per esempio 23.473, sufficiente a renderlo strettamente maggiore di 23.47. Anche se nel risultato vengono mostrate solo due cifre decimali che potrebbero risultare ingannevoli, sappiamo che anche la terza riga rispetta la condizione.

4.7 Operatori di aggregazione

Gli *operatori di aggregazione* sono degli operatori che permettono di calcolare *un solo* valore a partire dai valori ottenuti con una `SELECT`.
Più precisamente, questi operatori considerano i valori contenuti in una colonna, eseguono un qualche tipo di operazione (che dipende dall'operatore) e restituiscono *una sola* riga in cui mostrano il risultato dei calcoli eseguiti.
In questo corso affronteremo con più attenzione i seguenti operatori:

- `COUNT`
- operatori aritmetici: `SUM, MIN, MAX, AVG`
- operatori booleani: `BOOL_AND, BOOL_OR`

A eccezione di `COUNT`, questi operatori ritornano `NULL` quando non viene selezionata alcuna riga su cui performare il calcolo: ad esempio, la somma `SUM` di nessuna riga ritorna `NULL`, non zero come ci si aspetterebbe.
È fondamentale ricordarsi utilizzare gli operatori di aggregazione insieme alla clausola `GROUP`

BY se nella SELECT compaiono anche colonne che non sono coinvolte nell'operazione di aggregazione, perché il risultato deve essere sempre *una sola riga!*

Esempio

La seguente query

```
SELECT cs.nome, COUNT(i.nomeins)
FROM corsostudi cs
    JOIN inserogato ie ON ie.id_corsostudi = cs.id
    JOIN insegn i ON ie.id_insegn = i.id
WHERE ie.annoaccademico = '2010/2011';
```

restituisce questo errore:

```
ERROR: column "cs.nome" must appear in the GROUP BY clause or be used in an
    aggregate function
LINE 1: select cs.nome, count(i.nomeins)
```

L'errore spiega che, dal momento che cs.nome non è coinvolto nell'operazione di COUNT, bisogna effettuare un raggruppamento rispetto ad esso.
La query corretta è:

```
SELECT cs.nome, COUNT(i.nomeins)
FROM corsostudi cs
    JOIN inserogato ie ON ie.id_corsostudi = cs.id
    JOIN insegn i ON ie.id_insegn = i.id
WHERE ie.annoaccademico = '2010/2011'
GROUP BY cs.nome;     -- !!
```

In questo modo vengono formati dei gruppi: tutte le righe che appartengono a un certo gruppo presentano lo stesso per l'attributo cs.nome. Dopodiché, viene effettuata l'operazione di COUNT sulle righe appartenenti allo stesso gruppo e il risultato sarà una riga sola contenente il nome del corso di studi e il risultato di COUNT. Ovviamente nel risultato complessivo avremo tante righe quanti sono i valori distinti di cs.nome.

4.7.1 COUNT

Questo operatore resituisce il numero di tuple *significative* nel risultato di una interrogazione. Con *"significative"* si intende quelle tuple che sono effettivamente valorizzate, ovvero non contengono NULL.
La sintassi è:

```
COUNT({ * | expr | ALL expr | DISTINCT expr})
```

dove `expr` è un'espressione che usa attributi e funzioni di attributi ma **non operatori di aggregazione**. Nel dettagli:

- `COUNT(*)` ritorna il numero di tuple totali del risultato dell'interrogazione, contando anche quelle che contengono valori nulli.
- `COUNT(expr)` ritorna il numero di tuple che soddisfano una certa condizione.[1].
- `COUNT(ALL expr)` ha lo stesso significato di `COUNT(expr)`
- `COUNT(DISTINCT expr)` è come `COUNT(expr)`, ma con l'ulteriore condizione che si contano solo i valori *distinti* di `expr`.

4.7.2 SUM, MIN, MAX, AVG

Questi sono tutti operatori aritmetici che determinano un valore numerico (`SUM/AVG`) o alfanumerico (`MAX/MIN`)[2].
La sintassi di questi operatori è:

```
{SUM | MAX | MIN | AVG}({expr | DISTINCT expr})
```

Il significato di aggiugnere la keyword `DISTINCT` all'interno delle parentesi ha lo stesso significato che per `COUNT`.

4.7.3 BOOL_AND, BOOL_OR

Gli operatori `BOOL_AND` e `BOOL_OR` sono detti operatori booleani perché eseguono un'operazione logica sull'insieme di righe e restituiscono un valore di verità, `true/false`.

```
{BOOL_AND | BOOL_OR} (expr)
```

Come si intuisce facilmente dal nome, `BOOL_AND` verifica se tutte le righe del risultato soddisfano l'espressione specificata, mentre `BOOL_OR` verifica che almeno una delle righe soddisfi l'espressione.

4.8 Interrogazioni nidificate

Un'interrogazione si dice *nidificata* o *innestata* quando è presente all'interno di un'altra interrogazione. La query innestata può comparire nel `FROM`, nel `WHERE` o nel `SELECT`.

[1] spesso, `expr` è semplicemente il nome di una colonna. In questo caso, si contano tutti i valori *significativi* della colonna, ovvero tutti i valori non nulli

[2] `MAX` e `MIN` possono essere usati anche su stringhe!

Esempio

Seleziono i titoli delle mostre in cui il prezzo è massimo tra tutte le mostre della tabella.

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

Con le query nidificate è possibile effettuare il confronto tra un attributo *un valore singolo* e il risultato di una query (*possibile insieme di valori*). Per effettuare questi confronti, **non è possibile utilizzare gli operatori di confronto stardard**, ovvero: $<, <=, >, >=, <>, =$. Invece, sono stati creati degli operatori appositi, tra cui:

- `[NOT] EXISTS`
- `[NOT] IN`
- `ALL`
- `ANY/SOME`

4.8.1 EXISTS

La sintassi di questo operatore è:

```
EXISTS (subquery)
```

L'argomento di `EXISTS` è un qualunque comando `SELECT`. La subquery viene valutata per determinare se produce delle righe: se viene prodotta *almeno una riga*, il risultato dell'`EXISTS` sarà `true`; altrimenti, sarà `false`.

L'utilizzo della clausola `EXISTS` ha senso solo se all'interno della subquery si selezionano le righe usando qualche valore della riga corrente della `SELECT` principale, ovvero se c'è *data binding*.

Problema: data binding

Il *data binding* rende l'esecuzione dell'intera query poco efficiente! Questo perché, dato che il risultato della subquery dipende dai valori della riga corrente nella `SELECT` principale, è necessario ri-eseguire la subquery per ogni riga analizzata dalla query principale. L'inefficienza si manifesta soprattutto quando la subquery non produce alcuna riga, perché si è costretti a fare il test su tutte le righe della tabella su cui la subquery lavora. Se invece viene prodotta almeno una riga, l'esecuzione si ferma alla prima riga trovata (dalla doc: *"The subquery will generally only be executed long enough to determine whether at*

least one row is returned, not all the way to completion").

Dal momento che il risultato della query dipende solo dal fatto che vengano prodotte delle righe o meno, e non dal contenuto delle righe prodotte, l'output della subquery è irrilevante: è convenzione scrivere i test di esistenza nella forma `EXISTS(SELECT 1 ... )`.

Esempio

Si vogliono visualizzare i nomi dei corsi di studio che nel 2006/2007 hanno erogato insegnamenti il cui nome contiene la sottostringa 'Info':

```
SELECT CS.nome
FROM CorsoStudi CS
WHERE EXISTS (
    SELECT 1
    FROM InsErogato IE
        JOIN Insegn I ON IE.id_insegn = I.id
    WHERE I.nomeins LIKE '%Info%' AND
        IE.annoaccademico = '2006/2007' AND
        IE.id_corsostudi = CS.id);
```

Esistono modi più efficienti per ottenere lo stesso risultato di `EXISTS` con altri operatori.

4.8.2 IN

Ci sono due possibili sintassi per questo operatore: nel caso in cui si voglia confrontare il valore di una `expression` (che può essere anche un singolo attributo) con un insieme di valori prodotti da una `subquery`:

```
expression [NOT] IN (subquery)
```

Osservazione: se l'espressione a sinistra di `IN` è `null`, oppure se nell'insieme dei valori generati dalla subquery non ce n'è alcuno che combacia con i valori dell'espressione e almeno uno di questi è `NULL`, allora il risultato di `IN` non sarà `false`, bensì `null`. Questa è in accordo con il modo in cui vengono valutate le espressioni booleane in presenza di valori nulli in SQL, perché la semantica dell'operatore `IN` è (in pseudo-codice):

```
if(expression = val_1 OR expression = val_2 OR ... OR expression = val_n){
return true
}
```

Dove `val_1, val_2, ..., val_n` sono gli n valori prodotti dalla subquery.
Comunque, avere una clausola `WHERE null` oppure `WHERE false` produce lo stesso effetto: il risultato della query principale contiene 0 righe.

Nel caso invece in cui si voglia verificare la presenza di una riga nel risultato di una `subquery`:

```
ROW(expression, [, ...]) [NOT] IN (subquery)
```

Di nuovo, `expression` può essere un'espressione vera e propria oppure un attributo.
Il numero di colonne prodotte dalla subquery deve essere uguale al numero di attributi specificati nella `ROW`.
L'operatore ritorna `true` se la riga specificata è presente anche nell'insieme di righe prodotte dalla subquery (con "essere presente" si intende "avere gli stessi valori di un'altra riga").

Esempio

```
SELECT I.nome, I.cognome
FROM Impiegato I
WHERE ROW(I.nome, I.cognome) IN (
SELECT I1.nome, I1.cognome
FROM ImpiegatoAltraAzienda I1 );
```

Come per la clausola `EXISTS`, non sempre le subquery viene completamente valutata: l'esecuzione viene interrotta non appena viene trovato il primo match.

4.8.3 ANY/SOME

`ANY` e `SOME` sono sinonimi e possono essere usati intercambiabilmente. La sintassi è la seguente:

```
expression operator { ANY | SOME } (subquery)
```

dove `expression` è un'espressione che coinvolge gli attributi della `SELECT` principale, `(subquery)` è una `SELECT` che deve restituire *una sola colonna* (!), `operator` $\in <, <=, >, >=, <>, =$, quindi è un operatore di confronto.
Il risultato sarà `true` se e solo se `expression` è `operator` rispetto al valore di una qualsiasi riga del risultato di `(subquery)`.
Altrimenti, il risultato è `false`, anche nel caso in cui la subquery restituisca 0 righe.
Osservazione: l'operatore `IN` è semanticamente equivalente a `= ANY`.
Il valore `null` viene prodotto nelle stesse circostanze specificate per `IN`, ovvero quando non viene trovato alcun match e *almeno un* valore della subquery è `null`.

4.8.4 ALL

La sintassi dell'operatore `ALL` è indentica a quella degli operatori `ANY` e `SOME`:

```
expression operator ALL (subquery)
```

oppure, nel caso si voglia effettuare il confronto tra righe piuttosto che tra valori singoli:

```
ROW(expression, [,...]) operator ALL (subquery)
```

In cui `subquery` deve restituire un insieme di righe che ha gli stessi attributi specificati in `ROW(expression, [,...])`. Il risultato di `ALL` sarà:

- `true` se il risultato del confronto tra `expression` e le righe della subquery dà `true` per ogni riga della subquery
- `false` se esiste almeno un confronto che dà `false`.
- `null` se non esiste alcun confronto che restituisce `false` ed è presente almeno un confronto che restituisce null

Questo è in accordo con le regole per la valutazione di espressioni booleane di PostgreSQL in presenza di valori nulli.
Così come per gli altri operatori precedentemente illustrati, anche in questo caso la valutazione di `ALL` è *lazy*.
Osservazione: `NOT IN` è semanticamente equivalente a scrivere `<> ALL`

4.8.5 Operatori insiemistici

Esistono tre operatori specifici per effettuare operazioni insiemistiche sui risultati di due o più query, ovvero:

- `UNION`
- `INTERSECT`
- `EXCEPT`

La sintassi per questi operatori è:

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

Gli operatori si possono applicare solo quando `query1` e `query2` producono risultati con lo stesso numero di colonne e di tipo compatibile tra loro.
Tutti gli operatori eliminano dal risultato i duplicati a meno che non sia specificata la clausola `ALL`.

UNION Viene calcolata l'**unione** tra i due insiemi di righe generati da `query1` e `query2`: una riga compare nel risultato finale se appartiene a uno dei due insiemi (si ricorda che, da un punto di vista logico, l'operazione di unione corrisponde a un OR).
L'ordine in cui vengono specificate le query è irrilevante dal punto di vista del risultato finale.

Esempio

Si vogliono visualizzare i nomi degli insegnamenti e i nomi dei corsi di laurea che non iniziano per 'A' mantenendo i duplicati.

```
SELECT nomeins
FROM Insegn
WHERE NOT nomeins LIKE 'A%'

UNION ALL

SELECT nome
FROM CorsoStudi
WHERE NOT nome LIKE 'A%';
```

INTERSECT Viene calcolata l'**intersezione** tra i due insiemi di righe generati da `query1` e `query2`: una riga compare nel risultato finale se appartiene a entrambi gli insiemi (si ricorda che, da un punto di vista logico, l'operazione di unione corrisponde a un AND).
L'ordine in cui vengono specificate le query è irrilevante dal punto di vista del risultato finale.

Esempio

Si vogliono visualizzare i nomi degli insegnamenti che sono anche nomi di corsi di laurea.

```
SELECT nomeins
FROM Insegn

INTERSECT ALL

SELECT nome
FROM CorsoStudi;
```

EXCEPT Viene calcolata la **differenza** tra i due insiemi di righe generati da `query1` e `query2`. In questo caso, l'ordine in cui vengono specificate le query è rilevante, perché l'opera-

zione di differenza **non è commutativa**. Quindi:

```
query1 EXCEPT query2
```

restituisce tutte le righe che appartengono all'insieme generato da `query1` ma non a quello di `query2`.
Viceversa:

```
query2 EXCEPT query1
```

restituisce tutte le righe che appartengono all'insieme generato da `query2` ma non a quello di `query1`.

Esempio

Si vogliono visualizzare i nomi degli insegnamenti che NON sono anche nomi di corsi di laurea.

```
SELECT nomeins
FROM Insegn

EXCEPT

SELECT nome
FROM CorsoStudi;
```

Capitolo 5

Introduzione agli indici e analisi delle prestazioni in SQL

Gli indici sono delle strutture dati ausiliarie che permettono di accedere ai dati di una tabella in modo efficiente. Al momento della creazione di una tabella, il DBMS crea automaticamente degli indici sugli attributi dichiarati `PRIMARY KEY` e `UNIQUE`, perché sono tra quelli più utilizzati per l'esecuzione di due operazioni critiche: la ricerca di una tupla precisa e il join tra tabelle.
È importante tenere un indice aggiornato in modo da mantenere le prestazioni anche a fronte di svariate modifiche del contenuto della base di dati, ma il costo di mantenimento può diventare oneroso se sono presenti molti indici sulla medesima tabella. Inoltre, bisogna considerare che una struttura dati richiede sempre un certo spazio per essere memorizzata, che può essere anche non indifferente.
A fronte di queste considerazioni preliminari, si intuisce subito che è importante saper definire gli indici con criterio. Fortunatamente, esistono anche degli strumenti che ci permettono di analizzare dettagliatamente il piano di esecuzione di una certa query, per capire se gli indici vengono utilizzati e se risultano utili ai fini di un miglioramento delle prestazioni.
Gli indici possono contribuire ad ottimizzare le prestazioni di comandi di `SELECT`, `UPDATE` e `DELETE` (questi ultimi guadagnano efficienza se contengono la clausola `WHERE`).

Oltre agli indici sulle chiavi primarie creati automaticamente dal sistema, l'utente può definire esplicitamente altri indici. L'istruzione che consente di fare ciò è la seguente:

```
CREATE INDEX newIndex ON thisTable(attribute);
```

dove `newIndex` è il nome che diamo all'indice, e può essere scelto arbitrariamente, mentre `ON thisTable(attribute)` indica che stiamo creando un indice sull'attributo `attribute` della tabella `thisTable`. Questa istruzione, che non specifica alcun tipo particolare di indice, crea un indice B-tree.
Una volta creato l'indice, sarà il sistema a occuparsi del suo aggiornamento. Tuttavia, è op-

portuno lanciare il comando `ANALYZE` periodicamente per costringere il sistema ad aggiornare le statisiche relative alle tabelle e permettere così al *query planner* di prendere decisioni più accurate.

5.1 Tipi di indice

PostgreSQL fornisce svariati tipi di indice: B-tree, Hash, GiST, SP-GiST e BRIN, ma in questo corso ci limiteremo solo allo studio di B-tree e Hash.
Per specificare il tipo di indice che si vuole creare, si usa la seguente sintassi:

```
CREATE INDEX idxName ON thisTable USING <INDEX_TYPE>(attribute)
```

dove `INDEX_TYPE` rappresenta, appunto, il tipo di indice desiderato.
Ricordiamo che, **a meno di una specifica diversa, PostgreSQL crea di default indici B-tree**.

5.1.1 B-Tree

Gli indici B-tree sono adatti a gestire condizioni di uguaglianza e di range su dati in cui è possibile stabilire un ordine. Il *query planner* cercherà di usare, se presente, un indice B-tree nei seguenti casi:

- l'attributo indicizzato è coinvolto in un confronto che usa gli operatori $<, <=, =, >=, >$ o gli operatori complessi `BETWEEN` e `IN`.
- l'attributo è coinvolto in un test `IS [NOT] NULL`
- l'attributo è coinvolto in un test di *pattern matching*, quindi si utilizza l'operatore `LIKE` o `SIMILAR TO`, ma **solo se il pattern è una costante o se riguarda l'inizio di una stringa**

Riguardo al caso del *pattern matching*, occorre fare alcune precisazioni.
Prima di tutto, quando si vuole creare un indice da usare in un'operazione di questo tipo e il database non supporta lo standard C locale, è bene utilizzare l'operatore `varchar_pattern_ops`: in questo modo, i valori sono confrontati carattere per carattere e non secondo le regole imposte dal C locale.
Un *locale* è un insieme di regole che stabiliscono in che modo rappresentare certi dati, ad esempio data, ora, importi monetari, rappresentazione di numeri decimali, etc.
Riguardo alle regole alfabetiche, i *locale* dei vari paesi potrebbero usare convenzioni diverse, ad esempio per il tedesco le lettere maiuscole ([A..Z]) vengono prima delle lettere minuscole ([a..z]), mentre il contrario succede per l'italiano, e queste differenze diventano rilevanti quando si deve ricercare una stringa in un B-tree!

```
CREATE INDEX indexName ON thisTable (attribute varchar_pattern_ops);
```

Inoltre, abbiamo precisato che ha senso creare un indice per il confronto tra stringhe solo se la stringa da confrontare è una costante oppure se è un *pattern* che riguarda l'inizio della stringa. Questa limitazione è abbastanza ovvia, dato che per disporre un insieme di stringhe in ordine alfabetico si procede confrontano prima il primo carattere, su parità del primo carattere si confronta il secondo carattere, e così via; non conoscendo la parte iniziale delle stringhe, risulta evidente l'impossibilità di disporle in ordine alfabetico.

Quando si usano gli indici hash, l'operatore `varchar_pattern_ops` non è necessario, perché questo tipo di indice viene usato solo per fare confronti di uguaglianza, quindi le stringhe vengono in ogni caso confrontate carattere per carattere.

5.1.2 Hash Index

Gli hash index possono risultare utili per gestire semplici *condizioni di uguaglianza*. La sintassi per creare un indice hash è:

```
CREATE INDEX [<index_name>] ON <table_name> USING hash(<column_name>)
```

Questa sintassi viene utilizzata anche per definire qualunque indice che abbia un tipo diverso da B-tree.
Gli indici di tipo hash hanno un uso molto limitato in PostgreSQL perché possono essere usati solo per i confronti di uguaglianza. Inoltre, la gestione di questo tipo di indice è più complicata in una base di dati replicata o in caso di crash, quindi se ne sconsiglia l'uso.

5.1.3 Indici multiattributo

Un indice può essere definito anche su più colonne (fino a 32) di una stessa tabella. Questo pratica risulta conveniente quando questi attributi sono spesso coinvolti in **congiunzioni di confronti**, ad esempio:

```
SELECT I.nomeins , I.codiceins
FROM Insegn I
    JOIN InsErogato IE ON I.id = IE.id_insegn
WHERE IE.annoaccademico = '2006/2007' AND IE.id_corsostudi = 4;
```

Notiamo che la condizione di selezione consiste in due confronti congiunti in un `AND`; in questo caso è vantaggioso creare il seguente indice multiattributo:

```
CREATE INDEX idx_ie ON InsErogato(annoaccademico, id_corsostudi);
```

Questa istruzione genera un B-tree i cui nodi sono ordinati prima secondo l'attributo `annoaccademico`, poi secondo `id_corsostudi`. Deduciamo quindi che **nella definizione di indici**

multiattributo, l'ordine in cui vengono specificati gli attributi è rilevante!
Il piano di esecuzione della query dopo la creazione dell'indice multiattributo è:

```
Hash Join (cost=280.22..315.57 rows=31 width=46)
  Hash Cond: (ie.id_insegn = i.id)
  -> Index Scan using idx_ie on inserogato ie (cost=0.42..35.34 rows=31 width=4)
       Index Cond: (((annoaccademico)::text = '2006/2007'::text) AND
           (id_corsostudi = 4))
  -> Hash (cost=177.69..177.69 rows=8169 width=50)
       -> Seq Scan on insegn i (cost=0.00..177.69 rows=8169 width=50)
    (6 rows)
```

La terza riga descrive il primo nodo figlio del piano di esecuzione e si può notare chiaramento l'utilizzo dell'indice multiattributo da noi creato.
Se avessimo creato due indici separati su ciascun attributo di `InsErogato`, ad esempio:

```
CREATE INDEX idx_ie_aa ON InsErogato(annoaccademico);
CREATE INDEX idx_ie_cs ON InseErogato(id_corsostudi);
```

avremmo avuto un piano di esecuzione diverso:

```
Hash Join (cost=357.82..392.57 rows=31 width=46)
  Hash Cond: (ie.id_insegn = i.id)
  -> Bitmap Heap Scan on inserogato ie (cost=78.02..112.34 rows=31 width=4)
       Recheck Cond: ((id_corsostudi = 4) AND ((annoaccademico)::text =
           '2006/2007'::text))
       -> BitmapAnd (cost=78.02..78.02 rows=31 width=0)
            -> Bitmap Index Scan on idx_ie_cs (cost=0.00..6.49 rows=425 width=0)
                 Index Cond: (id_corsostudi = 4)
            -> Bitmap Index Scan on idx_ie_aa (cost=0.00..71.26 rows=4955
               width=0)
                 Index Cond: ((annoaccademico)::text = '2006/2007'::text)
  -> Hash (cost=177.69..177.69 rows=8169 width=50)
       -> Seq Scan on insegn i (cost=0.00..177.69 rows=8169 width=50)
(11 rows)
```

Si il primo nodo figlio dell'albero (sesta riga), chiamato `BitmapAnd`: la presenza di questo nodo sta a significare che nel piano di esecuzione si fa la congiunzione tra i risultati ottenuti da due sottonodi. Nel nostro caso, prima si selezionano i blocchi contenenti le righe con `id_corsostudi = 4` usando l'indice `idx_ie_cs`, dopodiché si selezionano i blocchi contenenti le righe con `annoaccademico = '2009/2010'` usando l'indice `idx_ie_aa`.
Una volta fatto ciò, il nodo `BitmapAnd` fa la congiunzione (l'operazione logica di `AND`) tra i risultati dei due nodi.

Non sempre però è possibile creare indici multiattributo per aumentare le prestazioni di esecuzione: quando la condizione di selezione consiste in una *disgiunzione* di condizioni, conviene creare indici distinti.

```
SELECT I.nomeins , I.codiceins
FROM Insegn I
    JOIN InsErogato IE ON I.id = IE.id_insegn
WHERE IE.annoaccademico = '2006/2007' OR IE.id_corsostudi = 4;
```

In questo caso, un indice multiattributo mi aiuterebbe a trovare solo le tuple che soddisfano `IE.annoaccademico = '2006/2007' AND IE.id_corsostudi = 4`, ma poi dovrei effettuare un altro tipo di ricerca per includere anche quelli che soddisfano solo `IE.annoaccademico = '2006/2007'` oppure `IE.id_corsostudi = 4`.
Come regola generale, quindi, **gli indici multiattributo vanno creati solo se la condizione di selezione riguarda una congiunzione di confronti che coinvolge attributi diversi.**

Un indice multiattributo di tipo B-tree è più efficiente quando le condizioni più stringenti riguardano i primi attributi specificati nell'indice. Ad esempio, se la query contiene una selezione del tipo:

```
WHERE a = 5 AND b >= 42 AND c < 77
```

nell'indice conviene specificare prima la colonna `a`:

```
CREATE INDEX idx ON myTable(a, b, c)
```

L'indice verrà scansionato a partire dalla prima tupla che ha `a = 5` e `b = 42`, fino all'ultima tupla con `a = 5`. Tutte le tuple con `c >= 77` non verranno inserite nel risultato, ma comunque verranno analizzate. (chiedere chiarimenti su questa parte)

5.2 Analisi di utilizzo degli indici

Abbiamo già spiegato che, se opportunamente definiti, gli indici possono aumentare di molto le prestazioni del DBMS, ma è importante definirli con criterio, in quanto occupano una quantità di memoria non indifferente e devono essere mantenuti aggiornati. PostgreSQL permette di analizzare il piano di esecuzione di una query grazie al comando `EXPLAIN query`, che ci permette anche di vedere quali indici sono effettivamente utilizzati durante la query e quali no, oppure quali operazioni potrebbero essere ottimizzate.
Il *piano di esecuzione* di una query è un albero di nodi di esecuzione, le cui foglie sono "nodi di scansione": l'esecuzione di questi nodi restituisce gli indirizzi delle righe di una tabella.
In PostgreSQL, l'output mostra una riga per ciascun nodo dell'albero dove si indica il tipo di operazione e una stima del costo di esecuzione. La prima riga dell'output rappresenta il nodo radice e contiene la stima del costo totale di esecuzione della query, ed è proprio questo il costo

che l'ottimizzatore tenta di minimizzare.

Esempio

Questa query

```
EXPLAIN SELECT IE.id
FROM InsErogato IE
    JOIN CorsoStudi CS ON IE.id_corsostudi = CS.id
WHERE IE.annoaccademico = '2013/2014' AND
    CS.nome = 'Laurea in Informatica';
```

produce il seguente piano di esecuzione:

```
Nested Loop (cost=0.00..6131.59 rows=8 width=4)
Join Filter: (ie.id_corsostudi = cs.id)
    -> Seq Scan on corsostudi cs (cost=0.00..96.94 rows=1 width=4)
Filter: ((nome)::text = 'Laurea in Informatica'::text)
    -> Seq Scan on inserogato ie (cost=0.00..5970.21 rows=5155 width=8)
Filter: ((annoaccademico)::text = '2013/2014'::text)
```

Il nodo radice ci dice che il join viene eseguito con la tecnica *Nested Loop* (per i dettagli, si vedano le dispense di teoria). Inoltre, sono presenti tre campi che ci forniscono ulteriori informazioni:

- `cost` rappresenta una *stima* del costo di esecuzione della query, in termini di accessi a memoria secondaria
- `rows` è il numero totale di righe che fanno parte del risultato
- `width` è una misura della dimensione di ogni riga in byte

Subito sotto al nodo radice troviamo la dicitura `Join Filter`, che ci dice su quali attributi viene fatto il join.
Tutti gli altri nodi (introdotti da una freccia `->`) presentano un certo grado di indentazione, che indica sostanziamente il livello di profondità a cui si trovano.
Il primo nodo che incontriamo è `Seq Scan on corsostudi cs`, ci dice che è stata fatta una scansione sequenziale della tabella `corsostudi` ed è stato applicato un certo filtro sull'attributo `nome`.
Più in basso c'è il nodo `Seq Scan on inserogato ie`, che dichiara una scansione sequenziale della tabella `inserogato` con un filtro su `annoaccademico`.

Un modo per ottimizzare la query è quello di rimuovere, quando possibile, le scansioni sequenziali. Nel caso specifico di questa query, un'idea potrebbe essere quella di creare due indici su `inserogato(annoaccademico)` e su `corsostudi(nome)`:

```
CREATE INDEX idx_ie_aa ON inserogato(annoaccademico);
CREATE INDEX idx_cs_nome ON corsostudi(nome);

EXPLAIN SELECT IE.id
FROM InsErogato IE
    JOIN CorsoStudi CS ON IE.id_corsostudi = CS.id
WHERE IE.annoaccademico = '2013/2014' AND
    CS.nome = 'Laurea in Informatica';
```

La prima riga del piano di esecuzione è:

```
Hash Join (cost=65.97..3641.10 rows=8 width=4)
...
```

Gli accessi sono quasi dimezzati, 3641 rispetto ai 6131 di prima!
Si noti che in questo caso non è possibile creare un indice multiattributo perchè gli attributi su cui si fa la selezione appartengono a tabelle diverse.
Nel piano di esecuzione sono presenti anche delle righe che ci confermano che gli indici sono stati utilizzati!

```
...
->  Bitmap Heap Scan on inserogato ie (cost=63.47..3619.19 rows=5155 width=8)
    Recheck Cond: ((annoaccademico)::text = '2013/2014'::text)
->  Bitmap Index Scan on idx_ie_aa (cost=0.00..62.18 rows=5155 width=0)
    Index Cond: ((annoaccademico)::text = '2013/2014'::text)
```

Analizziamo più nel dettaglio i nodi:

- `Bitmap Index Scan on idx_ie_aa` ci dice che è stata eseguita una scansione dell'indice `idx_ie_aa`, sulla condizione specificata appena sotto da `Index Cond`.
  Questa operazione non restituisce le righe, bensì gli **indirizzi** dei blocchi in cui si trovano tutte le righe che potrebbero potenzialemente far parte del risultato.

- `Bitmap Heap Scan on inserogato ie` ci dice che il "mucchio" (*heap*) di indirizzi restituiti dal nodo precedente viene scansionato, ricontrollando la condizione sull'anno accademico (`Recheck Cond: ...`)

Il nodo `Bitmap Heap Scan`, inoltre, ordina *fisicamente* gli indirizzi delle righe trovati prima di leggerli, per minimizzare il tempo di estrazione delle righe. La parola *"bitmap"* indica il meccanismo che esegue l'ordinamento. Perché viene fatto un *recheck* della condizione sull'anno accademico? Perché il nodo precedente resituisce solo gli indirizzi dei *blocchi* in cui si trovano le

righe che potrebbero far parte del risultato! All'interno di un blocco potrebbero trovarsi anche delle righe che non rispettano la condizione specificata, quindi una volta ottenuti i blocchi di interesse bisogna ri-analizzare le singole righe.

5.3 Criteri per creare indici

In generale, conviene sempre creare degli indici su:

- attributi di *join*
- attributi di *selezione* (presenti nella clausola `WHERE`)

Ricordiamo che, al momento della creazione di una tabella, il sistema crea automaticamente degli indici su tutti gli attributi dichiarati `PRIMARY KEY` o `UNIQUE`, quindi in questo caso non è necessario ricrearli.
I criteri per creare degli indici utili variano anche a seconda della tecnica di join utilizzata (che si può conoscere tramite il comando `EXPLAIN`).

Nested Loop Join Esegue il join A⋈B come due cicli `for` innestati, quello più esterno scorre tutte le righe di A mentre quello più interno scorre le righe di B.

Pseudocodice

Supponiamo che `table1` sia la tabella esterna e `table2` la tabella interna.
Supponiamo inoltre che il join sia fatta sia fatto sulla condizione `table1.a = table2.b`

```
for row in table1{
    for row in table2{
        if(table1.a = table2.b){
            add row to result
        }
    }
}
```

Questa tecnica di join è applicabile a qualunque situazione ed è la più semplice, ma non sempre è la più efficiente: la complessità computazionale di due cicli `for` innestati è molto alta, quindi per tabelle molto grandi l'esecuzione può essere molto lenta.
In questo caso, valgono le due regole generali di ottimizzazione dichiarate sopra.

Merge Scan Join È una tecnica che richiede di avere le tabelle ordinate secondo l'attributo di join; questo è si verifica se:

- le tabelle sono *fisicamente* ordinate
- esistono degli indici che permettono di accedere in modo ordinato alle righe delle tabelle

Di solito viene adottato quando nella query sono presenti le clausole `ORDER BY`, `DISTINCT` o `GROUP BY`[1].
Il sistema può decidere di applicare il Merge Join anche su tabelle inizialmente non ordinate, ma in questo caso un overhead per eseguire l'ordinamento fisico delle tabelle prima di effettuare il join.

Esempio: piano di esecuzione di un Merge Join

```
EXPLAIN SELECT *
FROM t1 , t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                      QUERY PLAN
-------------------------------------------------------------
Merge Join (cost=198.11..268.19 ROWS=10 width=488)
    Merge Cond: (t1.unique2 = t2.unique2)
    -> INDEX Scan USING t1_unique2 ON t1 (cost=0..656 ROWS=101..)
        Filter: (unique1 < 100)
    -> Sort (cost=197.83..200.33 ROWS=1000..)
        Sort KEY: t2.unique2
        -> Seq Scan ON t2 (cost=0.00..148.00 ROWS=1000..)
```

Notiamo che la tabella `t1` è già ordinata via indice: questo si può desumere dalla terza riga del piano di esecuzione, in cui si dice che viene fatta un `INDEX Scan` su `t1` usando l'indice `t1_unique2`. Questa scansione, quindi, non è ulteriormente ottimizzabile.

La tabella `t2`, invece, non era ordinata al momento dell'esecuzione della query: questo si può desumere dal fatto che l'altro nodo figlio (quinta riga) dichiara un nodo `Sort` sull'attributo `unique2` di `t2`. Possiamo ottimizzare la query creando un indice su `t2.unique` ed eliminando così il `Seq Scan` (ultima riga) necessario per eseguire l'ordinamento

Hash Based Join Può essere applicato nel caso di *equi-join* (che rappresentano peraltro un caso molto frequente: il join viene fatto testando una condizione di uguaglianza su attributi non comuni).
Viene creata una funzione hash sugli attributi di join, quindi la tabella esterna viene interamente

[1] Ricordiamo che per effettuare l'eliminazione dei duplicati e i raggruppamenti è necessario eseguire prima un ordinamento sulla tabella.

scansionata e le sue righe vengono inserite in una tabella hash. La stessa cosa viene fatta sulla tabella interna. Per verificare l'uguaglianza sugli attributi di join si verifica l'uguaglianza dei valori prodotti dalla funzione hash.

Esempio: piano di esecuzione di un Hash Join

```
EXPLAIN SELECT *
FROM Insegn I
    JOIN Inserogato IE ON ie.id_insegn=i.id
WHERE ie.annoaccademico='2013/2014';

                              QUERY PLAN
-----------------------------------------------------------------------
Hash Join (cost=287.80..6328.90 ROWS=5155 width=641)
    Hash Cond: (ie.id_insegn = i.id)
    -> Seq Scan ON inserogato ie (cost=0.00..5970.21 ROWS=5155
        w=578)
        Filter: ((annoaccademico)::TEXT = '2013/2014'::TEXT)
    -> Hash (cost=185.69..185.69 ROWS=8169 width=63)
        -> Seq Scan ON insegn i (cost=0.00..185.69 ROWS=8169 w=63)
```

Il nodo `Hash` prepara una hash table scansionando sequenzialmente la tabella `insegn i`. In questo caso, lo scan sequenziale non si può ottimizzare poiché dobbiamo calcolare il valore hash di tutte le righe della tabella. Eventualmente, se la clausola `WHERE` contiene una selezione sulle righe della tabella (in questo caso, sulle righe di `insegn`), si può creare un indice sull'attributo di selezione.

Il nodo `Hash Join` effettua il join vero e proprio: per ogni riga fornita dal primo figlio (`Seq Scan`), cerca nel secondo figlio (`Hash`) la riga da unire secondo la condizione di join.
Dato che c'è una condizione di selezione sulla tabella `inserogato`, possiamo ottimizzare lo scan sequenziale creando un indice sull'attributo di selezione (`annoaccademico`)

Capitolo 6

Introduzione al controllo della concorrenza

In questo capitolo descriveremo il comportamento di PostgreSQL quando due o più transazioni agiscono sui dati in modo concorrente. Sappiamo che tra le proprietà ACID di una transazione c'è anche l'**isolamento**, quindi l'esecuzione di una transazione deve essere rappresentare un "compartimento stagno", non deve essere influenzata dall'esecuzione di altre transazioni.
L'obiettivo del controllo della concorrenza è pertanto quello di permettere un accesso efficiente per tutte le transazioni attive garantendo allo stesso tempo l'integrità dei dati, evitando cioè il manifestarsi di determinate anomalie.
Le possibili anomalie che si possono manifestare sono:

- **lettura sporca** (*dirty read*): una transazione legge i dati scritti da un'altra transazione che non ha effettuato il `commit`
- **letture inconsistenti** (*non-repeatable read*): una transazione che ripete una lettura degli stessi dati in istanti diversi osserva dei cambiamenti negli stessi, dovuti a una transazione che li ha modificati e ha effettuato il `commit`
- **aggiornamento fantasma** (*phantom read*): una transazione esegue una query che restituisce un insieme di righe che dovrebbero soddisfare una certa condizione, ma questo insieme è cambiato poiché un'altra transazione nel frattempo ha modificato i dati relativi alla condizione.
- **mancata serializzazione** (*serialization anomaly*): il risultato di un gruppo di transazioni (nessun abort) non è compatibile con alcun ordine di esecuzione seriale delle stesse.

PostgreSQL riesce a mantenere la consistenza dei dati utilizzando il modello *Multiversion Concurrency Control, MVCC*, che risulta essere meno restrittivo ma ugualmente efficace rispetto al Locking a due fasi (2PL).

La differenza principale tra i due approcci è che nel modello MVCC i lock acquisiti per interrogare la base di dati non vanno in conflitto con i lock acquisiti per scrivere i dati. Di conseguenza, la lettura non blocca mai la scrittura, la scrittura non blocca mai la lettura, **ma una scrittura può bloccare un altra scrittura!**[1]
Il funzionamento del modello MVCC si basa sul concetto di snapshot:

> Ogni transazione vede uno *snapshot* dello stato della base di dati in un certo istante. Le letture su questa istantanea sono sempre possibili e non sono mai bloccate anche se ci sono altre transazioni che stanno modificando la base di dati.

Questo impedisce ai comandi di visualizzare dati inconsistenti prodotti da transazioni concorrenti che eseguono inserimenti, cancellazioni o aggiornamenti.

6.1 Livelli di isolamento

I cinque livelli di isolamento messi a disposizione da PosgreSQL sono, in ordine decrescente di isolamento:

- **Serializable**, garantisce l'intera transazione sia eseguita in qualche ordine sequenziale rispetto ad altre transazioni. C'è un completo isolamento da tutte le transazioni concorrenti

- **Repeatable Read**, garantisce che i dati letti durante la transazione non cambieranno a causa di altre transazioni: rifacendo la lettura dei medesimi dati, si ottengono sempre gli stessi

- **Read Committed**: garantisce che qualunque `SELECT` di una transazione vede solo i dati confermati (`COMMITTED`) prima che la `SELECT` inizi.
  È il livello di isolamento di default usato da PostgreSQL.

- **Read Uncommitted**: livello messo a disposizione perché previsto dallo standard SQL, ma è implementato come Read Committed. Quindi, di fatto, questo livello non esiste!

Di seguito proponiamo uno schema dei possibili livelli di isolamento insieme alle anomalie che riescono a impedire:

[1] Dalla doc: *The main advantage to using the MVCC model of concurrency control rather than locking is that in MVCC locks acquired for querying (reading) data do not conflict with locks acquired for writing data, and so reading never blocks writing and writing never blocks reading.* (link: https://www.postgresql.org/docs/8.3/mvcc-intro.html)

Livello di isolamento	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Serializable	No	No	No	No
Repeatable Read	No	No	Sì, ma non in PG	Sì
Read Committed	No	Sì	Sì	Sì
Read Uncommitted	Sì, ma non in PG	Sì	Sì	Sì

Si può imporre che una transazione esegua con un certo livello di isolamento tramite la seguente istruzione:

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL <LEVEL>
...
COMMIT;
```

dove `<LEVEL>` deve essere sostituito dalle keyword `REPEATABLE READ` o `SERIALIZABLE`. Tecnicamente è possibile anche scrivere `READ COMMITTED`, ma è inutile visto che questo è il livello implementato di default.

6.1.1 Read Committed

È il livello di default, quindi non ha bisogno di essere specificato tramite un `SET TRANSACTION ISOLATION LEVEL`.
L'unico caso in cui l'istantanea dei dati visibili alla transazione può cambiare è quando un'altra transazione concorrente effettua il `commit` delle sue modifiche. Prima del `commit`, l'istantanea rimane immutata.

Nonrepeatable Read Una delle anomalie possibili con questo livello di isolamento è la lettura inconsistente: supponiamo di avere due transazioni concorrenti, T1 che effettua una modifica sulla base di dati, T2 che effettua due `SELECT` distinte sui dati. Se T2 esegue una lettura e poi T1 effettua la modifica ed fa il `commit` *prima* della successiva lettura di T1, allora T1 vedrà vedrà due istantanee diverse in momenti diversi.

Phantom Read L'aggiornamento fantasma si verifica se, come prima, abbiamo due transazioni concorrenti, di cui una (diciamo T1) esegue due letture diverse *con condizione*, mentre l'altra (diciamo T2) esegue una modifica sugli stessi dati. Supponiamo che T1 inizi e selezioni

un certo sottoinsieme di righe che soddisfano una certa condizione; successivamente T2 inizia ed esegue delle modifiche che coinvolgono anche alcune delle righe precedentemente selezionate da T1. Quando T2 effettua il `commit`, T1 dovrà ricontrollare le righe *che aveva precedentemente selezionato*[2] per verificare che soddisfino ancora le condizioni.

Modifiche in conflitto Se la modifica della transazione T1 riguarda dei dati su cui sta agendo nello stesso momento un'altra transazione T2, *e le due modifiche sono in conflitto* (ad esempio, entrambe tentano di aggiungere la stessa tupla), allora T1 viene messa *in attesa*. La decisione di far continuare T1 o no dipende dall'esito di T2: se T2 effettua un `commit`, T1 viene abortita, perché la sua modifica andrebbe in conflitto con quella di T2; se invece T2 sceglie un `rollback`, allora T1 può continuare!
Si noti che la transazione viene sospesa solo se le due modifiche non possono coesistere nella base di dati! Invece, se le modifiche effettuate non minacciano l'integrità dei dati (ad esempio, se T1 e T2 aggiungono due tuple diverse e tutti i vincoli vengono rispettati), nessuna delle due viene messa in attesa e le modifiche possono essere rese effettive senza dover aspettare il `commit` dell'altra transazione.

6.1.2 Repeatable Read

Con il livello di isolamento Repeatable Read, l'istantanea dei dati su cui lavora la transazione non cambia mai durante l'esecuzione. Qualunque modifica (seguita da `commit` o no) effettuata da un'altra transazione concorrente è invisibile, gli unici cambiamenti che possono essere percepiti sono quelli effettuati all'interno della stessa transazione.
Questo evidentemente impedisce il presentarsi di anomalie quali lettura sporca, letture inconsistenti e aggiornamento fantasma. L'unica anomalia concessa è la mancata serializzazione.

Se una transazione T1, che lavora con il livello di isolamento Repeatable Read, tenta di modificare dei dati in contemporanea con un'altra transazione T2 che ha cominciato le sue modifiche prima, allora si adotta il seguente protocollo:

1. T1 viene messa in attesa finché T2 non ha deciso il suo esito
2. se T2 effettua un `rollback`, allora l'istantanea di T1 è consistente con lo stato reale della base di dati, quindi a T1 è concesso di portare a termine la sua modifica
3. se T2 effettua un `commit`, allora l'istantanea di T1 non è più consistente con lo stato reale della base di dati, quindi le modifiche che vorrebbe effettuare rischiano di danneggiare l'integrità dei dati. Per questo motivo, a T1 non è concesso di proseguire e si presenta il seguente errore:

[2]Quindi il recheck riguarda solo un certo sottoinsieme righe! Se a causa della modifica altre righe, non considerate dalla `SELECT` precedente, soddisfano il vincolo, non verranno comunque incluse nel risultato finale.

```
ERROR: could NOT serialize access due TO concurrent UPDATE.
```

a indicare che non è stato possibile serializzare l'accesso ai dati da parte delle due transazioni. T1 è costretta a fare un `rollback` e quando ritenterà la sua modifica vedrà i dati come modificati da T2.

Si noti che nel caso di `READ COMITTED` le transazioni non vengono messe in attesa a meno che le operazioni che vogliono fare non siano in conflitto.
Nel livello `REPEATABLE READ` *qualunque tentativo di modifica* da parte di una transazione sui dati condivisi con un'altra transazione porta alla sospensione della transazione che ha iniziato dopo. La modifica potrà essere portata a termine solo se la transazione che bloccava l'altra effettua un `rollback`, anche se queste modifiche non sono in conflitto tra loro!

6.1.3 Serializable

L'esecuzione di un certo gruppo di transazioni concorrenti si dice *serializzabile* se il risultato dell'esecuzione appare come se le transazioni avessero eseguito in qualche ordine seriale.
Il livello di isolamento `SERIALIZABLE` è il più stringente in assoluto e riesce a catturare anche l'anomalia di mancata serializzazione: quando due (o più) transazioni concorrenti effettuano il `commit`, il sistema verifica che il risultato ottenuto sia compatibile con una qualche esecuzione seriale. Se ciò non dovesse essere possibile, viene lanciato il seguente errore:

```
ERROR: could not serialize access due to read/write dependencies among
    transactions
```

Nella maggior parte dei casi, l'anomalia di mancata serializzazione si verifica quando due transazioni vogliono modificare dei dati basandosi sugli stessi valori che devono modificare!

Esempio: anomalia di mancata serializzazione

Supponiamo di avere una tabella `myTable` con un unico attributo, `x`.
Vengono eseguite le seguenti transazioni:

```
1 BEGIN TRANSACTION                 BEGIN TRANSACTION                   1
  ISOLATION LEVEL REPEATABLE READ;  ISOLATION LEVEL REPEATABLE READ;
2 INSERT INTO myTable(x)                                                2
  SELECT MAX(x)+1 FROM myTable;     INSERT INTO myTable(x)              3
  INSERT 0 1                        SELECT MAX(x)+1 FROM myTable;
3                                   INSERT 0 1
4 COMMIT;                           COMMIT;                             4
```

Supponendo che il valore massimo contenuto nella tabella sia 10, una qualunque esecuzione seriale delle due transazioni porterebbe all'inserimento di due righe, una con valore

11 e l'altra con valore 12.
In questo caso, invece, inseriscono entrambe 11! Il risultato ottenuto non è consistente con nessun ordine di esecuzione seriale. Se eseguissimo queste transazioni imponendo il livello di isolamento `SERIALIZABLE`, verrebbe sollevato un errore.

Capitolo 7

Interazione tra Python e PostgreSQL

È possibile interagire con una base di dati PostgreSQL tramite un programma Python grazie alla libreria psycopg2, che implementa la API ufficiale di Python DB-API 2.0 per l'interazione dei programmi Python con basi di dati esterne.
psycopg2 è una libreria scritta quasi completamente in C che implementa i cursori lato server e lato client, comunicazioni asincrone, notifiche e il comando COPY.

7.1 Connessione alla base di dati

L'accesso alla base di dati si effettua tramite un oggetto Connection. Il metodo connect() è un metodo statico di psycopg2 che accetta i parametri necessari per stabilire la connessione, ovvero: l'indirizzo dell' host, il nome del database, il nome dello user e la sua password:

```
connector = psycopg2.connect(host="dbserver.scienze.univr.it", \
database="db0", user="user0", password="xxx" )
```

Si noti che in Python è possibile passare i parametri attuali specificando i parametri formali a cui si riferiscono, in modo da non doversi necessariamente ricordare l'ordine in cui vanno passati.

All'oggetto connector possono essere applicati i seguenti metodi:

- cursor(): ritorna un *cursore* per la base di dati. Un oggetto cursore permette di inviare comandi SQL al DBMS e di accedere al risultato delle query. Verrà approfondito nella sezione 1.2
- commit(): effettua il commit della transazione corrente. Di default, psycopg2 apre automaticamente una transazione nel momento in cui viene inviato il primo comando. Se non viene effettuato il commit esplicitamente prima di chiudere la connessione, tutte le modifiche vanno perse.

- `rollback()`: abortisce la transazione corrente

- `close()`: chiude la connessione corrente. Se non è stato effettuato il `commit`, implica un `rollback` delle operazioni non registrate.

Inoltre è possibile accedere o modificare alcune proprietà:

- `autocommit`: se impostata a `True`, ogni comando inviato è una transazione isolata, quindi viene eseguito il `commit` di ogni comando. Il valore di default è `False` e implica che il termine di una transazione debba essere dichiarato esplicitamente dal programmatore tramite un `commit` o un `rollback`. Impostare a `True` questa proprietà è buona pratica quando si hanno programmi molto lunghi, per evitare di lasciare la sessione nello stato *"idle in transaction"*.

- `readonly`: se impostata a `True`, permette di inviare solo comandi di interrogazione (`SELECT`) ma non comandi DDL o DML. Questa proprietà è utile quando si sa a priori che non si invieranno comandi di definizione/modifica dei dati; impostare `readonly = True` ottimizza la connessione (...)

- `isolation_level`: modifica il livello di isolamento per la prossima transazione. I valori leciti sono: `'READ UNCOMMITTED'`, `'READ COMMITTED'`, `'REPEATABLE READ'`, `'SERIALIZABLE'`, `'DEFAULT'`. È preferibile assegnare questa variabile subito dopo la crazione di una connessione.

- `status`: è una proprietà *read-only* (!) che rappresenta lo stato della connessione. È indefinito per connessioni chiuse.

L'impostazione di una certa proprietà si effettua con una sintassi di questo tipo:

```
connector.isolation_level = 'REPEATABLE READ'
connector.autocommit = 'True'
```

È possibile creare una connessione tramite la parola chiave `with`:

```
with psycopg2.connect(...) as connector:
...
```

dove l'identificatore dopo `as` sarà il nome dell'oggetto e i comandi da inviare al DBMS dovranno essere scritti all'interno del blocco.
Il vantaggio di usare questa notazione è che la connessione viene automaticamente chiusa all'uscita dal blocco, quindi non c'è bisogno di usare il metodo `close()`.

7.2 Cursore

Il cursore è un oggetto che gestisce l'interazione con la base di dati, si crea con l'istruzione `connector.cursor()`. Se il comando inviato è una query, il risultato verra memorizzato all'interno del cursore.

Il cursore è legato a una certa connessione per tutta la durata della sua vita. I cursori creati all'interno di una stessa sessione non sono isolati: ogni modifica del database effettuata da un cursore è immediatamente visibile a tutti gli altri cursori. I cursori che invece appartengono a sessioni diverse possono essere isolati oppure no, in base al livello di isolamento impostato.

I metodi da applicare al cursore per l'esecuzione di comandi sono:

- `execute(<comando> [, <parametri>])`: prepara ed esegue il comando SQL usando i parametri indicati, che **devono essere passati come tupla o come dict!**. Il metodo ritorna `None` ed evenutali risultati si possono ricavare dal cursore applicando i metodi `fetchone()` o `fetchmany()`.

  Esempio di query "statica" (senza parametri):

  ```
  cur.execute("CREATE TABLE test (id INTEGER PRIMARY KEY, num NUMERIC,
      data VARCHAR)")
  ```

  Esempio di query "dinamica" (con parametri):

  ```
  cur.execute("INSERT INTO test (id, num, data) VALUES (%s, %s, %s)",
      (100, 18.50, 'abc'))
  ```

- `executemany(<comando>, <lista di parametri>)`: prepara ed esegue un comando SQL per ciascun valore nella lista di parametri:

  ```
  cur.execute("INSERT INTO test (id, num, data) VALUES (%s, %s, %s)",
      (100, 18.50, 'abc'), (68, 12.00, 'bar'), (14, 30, 'tree'))
  ```

  Questo comando è pensato per fare più volte lo stesso tipo di modifica.

Attenzione

L'utilizzo di `executemany()` è fortemente sconsigliato, in quanto l'implementazione attuale è meno efficiente dell'esecuzione di ripetute `execute()` singole oppure, ancora meglio, un unico `INSERT` con più tuple:

```
cur.execute("INSERT INTO test (id, num, data) VALUES (%s, %s, %s), (%s,
    %s), \ (%s, %s)", (100, 18.50, "abc", 68, 12.00, 'bar', 14, 30, 'tree'))
```

7.3 Passaggio di parametri

Si noti che l'unico segnaposto utilizzato per il passaggio di parametri è `%s`, quindi non viene fatta distinzione tra i diversi tipi dei paramentri passati. Questo è possibile perché la conversione da tipo Python a tipo SQL viene fatta in automatico per tutti i tipi fondamentali.

Python	PostgreSQL
None	NULL
int	SMALLINT, INTEGER, BIGINT
date	DATE
bool	BOOLEAN
Decimal	NUMERIC
time	TIME
float	REAL, DOUBLE
str	VARCHAR, TEXT
datetime	TIMESTAMP

È possibile anche dare un nome agli argomenti usando la sintassi `%(name)s` per i segnaposti della query e dichiarando il mapping con i valori effettivi al momento del passaggio di parametri. In questo modo, è possibile specificare i valori in qualunque ordine.

Esempio

```
cur.execute("""INSERT INTO some_table (an_int, a_date, another_date,
    a_string) VALUES (%(int)s, %(date)s, %(date)s, %(str)s);""", \
{'int': 10, 'str': "O'Reilly", 'date': datetime.date(2005, 11, 18)})
```

Come ultima accortezza, si ricordi che i parametri vanno passati *sempre* sotto forma di *tupla* o *lista*, anche se bisogna passare un singolo valore:

```
cur.execute("INSERT INTO foo VALUES (%s)", "bar") # WRONG
```

```
cur.execute("INSERT INTO foo VALUES (%s)", ("bar")) # WRONG
cur.execute("INSERT INTO foo VALUES (%s)", ("bar",)) # correct
cur.execute("INSERT INTO foo VALUES (%s)", ["bar"]) # correct
```

7.4 Estrazione del risultato

Abbiamo già accennato che le che le tuple risultanti da una query vengono memorizzate all'interno del cursore, quindi esistono comandi specifici che permettono di prelevare queste tuple dal cursore.
All'interno del cursore c'è un "puntatore" che punta alla prossima tupla da estrarre, e man mano che vengono estratte le tuple questo puntatore si sposta sempre più in basso nella tabella. Una volta che la tabella è terminata, non è più possibile "riposizionare" il puntatore in cima e rieseguire l'estrazione, bisogna eseguire nuovamente la query.

- `fetchone()`: ritorna *una sola* tupla della tabella risultato. Se la query non ha fornito alcun risultato, restituisce `None`.

  ```
  >>> cur.execute("SELECT * FROM test WHERE id = %s", (3,))
  >>> cur.fetchone()
  (3, 42, 'bar')
  ```

- `fetchmany(<numero righe>)`: ritorna le prime `<numero righe>` tuple dalla tabella risultato sotto forma di*lista di tuple* a partire dalla tupla a cui è arrivato il puntatore, o una lista vuota nel caso non ci siano tuple. Se non viene specificato un numero, la classe cursore dispone di un attributo, `arraysize`, che specifica il numero di righe da prelevare ogni volta che viene invocata la `fetchmany()`. Questo valore è settato di default a uno, quindi chiamare una `fetchmany()` senza parametri equivale ad effettuare una `fetchone()`.

  ```
  >>> cur.execute("SELECT * FROM test WHERE id < %s", (4,))
  >>> cur.fetchmany(3)
  [(1, 100, "abc'def"), (2, None, 'dada'), (3, 42, 'bar')]
  >>> cur.fetchmany(2)
  []
  ```

- `fetchall()`: estrae tutte le tuple (rimanenti), restituendole sotto forma di lista di tuple.

  ```
  >>> cur.execute("SELECT * FROM test;")
  >>> cur.fetchall()
  [(1, 100, "abc'def"), (2, None, 'dada'), (3, 42, 'bar')]
  ```

Dato che un oggetto `cursor` è *iterabile*, un altro modo per estrarre tutte le righe del risultato senza usare i metodi `fetch` elencati sopra è quello di iterare con un ciclo `for-each` sul cursore stesso:

```
>>> cur.execute("SELECT * FROM test WHERE id < %s", (4,))
>>> for record in cur:
...     print(record, end=", ")
(1, 100, "abc'def"), (2, None, 'dada'), (3, 42, 'bar'),
```

La classe cursore dispone inoltre di alcuni attributi che possono essere utili per ottenere informazioni sull'esecuzione del comando:

- `rowcount`: attributo read-only che contiene il numero di righe prodotte o modificate dall'ultima `execute()`. Assume il valore `-1` nel caso in cui non sia stato ancora inviato alcun comando oppure se il numero di righe è troppo grande e non può essere determinato a priori.
- `rownumber`: specifica la posizione del cursore rispetto alle righe della tabella risultato, nonché la posizione della prossima tupla da resituire.
- `statusmessage`: attributo read-only che contiene il testo del messaggio ritornato dall'ultimo comando inviato:

  ```
  >>> cur.execute("INSERT INTO test (num, data) VALUES (%s, %s)", (42, 'bar'))
  >>> cur.statusmessage
  'INSERT 0 1'
  ```

  Per chiudere un cursore, si utilizza il metodo `close()`. Tuttavia, questo si può evitare se si utilizza la keyword `with` al momento della creazione del cursore:

  ```
  with connector.cursor() as cur:
  ...
  ```

  Tutti i comandi che utilizzano il cursore `cur` devono essere scritti all'interno del blocco; quando si trova la prima istruzione al di fuori del blocco, il cursore viene chiuso in automatico senza bisogno di invocare la `close()`.

Capitolo 8

Interazione tra Java e PostgreSQL

Un programma Java può interagire con una base di dati esterna tramite l'API ufficiale, `JDBC` (Java DataBase Connectivity). Per PostgreSQL, esistono due implementazioni diverse di questa API, ma in questo corso di studierà solo la libreria `JDBC 42.x`[1], scritta interamente in Java.

8.1 Connessione alla base di dati

Prima di effettuare la connessione vera e propria, bisogna assicurarsi di caricare il driver tramite la seguente istruzione:

```
Class.forName("org.postgresql.Driver")
```

dove `org.postgresql.Driver` è il nomer del driver. Il metodo `forName(...)` è un metodo statico che ritorna l'oggetto `Class` associato al nome che viene passato come parametro, utilizzando il class loader.

La connessione vera e proprie viene stabilita tramite il seguente comando:

```
Driver.getConnection(<url>, <user>, <password>)
```

che ritorna un oggetto di tipo `Connection`. I parametri da passare sono:

- `url`, che può essere specificato nella forma `jdbc:postgresql://host/database`[2]
- `user`, ovvero il nome dell'utente
- `password`, ovvero la password dell'utente

Per stabilire una connessione alla base di dati dell'università, ad esempio, si deve scrivere:

[1] https://jdbc.postgresql.org/

[2] Gli altri formati accettati sono descritti alla pagina https://jdbc.postgresql.org/documentation/head/connect.html

```
DriverManager.getConnection("jdbc:postgresql://dbserver.scienze.univr.it/id311fgy",
    "id311fgy", "...")
```

Una volta creato l'oggetto `Connection`, si possono invocare alcuni metodi per inviare query alla base di dati, estrarre il risultato, o effettuare modifiche.
Ogni volta che vogliamo inviare un comando SQL, abbiamo bisogno di un oggetto `Statement` o `PreparedStatement` (vedi sezione 8.2), che possono essere instanziati tramite i seguenti metodi:

- `createStatement()` ritorna un oggetto `Statement`
- `prepareStatement("query")` ritorna un oggetto `PreparedStatement`

Gli altri metodi fondamentali della classe `Connection` sono:

- `commit()` registra la transazione. **Attenzione:** le connessioni JDBC sono in autocommit, quindi a meno di una specifica diversa (ovvero, a meno di eseguire `autoCommit(false)`) ogni comando viene inviato come una transazione singola e il commit viene fatto in modo automatico
- `rollback()` abortisce la transazione corrente
- `close()` chiude la connessione corrente

8.2 Esecuzione di comandi SQL

Nella sezione precedente abbiamo accennato al fatto che l'invio di comandi SQL necessita degli oggetti di tipo `Statement` o `PreparedStatement`.
La differenza tra i due sta nel fatto che il primo permette di inviare solo query "statiche", di cui conosciamo tutti i parametri fin dall'inizio, il secondo invece consente di inviare query con parametri variabili. Per capire meglio la differenza, si osservino i seguenti esempi: qui si fa uso di un'oggetto `Statement`, perché il testo della query è completamente noto a propri.

```
Statement st = conn.createStatement();
ResultSet rs = st.executeQuery("SELECT * FROM mytable WHERE column = 500");

while(rs.next()){
    System.out.println(rs.getString("column"))
}
```

In questo secondo pezzo di codice, invece, si deve ricorrere a un `PreparedStatement`, perché il valore da inserire nella clausola `WHERE` dipende dall'input che fornirà l'utente.

```
Scanner keyboard = new Scanner(System.in);
System.out.println("Insert integer value: ");
```

```
int n = keyboard.nextInt();

PreparedStatement pst = conn.prepareStatement("SELECT * FROM mytable WHERE
    column = ?");
pst.setInt(1, n);

ResultSet rs = pst.executeQuery();

while (rs.next()){
    System.out.println(rs.getString("column"));
}
```

La query si esegue applicando all'oggetto `PreparedStatement` il metodo `executeQuery()` e il risultato della query viene memorizzato in oggetto di tipo `ResultSet`.
Per eseguire un comando di modifica (DDL o DML), invece, c'è il metodo `executeUpdate()`, che restiuisce il numero di righe che sono state modificate (nel caso di istruzioni DML) oppure 0 per comandi che non restituiscono nulla (solitamente istruzioni DDL).
Per parametrizzare la query si utilizza un insieme di metodi nella forma `set*(n, <valore>)`, che permettono di sostituire la `n`-esima occorrenza di `?` con `<valore>`.
Java mette a disposizione un metodo `set*()` per ogni tipo fondamentale, quindi abbiamo: `setString(<indice>, <valore>)`, `setInt(<indice>, <valore>)` e simili.

8.3 Estrazione del risultato

Abbiamo detto che il risultato di una query inviata tramite il comando `executeQuery()` viene memorizzato in forma di tabella all'interno di un oggetto speciale chiamato `ResultSet`.
Questo oggetto dispone di un cursore che all'inizio è posizionato subito prima della prima riga del risultato. Invocando il metodo `next()`, viene spostato alla riga successiva, in modo che punti sempre alla prossima riga da restituire. Quando la tabella è terminata, `next()` ritorna `false`.
Similmente a quanto già visto per Python, il cursore può spostarsi solo in avanti, e una volta terminata la scansione della tabella non è più possibile riportarlo all'inizio.
Vediamo quindi i metodi principali della classe `ResultSet`:

- `next()`: sposta il cursore alla prossima riga. Restituisce `true` se la prossima riga esiste, `false` altrimenti

- `get*(<nome colonna>)`: sono un insieme di metodi che permettono di estrarre un dato di un certo tipo dalla colonna `nome colonna` della riga corrente. Come per l'insieme siste un metodo `get*()` per ogni tipo fondamentale di Java, quindi abbiamo: `getString("...")`, `getInt("...")`, `getDate("...")`, `getBoolean("...")` e simili.
  In alternativa, è possibile passare come parametro l'indice della colonna, quindi si può

scrivere un comando del tipo `getString(2)`, che restituirà la stringa contenuta nella seconda colonna della riga corrente.[3]

[3]Le colonne sono numerate a partire da 1