**Università degli Studi di Verona**

DIPARTIMENTO DI INFORMATICA
Laurea in Informatica

# Basi di Dati

Programma di laboratorio

Autori:
**Davide Bianchi**
**Matteo Danzi**

**Anno Accademico 2016/2017**

# Indice

# 1 Gestione base di dati con Postgresql

Di seguito si trova una panoramica dei comandi Postgres più comuni per la gestione di una base di dati.

## 1.1 Comando CREATE TABLE

Il comando `CREATE TABLE` è usato per creare tabelle nella base di dati. La sintassi generale è:

```
CREATE TABLE nomeTabella (
    nomeAttributo dominioAttributo vincoli,
    ...
);
```

dove `nomeAttributo` è il nome dell'attributo nella tabella, `dominioAttributo` è il dominio dell'attributo da aggiungere alla tabella.

### 1.1.1 Domini elementari

I domini di default disponibili in Postgres sono:

- `BOOLEAN`: valori booleani (true/false);
- `INTEGER`: valori interi a 4 byte;
- `SMALLINT`: valori interi a 2 bit;
- `NUMERIC(p, s)`: valori decimali approssimati, dove `p` è la precisione del numero (cifre a sinistra e a destra della virgola) e `s` la scala (numero di cifre decimali dopo la virgola);
- `DECIMAL(p, s)`: valori deciamli approssimati, con i parametri uguali a `NUMERIC`.
- `REAL`: valori in virgola mobile a 6 cifre decimali;
- `DOUBLE PRECISION`: valori in virgola mobile approssimati a 15 cifre decimali;

**Nota:** Se si devono rappresentare importi di denaro che contengono anche decimali, **MAI** usare `REAL` o `DOUBLE PRECISION` ma usare `NUMERIC`!

### 1.1.2 Domini di caratteri

- `CHARACTER`: singoli caratteri;
- `CHARACTER(n)`:stringa di caratteri di lunghezza n;
- `VARCHAR`: stringhe di caratteri di lunghezza variabile;
- `TEXT`: testo libero (solo Postgres).

### 1.1.3 Domini di bit/booleani

- `BIT`: singoli bit;
- `VARBIT(n)`: stringa di bit di lunghezza fissa;
- `VARBIT`: stringa di bit di lunghezza arbitraria.
- `BOOLEAN`: valori booleani, possono essere solo singoli.

**Nota:** non sono ammesse stringhe di booleani.

### 1.1.4 Domini di tempo

- `DATE`: date rappresentate tra apici e nel formato `YYYY-MM-DD`;
- `TIME(precisione)`: misure di tempo nel formato `hh:mm:ss:[precisione]`;
- `INTERVAL`: intervalli di tempo.
- `TIME/TIMESTAMP WITH TIME ZONE`: con tutti i dati per il tempo, con indicazioni sul fuso.

## 1.2 Comando CREATE DOMAIN

Questo comando è usato per creare un dominio utente **invariabile nel tempo**.

```
CREATE DOMAIN nome AS tipoBase [default]
    [vincolo]
```

I valori di default e i vincoli sono opzionali.
**Esempio:**

```
CREATE DOMAIN giorniSettimana AS CHAR(3)
    CHECK ( VALUE IN ('LUN', 'MAR', 'MER', 'GIO', 'VEN',
    'SAB', 'DOM'));
```

## 1.3 Vincoli di attributo e di tabella

Vincoli di attributo/intrarelazionali specificano proprietà che devono essere soddisfatte da ogni tupla di una singola relazione della base di dati.

```
[ CONSTRAINT vincolo ]
{ NOT NULL |
  CHECK ( espressione ) [ NO INHERIT ] |
  DEFAULT valore |
  UNIQUE |
  PRIMARY KEY |
  REFERENCES tabella [ ( attributo ) ]
    [ ON DELETE azione ] [ ON UPDATE azione ] }
```

Vincoli di tabella:

```
[ CONSTRAINT vincolo ]
{ CHECK ( espressione ) [ NO INHERIT ] |
  UNIQUE ( attributo [, ... ]) |
  PRIMARY KEY ( attributo [, ... ]) |
  FOREIGN KEY ( attributo [, ... ])
  REFERENCES reftable [ ( refcolumn [ , ... ]) ]
    [ ON DELETE azione ] [ ON UPDATE azione ] }
```

- `NOT NULL`: determina che il valore nullo non è ammesso come valore dell'attributo.
- `DEFAULT valore`: specifica un valore di default per un attributo quando un comando di inserimento dati non specifica nessun valore per l'attributo.

  **Esempio:**

  ```
  nome VARCHAR (20) NOT NULL,
  cognome VARCHAR (20) NOT NULL DEFAULT ''
  ```

- `UNIQUE`: impone che i valori di un attributo (o di un insieme di attributi) siano una **superchiave**.
- `PRIMARY KEY`: identifica l'attributo che rappresenta la chiave primaria della relazione:

  - Si usa una sola volta per tabella.
  - Implica il vincolo `NOT NULL`.

  **Esempio:**

```
matricola CHAR(6) PRIMARY KEY;
```

  oppure su più attributi

```
nome VARCHAR(20),
cognome VARCHAR(20),
PRIMARY KEY(nome, cognome)
```

- `CHECK (vincolo)`: specifica un vincolo generico che devono soddisfare le tuple della tabella.

### 1.3.1 Vincoli di integrità referenziale

Un vincolo di integrità referenziale si dichiara nella tabella interna e ha due possibili sintassi.

- `REFERENCES`: **vincolo di attributo**, da usare quando il vincolo è su un singolo attributo della tabella interna, $|A| = 1$.
- `FOREIGN KEY`: **vincolo di tabella** , da usare quando il vincolo coinvolge più attributi della tabella interna, $|A| > 1$.

**Esempio:**

```
CREATE TABLE Interna
...
attributo VARCHAR(15) REFERENCES TabellaEsterna (chiave)
...

...
piano VARCHAR (10),
stanza INTEGER,
FOREIGN KEY (piano, stanza) REFERENCES Ufficio (piano, nStanza)
```

## 1.4 Comando ALTER TABLE

La struttura di una tabella si può modificare dopo la sua creazione con il comando `ALTER TABLE`.

- Aggiunta di un nuovo attributo con `ADD COLUMN`:

```
ALTER TABLE impiegato ADD COLUMN stipendio NUMERIC(8,2);
```

- Rimozione di un attributo con `DROP COLUMN`:

```
ALTER TABLE impiegato DROP COLUMN stipendio;
```

- Modifica di un valore di default di un attributo con `ALTER COLUMN`:

```
ALTER TABLE impiegato ALTER COLUMN stipendio
    SET DEFAULT 1000.00;
```

## 1.5 Comando INSERT INTO

Una tabella viene popolata con il comando `INSERT INTO`:

```
INSERT INTO impiegato (matricola, nome, cognome)
    VALUES ('A00001', 'Mario', 'Rossi'),
           ('A00002', 'Luca', 'Bianchi');
```

## 1.6 Comando UPDATE

Una tupla di una tabella può essere modificata con il comando UPDATE:

```
UPDATE tabella
    SET attributo = espressione [, ... ]
    [ WHERE condizione ];
```

condizione è una espressione booleana che seleziona quali righe aggiornare. Se WHERE non è presente, tutte le tuple saranno aggiornate.

**Esempio:**

```
UPDATE impiegato
    SET stipendio = stipendio * 1.10
    WHERE nomeDipartimento = 'Vendite';
UPDATE impiegato
    SET telefono = '+39' || telefono;
```

**Nota:** L'operatore '||' concatena due espressioni e ritorna la stringa corrisp.

## 1.7 Comando DELETE

Le tuple di una tabella vengono cancellate con il comando DELETE:

```
DELETE FROM impiegato WHERE matricola = 'A001';
```

Una tabella viene cancellata con il comando DROP TABLE.

## 1.8 Politiche di reazione

In SQL si possono attivare diverse politiche di adeguamento della tabella interna

```
FOREIGN KEY ( column_name [ , ... ]) REFERENCES
    reftable [ ( refcolumn [ , ... ]) ]
   ON DELETE reazione ON UPDATE reazione
```

- CASCADE: la modifica del valore di un attributo riferito nella tabella master si propaga anche in tutte le righe corrispondenti nelle tabelle slave.
- SET NULL: la modifica del valore di un attributo riferito nella tabella master determina che in tutte le righe corrispondenti nelle tabelle slave il valore dell'attributo referente è posto a NULL (se ammesso).
- SET DEFAULT: la modifica del valore di un attributo riferito nella tabella master determina che in tutte le righe corrispondenti nelle tabelle slave il valore dell'attributo referente è posto al valore di default (se esiste).
- NO ACTION: indica che non si fa nessuna azione. Il vincolo però deve essere sempre valido. Quindi, la modifica del valore di un attributo riferito nella tabella master non viene effettuata.

## 1.9 Query sul database

In SQL, esiste solo un comando per interrogare un base di dati: SELECT.

```
SELECT [ DISTINCT ]
[ * | expression [[ AS ] output_name ] [ , ...] ]
[ FROM from_item [ , ...] ]
[ WHERE condition ]
[ GROUP BY grouping_element [ , ...] ]
[ HAVING condition [ , ...] ]
[ { UNION | INTERSECT | EXCEPT } [ DISTINCT ]
    other_select ]
```

```
[ ORDER BY expression [ ASC | DESC | USING operator ]]
...
```

dove

- `*` è un'abbreviazione per indicare tutti gli attributi delle tabelle.
- `expression` è un'espressione che determina un attributo.
- `output_name` è il nome assegnato all'attributo che conterrà il risultato della valutazione dell'espressione `expression` nella relazione risultato.
- `from_item` è un'espressione che determina una sorgente per gli attributi.
- `condition` è un'espressione booleana per selezionare i valori degli attributi.
- `grouping_element` è un'espressione per poter eseguire operazioni su più valori di un attributo e considerare il risultato.
- `DISTINCT` : se presente richiede l'eliminazione delle tuple duplicate.

### 1.9.1 Operatore LIKE e SIMILAR TO

Nella clausola `WHERE` può apparire l'operatore `LIKE` per il confronto di stringhe. `LIKE` è un operatore di pattern matching. I pattern si costruiscono con i caratteri speciali `_` (1 carattere qualsiasi) e `%` (0 o più caratteri qualsiasi):

```
WHERE attributo [ NOT ] LIKE 'pattern';
```

L'operatore `SIMILAR TO` è un `LIKE` più espressivo che accetta espressioni regolari (versione SQL) come pattern. Esempi di componenti di espressioni regolari:

- `_` = 1 carattere qualsiasi. `%` = 0 o più caratteri qualsiasi.
- `*` = ripetizione del precedente match 0 o più volte. `+` = ripetizione del precedente match UNA o più volte.
- `{n,m}` = ripetizione del precedente match almeno n e non più di m volte.
- `[...]` = ... è un elenco di caratteri ammissibili

**Esempio:** Studenti con cognome che inizia con 'A' o 'B', o 'D', o 'N' e finisce con 'a':

```
SELECT cognome, nome, città FROM Studente
WHERE cognome SIMILAR TO '[ABDN]{1}%a';
```

### 1.9.2 Operatore BETWEEN

Nella clausola `WHERE` può apparire l'operatore `[ NOT ] BETWEEN` per testare l'appartenenza di un valore ad un intervallo. Gli estremi dell'intervallo sono **inclusi**.
**Esempio:** Tutti gli studenti che hanno matricola tra 'IN0002' e 'IN0004'.

```
SELECT cognome, nome, matricola FROM Studente
WHERE matricola BETWEEN 'IN0002' AND 'IN0004';
```

### 1.9.3 Operatore IN

Nella clausola `WHERE` può apparire l'operatore `[ NOT ] IN` per testare l'appartenenza di un valore ad un insieme.
**Esempio:** Tutti gli studenti che hanno matricola nell'elenco 'IN0001', 'IN0003' e 'IN0005'.

```
SELECT cognome, nome, matricola FROM Studente
WHERE matricola IN ('IN0001', 'IN0003', 'IN0005');
```

### 1.9.4 Operatore IS NULL

Nella clausola `WHERE` può apparire l'operatore `[ NOT ] IS NULL` per testare se un valore è NOT KNOWN (==NULL) o no.
**Esempio:** Tutti gli studenti che NON hanno una città.

```
SELECT cognome, nome, città FROM Studente
WHERE città IS NULL;
```

**Nota:** In SQL, `NULL` non è uguale a `NULL`. NON SI PUÒ usare `'='` o `'<>'` con il valore NULL!

### 1.9.5 Operatore ORDER BY

La clausola `ORDER BY` ordina le tuple del risultato in ordine rispetto agli attributi specificati.
**Esempio:** Tutti gli studenti in ordine decrescente rispetto al cognome e crescente (lessicografico) rispetto al nome.

```
SELECT cognome, nome
FROM Studente
ORDER BY cognome DESC, nome;
```

### 1.9.6 Operatori di aggregazione

Sono operatori che permettono di determinare **un** valore considerando i valori ottenuti da una `SELECT`. Due tipi principali:

- `COUNT`
- `MAX`, `MIN`, `AVG`, `SUM`

Quando si usano gli operatori aggregati, dopo la `SELECT` *non* possono comparire espressioni che usano i valori presenti nelle singole tuple perché il risultato è sempre e solo una tupla.
`COUNT` restituisce il numero di tuple significative nel risultato dell'interrogazione:

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

dove `expr` è un'espressione che usa attributi e funzioni di attributi ma non operatori di aggregazione.
Tre casi comuni:

- `COUNT(*)` ritorna il numero di tuple nel risultato dell'interrogazione.
- `COUNT(expr)` ritorna il numero di tuple in ciascuna delle quali il valore expr è non nullo.
- `COUNT(ALL expr)` è un alias a `COUNT(expr)` .
- `COUNT(DISTINCT expr)` come con `COUNT(expr)` ma con l'ulteriore condizione che i valori di `expr` sono distinti.

`MAX`, `MIN`, `AVG`, `SUM` determinano un valore numerico (`SUM`/`AVG`) o alfanumerico (`MAX`/`MIN`) considerando le tuple significative nel risultato dell'interrogazione.
**Esempi:**
Calcola la media delle medie degli studenti.

```
SELECT AVG(media)::DECIMAL (5,2)
FROM Studente;
```

Calcola la media delle medie distinte degli studenti.

```
SELECT AVG(DISTINCT media)::DECIMAL(5,2)
FROM Studente;
```

### 1.9.7 Interrogazioni con raggruppamento

Un raggruppamento è un insieme di tuple che hanno medesimi valori su uno o più attributi caratteristici del raggruppamento.
La clausola `GROUP BY attr [, ...]` permette di determinare tutti i raggruppamenti delle tuple della relazione risultato (tuple selezionate con la clausola `WHERE` ) in funzione degli attributi dati.
In una interrogazione che fa uso di `GROUP BY` , possono comparire come argomento della `SELECT` solamente gli attributi utilizzati per il raggruppamento e funzioni aggregate valutate sugli altri attributi.
**Esempi:**
Visualizzare tutte le città raggruppate della tabella Studente.

```
SELECT città
FROM Studente
GROUP BY città;
```

Visualizzare tutte le città e i cognomi raggruppati.

```
SELECT cognome, LOWER(città)
FROM Studente
GROUP BY cognome, LOWER(città);
```

**Nota1:** NON SI POSSONO SPECIFICARE attributi che non sono raggruppati dopo il `SELECT`.

```
SELECT ~~cognome~~, città
FROM Studente
GROUP BY città;
```

**Nota2:** Si possono specificare espressioni con operatori di aggregazione su attributi non raggruppati.

```
SELECT LOWER(città) AS città,
  CAST(AVG(media) AS DECIMAL(5,2)) AS media
FROM Studente
GROUP BY LOWER(città);
```

- La clausola `WHERE` permette di selezionare le righe che devono far parte del risultato.
- La clausola `HAVING` permette di selezionare i raggruppamenti che devono far parte del risultato.
- La sintassi è `HAVING bool_expr` , dove `bool_expr` è un'espressione booleana che può usare gli attributi usati nel `GROUP BY` e/o gli altri attributi mediante operatori di aggregazione.

**Esempi:**
Visualizzare tutte le città raggruppate che iniziano con 'V' della tabella Studente.

```
SELECT città
FROM Studente
GROUP BY città
HAVING città LIKE 'V%';
```

Visualizzare tutte le città e i cognomi raggruppati con almeno due studenti con lo stesso cognome.

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

### 1.9.8 Comando JOIN

Si è visto che se sono presenti due o più nomi di tabelle, si esegue il prodotto cartesiano tra tutte le tabelle e lo schema del risultato può contenere tutti gli attributi del prodotto cartesiano. Il prodotto cartesiano di due o più tabelle è un `CROSS JOIN` . A partire da SQL-2, esistono altri tipi di `JOIN` (`join_type`): `INNER JOIN`, `LEFT OUTER JOIN` , `RIGHT OUTER JOIN` e `FULL OUTER JOIN`.

```
table_name [ NATURAL ] join_type table_name
  [ ON join_condition [ , ...]]
```

dove `join_condition` è un'espressione booleana che seleziona le tuple del join da aggiungere al risultato. Le tuple selezionate possono essere poi filtrate con la condizione della clausola `WHERE`.

#### INNER JOIN

Rappresenta il tradizionale $\Theta$ join dell'algebra relazionale. Combina ciascuna riga $r_1$ di $table_1$ con ciascuna riga di $table_2$ che soddisfa la condizione della clausola `ON`.

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

| cognome | nomerep |
|---|---|
| Rossi | Vendite |
| Verdi | Acquisti |

#### LEFT OUTER JOIN

Si esegue un `INNER JOIN`. Poi, per ciascuna riga $r_1$ di $table_1$ che non soddisfa la condizione con qualsiasi riga di $table_2$, si aggiunge una riga al risultato con i valori di $r_1$ e assegnando `NULL` agli altri attributi.

```
INSERT INTO Reparto (nomerep, sede, telefono)
   VALUES ('Finanza', 'Padova', '02 8028888');
SELECT R.nomeRep, I.cognome
FROM Reparto R LEFT OUTER JOIN Impiegato I
     ON I.nomerep = R.nomerep;
```

| nomerep | cognome |
|---|---|
| Acquisti | Rossi |
| Vendite | Verdi |
| Finanza | |

**Nota:** Il `LEFT OUTER JOIN` non è simmetrico!
Con le medesime tabelle si possono avere risultati diversi invertendo l'ordine delle tabelle nel join!

#### RIGHT OUTER JOIN

Si esegue un `INNER JOIN`. Poi, per ciascuna riga $r_2$ di $table_2$ che non soddisfa la condizione con qualsiasi riga di $table_1$, si aggiunge una riga al risultato con i valori di $r_2$ e assegnando `NULL` agli altri attributi.

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

| cognome | nomerep |
|---|---|
| Rossi | Vendite |
| Verdi | Acquisti |
| | Finanza |

#### FULL OUTER JOIN

È equivalente a: `INNER JOIN` + `LEFT OUTER JOIN` + `RIGHT OUTER JOIN` .
**Nota:** Non è equivalente a `CROSS JOIN`!

### 1.9.9 Interrogazioni nidificate

SQL permette il confronto di un valore (ottenuto come risultato di una espressione valutata sulla singola riga) con il risultato dell'esecuzione di una interrogazione SQL. L'interrogazione che viene usata nel confronto viene definita direttamente nel predicato interno alla clausola `WHERE`.
**Attenzione:** il confronto è tra un valore di un attributo (valore singolo) e il risultato di una interrogazione (possibile insieme di valori). Quindi:

- Gli operatori di confronto tradizionali ($<, >, <>, =, \dots$) **NON** non possono essere usati.
- Si devono usare dei nuovi operatori, `EXISTS`, `IN`, `NOT IN`, `ALL`, `ANY/SOME`, che estendono i tradizionali operatori a questo tipo di confronti.

### 1.9.10 Operatore EXISTS

```
EXISTS (subquery)
```

`(subquery)` è una `SELECT`.
`EXISTS` ritorna falso se `(subquery)` non contiene righe; vero altrimenti.
`EXISTS` è significativo quando nella `(subquery)` si selezionano righe usando i valori della riga corrente nella `SELECT` principale: data binding.
Determinare i nomi degli impiegati che sono diversi tra loro ma di pari lunghezza:

```
SELECT I.nome
FROM Impiegato I
WHERE EXISTS(
    SELECT 1 FROM Impiegato Ii WHERE I.nome <> Ii.nome
        AND CHAR_LENGTH(I.nome) = CHAR_LENGTH(Ii.nome)
);
```

`I.nome` nella subquery è il valore di nome nella riga corrente della `SELECT` principale.
**Nota:** L'operatore `NOT` può essere usato in coppia con `EXISTS`.

### 1.9.11 Operatore IN

```
[ ROW ] (expr [ ,...]) IN (subquery)
```

- `expr` è un'espressione costruita con un attributo della query principale. Ci possono essere 1 o più espressioni.
- La `(subquery)` deve restituire un numero di colonne pari al numero di espressioni in `(expr [,...])`.
- I valori dell'espressioni vengono confrontati con i valori di ciascuna riga del risultato di `(subquery)`.
- Il confronto ritorna vero se i valori sono uguali ai valori di almeno una riga della subquery.

**Esempio:**

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

### 1.9.12 Operatore ANY/SOME

```
expression operator ANY (subquery)
expression operator SOME (subquery)
```

`(subquery)` è una `SELECT` che deve restituire UNA sola colonna;
`expression` è un'espressione che coinvolge attributi della `SELECT` principale.
`operator` è un operatore di confronto, come '=' o '>='.
`ANY` : ritorna vero se `expression` è `operator` rispetto al valore di una qualsiasi riga del risultato di `(subquery)`.
`SOME` è uno sinonimo di `ANY`.

Visualizzare il nome degli insegnamenti che hanno un numero di crediti inferiore alla media dell'ateneo di un qualsiasi anno accademico:

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

### 1.9.13 Operatore ALL

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

`(subquery)` è una `SELECT` che deve restituire UNA sola colonna;
`expression` è un'espressione che coinvolge attributi della `SELECT` principale.
`operator` è un operatore di confronto, come '=' o '>='.
`ALL` : ritorna vero se `expression` è `operator` rispetto al valore di ciascuna riga del risultato di `(subquery)`.

Trovare il nome degli insegnamenti (o moduli) con almeno un docente e crediti maggiori rispetto ai crediti di ciascun insegnamento del corso di laurea con id=6.

```
SELECT DISTINCT I.nomeins, IE.crediti
FROM Insegn I
JOIN InsErogato IE ON I.id = IE.id_insegn
JOIN Docenza D ON IE.id = D.id_inserogato
WHERE IE.crediti > ALL (
    SELECT crediti FROM InsErogato
    WHERE id_corsostudi = 6
);
```

### 1.9.14 Operatori insiemistici UNION/INTERSECT/EXCEPT

Gli operatori insiemistici si possono utilizzare solo al livello più esterno di una query, operando sul risultato di due o più clausole `SELECT`.
Gli operatori insiemistici sono: `UNION`, `INTERSECT` e `EXCEPT`.
Si possono avere sequenze di `UNION`/`INTERSECT`/`EXCEPT`

$\text{query}_1$ { UNION or INTERSECT or EXCEPT } [ ALL ] $\text{query}_2$

- Gli operatori si possono applicare solo quando $\text{query}_1$ e $\text{query}_2$ producono risultati con lo stesso numero di colonne e di tipo compatibile fra loro.
- Tutti gli operatori eliminano i duplicati dal risultato a meno che `ALL` non sia stato specificato.
- `UNION` aggiunge il risultato di $\text{query}_2$ a quello di $\text{query}_1$ .
- `INTERSECT` restituisce le righe che sono presenti sia nel risultato di $\text{query}_1$ sia in quello di $\text{query}_2$ .

- `EXCEPT` restituisce le righe di $\text{query}_1$ che non sono presenti nel risultato di $\text{query}_2$ . In pratica esegue la differenza insiemistica.

**Esempi:**
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%';
```

Visualizzare i nomi degli insegnamenti che sono anche nomi di corsi di laurea.

```
SELECT nomeins
FROM Insegn

INTERSECT ALL

SELECT nome
FROM CorsoStudi;
```

Visualizzare i nomi degli insegnamenti che NON sono anche nomi di corsi di laurea.

```
SELECT nomeins
FROM Insegn

EXCEPT

SELECT nome
FROM CorsoStudi ;
```

### 1.9.15 Viste

- Le viste sono tabelle "virtuali" il cui contenuto dipende dal contenuto delle altre tabelle della base di dati.
- In SQL le viste vengono definite associando un nome ed una lista di attributi al risultato dell'esecuzione di un'interrogazione.
- Ogni volta che si usa una vista, si esegue la query che la definisce.
- Nell'interrogazione che definisce la vista possono comparire anche altre viste.
- SQL non ammette però:
  - dipendenze immediate (definire una vista in termini di se stessa) o ricorsive (definire una interrogazione di base e una interrogazione ricorsiva);
  - dipendenze transitive circolari ($V_1$ definita usando $V_2$, $V_2$ usando $V_3$, ..., $V_n$ usando $V_1$).

```
CREATE [ TEMP ] VIEW nome [ (col_name [ , ...]) ] AS query
```

- `TEMP` : la vista è temporanea. Quando ci si sconnette, la vista viene distrutta. È un'estensione di PostgreSQL. Nella base di dati `did2014` si possono fare solo viste temporanee.

- `column_name` : nomi delle colonne che compongono la vista. Se non si specificano, si ereditano dalla query.
- `query` deve restituire un insieme di attributi pari e nel medesimo ordine a quello specificato con `(column_name [, ...])` se presente.

**Esempi:**
Definire la vista che contiene gli insegnamenti erogati completi di nomeins e codiceins presi dalla tabella Insegn .

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

Si vuole determinare qual è il corso di studi con il massimo numero di insegnamenti (esclusi i moduli). Prima si crea una vista:

```
CREATE TEMP VIEW InsCorsoStudi(Nome, NumIns) AS
  SELECT CS.nome, COUNT(*)
  FROM CorsoStudi CS JOIN InsErogato IE
       ON CS.id = IE.id_corsostudi
  WHERE IE.modulo = 0
  GROUP BY CS.nome;
```

Poi la si usa:

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

**NOTA:** *Non è possibile usare due operatori di aggregazione in cascata!*

# 2 Indici

Gli indici sono strutture dati che permettono di accedere ad una tabella dati in maniera più efficiente. Dato che un indice è completamente scorrelato dalla tabella dati a cui si riferisce, deve sempre essere mantenuto aggiornato in base al contenuto della tabella cui si riferisce. Il costo dell'aggiornamento di un indice può essere significativo quando ci sono molti indici definiti sulla base di dati, per cui è bene usarli con saggezza ed applicarli nella maniera più efficiente possibile.

Una buona regola pratica per l'uso di indici, dal momento che costano tempo e memoria, è di applicarli in base alle query eseguite più frequentemente, tenendo anche presente che il sistema deve aggiornare l'indice per ogni operazione `INSERT`, `DELETE` e `UPDATE`.

## 2.1 Comando timing

Il comando `\timing` da un 'idea del tempo necessario all'esecuzione di una query. In un prompt di `psql` basta eseguire:

```
=> \timing
=> select * from tabella;
```

## 2.2 Comando CREATE INDEX

```
CREATE [ UNIQUE ] INDEX [nome]
ON tabella [ USING method ]
({ nomeAttr | (expression)  [ ASC | DESC ] [ , ...])
```

dove:

- `method` è il tipo di indice;
- `nomeAttr` o `expression` indicano su quali colonne o espressioni con colonne si deve creare l'indice;
- `ASC`/`DESC` indica se l'attributo è ascendente o discendente;
- `ALTER INDEX` e `DROP INDEX` permettono di modificare o rimuovere gli indici.

Una volta creato, l'indice è usato dal sistema ogni volta che l'ottimizzatore di query lo ritiene opportuno, ovvero solo quando il vantaggio derivato è di una certa consistenza. Un indice può anche essere utilizzato per ottimizzare l'esecuzione di `UPDATE` e `DELETE`, se nella clausola `WHERE` ci sono attributi indicizzati.

**Nota: PostgreSQL crea in automatico indici per gli attributi dichiarati come chiave primaria, quindi è inutile indicizzarli.**

## 2.3 Comando ANALYZE

Il comando `ANALYZE` è usato per forzare l'aggiornamento delle statistiche di esecuzione delle query quando uno o più nuovi indici sono creati.

```
CREATE INDEX ie_id_corsostudi ON Inserogato (id_corsostudi) ;
CREATE INDEX ie_id_insegn ON Inserogato (id_insegn);
CREATE INDEX ie_aa_C ON Inserogato (annoaccademico);
CREATE INDEX ie_aa_IT ON Inserogato (annoaccademico varchar_pattern_ops);
CREATE INDEX cs_nome ON Corsostudi (nome varchar_pattern_ops);
ANALYZE;
```

## 2.4 Tipi di indici

PostgreSQL ammette molti tipi di indice, tra i quali: **B-tree, hash, GiST, SP-GiST, Gin, Brin**. Ognuno di questi tipi usa una tecnica algoritmica diversa e risulta migliore di altri in alcune situazioni (vedi 2.2 per la specifica del tipo di indice da creare). Se il tipo di indice non è specificato, viene creato un indice di tipo B-tree.

Nello specifico caso dell'indice B-tree, questo viene utilizzato quando l'attributo coinvolto è usato con gli operatori di confronto di valore o con i comandi `BETWEEN`, `IN`, `IS NULL`, `IS NOT NULL` e `LIKE`. La keyword **varchar_pattern_ops** è inserita quando si vuole che l'indice consideri anche i pattern del tipo `LIKE 'stringa%'`.

## 2.5 Indici multi-attributo

Se si hanno query con condizioni su coppie o terne, a volte può essere più efficiente l'uso di un indice dichiarato su due attributi rispetto a due indici mono-attributo.

Ad esempio, con una query come:

```
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;
```

che controlla gli attributi `annoaccademico` e `id_corsostudi`, è utile usare un indice multi-attributo che indicizzi i due campi:

```
CREATE INDEX ie_aa_idcs ON Inserogato (annoaccademico, id_corsostudi);
```

Non sempre gli indici multi-attributo possono essere usati, come nel caso di espressioni con `OR`.

## 2.6 Comando EXPLAIN

Ogni DBMS ha un ottimizzatore di query che determina un piano di esecuzione per ogni query. Il comando `EXPLAIN [query]` permette di vedere il piano di esecuzione della query, facilitando l'analisi dei colli di bottiglia e l'ottimizzazione.

Un piano di esecuzione di una query è un albero di nodi di esecuzione, dove le foglie sono **nodi di scansione**, che restituiscono gli indirizzi di righe della tabella. I possibili tipi di scansione sono 3: sequenziali, indicizzate e bit-mapped. Se una query contiene `JOIN`, `GROUP BY`, `ORDER BY` o altre operazioni sulle righe, allora ci saranno altri nodi di esecuzione sopra i nodi foglia.

L'output del comando ha una riga per ogni nodo dell'albero di esecuzione dove viene indicato il tipo di operazione e una stima del costo di esecuzione. La prima riga contiene il costo complessivo della query.

**Esempio 1.** Considerare questa query:

```
EXPLAIN SELECT * FROM Insegn ;
```

Il piano corrispondente è:

```
Seq Scan ON insegn (cost=0.0..185.69 ROWS=8169 width=63)
```

ed ha un costo di 185.69 (in termini di accesso a disco), con un risultato stimato di 8169 righe (il numero potrebbe non essere assolutamente preciso).

**Esempio 2.** Considerare questa query:

```
EXPLAIN SELECT * FROM Insegn WHERE id < 1000;
```

Il piano corrispondente è:

```
Bitmap Heap Scan ON insegn ( cost=18. 60..13 2.79 ROWS=815...)
Recheck Cond : ( id < 1000)
-> Bitmap INDEX Scan ON insegn_pkey(cost=0..18.39 ROWS=815 width=0)
    INDEX Cond : ( id < 1000)
```

Prima viene eseguito il nodo foglia `Bitmap Index Scan` che, grazie all'indice, permette di ritornare un vettore di bit che marca le righe da considerare. Il vettore viene poi passato al nodo padre `Bitmap Heap Scan`, che esegue la selezione delle righe che hanno $id < 1000$.

**Esempio 3.** Considerare questa query:

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

Il piano corrispondente è:

```
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..)
```

`Merge JOIN` esegue il join ordinando le due tabelle rispetto agli attributi di join.

Esiste una variante estesa di `EXPLAIN`, `EXPAIN ANALYZE`, che esegue la query senza registrare le modifiche e mostra una stima verosimile dei tempi di esecuzione.

# 3 Transazioni concorrenti

Una transazione SQL è una sequenza di istruzioni ed è eseguita in maniera atomica. Gli stati intermedi della base di dati durante l'esecuzione della serie di istruzioni della transazione non sono visibili al di fuori della transazione stessa. Se una transazione termina senza errori, le modifiche vengono salvate, in caso contrario lo stato della base di dati rimane quello presente prima dell'inizio della transazione e non viene salvata nessuna modifica.

A volte può capitare che le transazioni accedano in modo concorrente alle stesse informazioni nella base di dati, e può non essere garantito il corretto svolgimento delle operazioni; in questo caso vengono impostati dei livelli di isolamento della transazione rispetto alle altre, con diversi effetti sull'accesso concorrente ai dati.

In PostgreSQL i livelli di isolamento sono 4:

- Read Committed
- Repeatable Read
- Serializable
- Read Uncommitted

Prendiamo come esempio la seguente tabella: `Web`:

| id | hits |
|---|---|
| 1 | 9 |
| 2 | 10 |

## 3.1 Read Committed

È il **livello di default**, basta scrivere `BEGIN;`

- `SELECT` vede solo i dati registrati (`COMMITTED`) in altre transazioni e quelli modificati da comandi precedenti nella stessa transazione.
- `UPDATE` e `DELETE` vedono i dati come `SELECT`
- Se i dati da aggiornare sono stati modificati ma **non registrati** in transazioni concorrenti, il comando deve:
  - Attendere il `COMMIT` o `ROLLBACK` della transazione concorrente.
  - Riesaminare le righe per verificare che soddisfano ancora i criteri del comando.

**Esempio: Non-repeatable reads**

$T_1$:

```
BEGIN;
UPDATE Web SET hits=hits+1;

COMMIT;
```

$T_2$:

```
BEGIN;

DELETE FROM Web WHERE hits=10;
```

`DELETE` non riesce a cancellare: la riga è quella con `id = 2`, che viene aggiornata da `UPDATE` il quale sblocca `DELETE` fino al `COMMIT`. `DELETE` riesamina la riga e trova che il criterio non è più soddisfatto.

## 3.2 Repeatable Read

Differisce da Read Committed per il fatto che i comando di una transazione **vedono sempre gli stessi dati**. Due `SELECT` identiche vedono sempre gli stessi dati.

- `UPDATE` e `DELETE` vedono i dati come `SELECT`.

- Se i dati da aggiornare sono stati modificati ma **non registrati** in transazioni concorrenti, il comando deve attendere:
  - il `COMMIT` e quindi i dati vengono cambiati e `UPDATE` e `DELETE` vengono bloccate con errore.
  - il `ROLLBACK` e quindi `UPDATE` e `DELETE` possono procedere.

**Esempio1: Cattura l'anomalia Non-repeatable reads**

$T_1$:

```
BEGIN TRANSACTION ISOLATION LEVEL
  REPEATABLE READ;

UPDATE Web SET hits=hits+1;

COMMIT;
```

$T_2$:

```
BEGIN;

DELETE FROM Web WHERE hits=10;

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

**Esempio2: Phantom Reads**

$T_1$:

```
BEGIN TRANSACTION ISOLATION LEVEL
  REPEATABLE READ;

INSERT INTO Web SELECT
  MAX(hits)+1 FROM Web;

COMMIT;
```

$T_2$:

```
BEGIN TRANSACTION ISOLATION LEVEL
  REPEATABLE READ;

INSERT INTO Web SELECT
  MAX(hits)+1 FROM Web;

COMMIT;
```

La tabella finale contiene due valori 11 e non 11 e 12 (inserimento fantasma).

## 3.3 Serializable

È **il più restrittivo** e garantisce che le transazioni siano eseguite **come se fossero sequenziali tra loro** (in un ordine non prestabilito). Si deve però prevedere la possibilità di transazioni abortite più frequenti per gli aggiornamenti concorrenti tipo Repeatable Read.
**Esempio:**

$T_1$:

```
BEGIN TRANSACTION ISOLATION LEVEL
  SERIALIZABLE;

INSERT INTO Web SELECT
  MAX(hits)+1 FROM Web;

COMMIT;
```

$T_2$:

```
BEGIN TRANSACTION ISOLATION LEVEL
  SERIALIZABLE;

INSERT INTO Web SELECT
  MAX(hits)+1 FROM Web;

COMMIT;
```

La $T_2$ riporta il seguente errore:

```
ERROR: could not serialize access due to READ/WRITE dependencies
       among transactions
DETAIL: reason code: canceled ON identification AS a pivot,
        during COMMIT attempt
HINT: the TRANSACTION might succeed if retried.
```

# 4 Psycopg2

*psycopg2* è una libreria Python che consente di collegarsi a database SQL ed eseguire statement di vario genere, il tutto da codice Python ad alto livello.

Le operazioni da svolgere per collegarsi ed effettuare le operazioni sono:

- Collegarsi al database:

```
def connect():
    conn = psycopg2.connect(host=[host],
        database=[db-name], user=[username], password=[password])
    return conn
```

  `conn` è un'istanza della classe `Connection`.

- Ottenere un cursore dalla connessione: un cursore è come un buffer che contiene temporaneamente i dati delle operazioni svolte e da svolgere

```
cursor=conn.cursor()
```

- Eseguire le operazioni da svolgere con le seguenti chiamate:

```
cursor.execute([statement],[params])
conn.commit()
```

  dove `statement` è lo statement da eseguire sulla base di dati, mentre `commit()` esegue effettivamente l'operazione. `params` è una o più tuple (o dizionario) che contengono i dati da inserire. Nella stringa dello statement i parametri da sostituire vanno rimpiazzati con dei segnaposto `%s`. La libreria si occuperà di fare tutte le conversioni, quindi non servono cast.

- Ottenere i risultati da elaborare con la chiamata di:

```
conn.fetchone() # legge una sola riga
conn.fetchmany([numero]) # legge [numero] righe
conn.fetchall() # legge tutte le righe
```

  Se non ci sono tuple, viene ritornato `None` nei primi due casi (`numero` è il numero di tuple da leggere dal risultato), una lista vuota nel terzo caso.

- Chiudere le risorse con `conn.close()` e `cursor.close()`.

In alternativa alla procedura appena illustrata si può usare il costrutto `with` con la seguente sintassi:

```
with conn.cursor() as cur:
    cur.execute([statement],[params])
    ...
```

In questo modo **non serve il commit**, in quanto viene fatto automaticamente al termine dell'esecuzione del corpo del `with`, la connessione invece va chiusa manualmente.

Un esempio di come venga usata questa liberia si può trovare nella sezione dedicata a Flask.

# 5 Flask

## 5.1 HTTP requests:

- `GET`: serve ad un client per **recuperare** una risorsa dal server (come la richiesta di una pagin web). Eventuali parametri da inviare al server sono specificati nella **query string** dell'URL.
- `POST`: serve ad un client per **inviare** informazioni al server. La maggior parte dei browser usa post per **inviare dati delle form ai server**. I dati sono specificati nel corpo della richiesta.

## 5.2 Accesso ai parametri della query string di una richiesta GET

**Query String** rappresentata dalla variabile `request.args` di tipo `dict`, accessibile direttamente dal metodo associato all'URL:

```
from flask import Flask, request
app = Flask(__name__)

@app.route('/login')
def login():
    user = request.args['user']
    role = request.args['role']
    return ...
```

## 5.3 Accesso ai parametri della query string di una richiesta POST

Listing 1: esempio di form html5

```
<form action="http://localhost:5000/login" method="post">
    <label> Name: <input type="text" name="user"/> </label><br>
    <label> Role: <input type="text" name="role"/> </label><br>
    <input type="submit" value="Invia">
</form>
```

I dati di un `POST` sono nel dict `request.form`

```
@app.route('/login')
def login():
    user = request.form['user']
    role = request.form['role']
    return ...
```

Il metodo `route()` associa un metodo a un URL in modalità `GET`, per usare lo stesso sistema in modalità `POST`, è necessario specificare esplicitamente i metodi che vengono utilizzati:

```
@app.route('/',method=['GET','POST'])
```

## 5.4 Esempio esercizio d'esame con integrazione HTML

Dall'esame del 04/07/2017:
Assumendo di avere una base di dati PostgreSQL che contenga le tabelle di questo tema d'esame, scrivere:

(a) Un template JINJA2 per una form HTML 5 che: (1) permetta di acquisire un codice fiscale (controllando il formato), (2) di selezionare una biblioteca dalla lista `biblioteche` passata come parametro al template e (3) invii i dati all'URL `/prestitiUtente` in modalità GET. Il formato di `biblioteche` è `[{id, nome}, ...]`.
Scrivere solo la parte della FORM, non tutto il documento HTML.

(b) Un metodo Python che, associato all'URL `/prestitiUtente` secondo il framework Flask, (1) legga i parametri codice fiscale e identificatore biblioteca, (2) si connetta alla base di dati 'X' (si assuma di dover specificare solo il nome della base di dati) e recuperi tutti i prestiti (idRisorsa, dataInizio, durata) associati al codice fiscale e biblioteca dati come parametri (scrivere la query!), (3) usi il metodo `render_template('view.html', ...)` per pubblicare il risultato passando la lista del risultato. Se il risultato dell'interrogazione è vuoto, il metodo deve passare il controllo a `render_template('nessunPrestitoOErrore.html')`.
Scrivere solo il metodo.

Soluzione (a):

```
<form action="/prestitiUtente" method="get">
  <label for="codicef">Codice fiscale: </label>
    <input id="codicef" name="cf" type="text" pattern="[A-Z]...">
  <br>
  <label for="biblioteca">Biblioteca: </label>
    <select id="biblioteca" name="biblio">
      {% for b in biblio %}
      <option value="{{b.id}}"> {{b.nome}} </option>
      {% endfor %}
    </select>
  <input type="submit" value="Invia">
</form>
```

Soluzione (b):

```
@app.route('/prestitiUtente', methods= ['GET'])
def getPrestiti():
  cf = request.args['cf']
  biblio = request.args['biblio']

  with psycopg2.connect(database='X') as conn:
    with conn.cursor as cur:
      cur.execute("SELECT P.idRisorsa, P.dataInizio, P.durata\
        FROM Prestito P WHERE P.idUtente = %s AND\
        P.idBiblioteca = %s", cf, int(biblio))
      prestiti = cur.fetchall()

      if not prestiti:
        return render_template('nessunPrestitoOErrore.html')

      return render_template('view.html', prestiti=prestiti,
        cf=cf, biblio=biblio)
```

## 5.5 Un altro esempio

Qui di seguito un altro esempio di applicazione scritta con Flask, pensata per la gestione delle spese.
File `controller.py`:

```
from datetime import datetime, date
from decimal import Decimal
from flask import *
import psycopg2

app = Flask(__name__)

HOST = [nome-host]
DATABASE = [nome-db]
USER =[username]


def connect():
  connection = psycopg2.connect(host=HOST, database=DATABASE,
    user=USER, password=[password])
  return connection


def get_cursor(connection):
  return connection.cursor()


def insert_data(tup):
  conn = connect()
  cursor = conn.cursor()
  cursor.execute('insert into Spese(date, description, import) values
    (%s, %s, %s)', tup)
  conn.commit()
  conn.close()

def remove_data(tup):
  conn = connect()
  cursor = conn.cursor()
  cursor.execute('delete from Spese where date=%s and description=%s
     and import=%s ', tup)
  conn.commit()
  conn.close()


@app.route('/', methods=['POST', 'GET'])
def fill_table():
  connection = connect()
  cursor = get_cursor(connection)
  cursor.execute('select date, description, import from Spese')
  connection.commit()
  table = cursor.fetchall()
  connection.close()

  if request.method == 'POST':
    if request.form['submit'] == 'Add entry':
      return redirect(url_for('new_entry'))
```

```
    elif request.form['submit'] == 'Remove entry':
      return redirect(url_for('remove_entry'))

  return render_template('main_table.html', table=table)


@app.route('/new_entry', methods=['POST', 'GET'])
def new_entry():
  if request.method == 'POST':
    if request.form['submit'] == 'Confirm':
      purchase_date = datetime.strptime(request.form['date'],
          '%d/%m/%Y')
      price = float(request.form['cost'])
      descr = request.form['descr']
      insert_data((date(purchase_date.year,
        purchase_date.month, purchase_date.day), descr,
         Decimal(price)))
      return redirect(url_for('fill_table'))

  return render_template('new_entry.html')


@app.route('/remove_entry', methods=['POST', 'GET'])
def remove_entry():
  if request.method == 'POST':
    if request.form['submit'] == 'Delete':
      purchase_date = datetime.strptime(request.form['date'],
          '%d/%m/%Y')
      price = float(request.form['cost'])
      descr = request.form['descr']
      remove_data((date(purchase_date.year,
        purchase_date.month, purchase_date.day), descr,
         Decimal(price)))
      return redirect(url_for('fill_table'))

  return render_template('remove_entry.html')


if __name__ == '__main__':
app.run()
```

Qui di seguito le pagine html create e utilizzate (puramente funzionali, non hanno nulla di estetico!).
File `main_table.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <h1>Gestione spese</h1>
</head>
<body>
<table border="1" cellpadding="5" cellspacing="5" width="500">
    <tr>
    <th>Data</th>
    <th>Descrizione</th>
    <th>Importo</th>
    </tr>
```

```
    {% for entry in table %}
        <tr>
        <td>{{entry[0]}}</td>
        <td>{{entry[1]}}</td>
        <td>{{entry[2]}}</td>
        </tr>
    {% endfor %}
</table>

<form method="post">
<input type="submit" name="submit" value="Add entry"/>
<input type="submit" name="submit" value="Remove entry"/>
</form>
</body>
</html>
```

File `new_entry.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <h1>Add new entry</h1>
</head>
<body>
<form method="post">
    Data:<input name="date" pattern="[0-9]{2}/[0-9]{2}/[0-9]{4}"/> <br>
    Importo:<input name="cost"/> <br>
    Descrizione:<input name="descr"/> <br>

    <input type="submit" name="submit" value="Confirm">
</form>
</body>
</html>
```

File `remove_entry.html`

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <h1>Remove entry</h1>
</head>
<body>
<form method="post">
    Data:<input name="date" pattern="[0-9]{2}/[0-9]{2}/[0-9]{4}"/> <br>
    Importo:<input name="cost"/> <br>
    Descrizione:<input name="descr"/> <br>

    <input type="submit" name="submit" value="Delete">
</form>
</body>
</html>
```

# 6 JDBC

JDBC è una libreria di funzioni Java che consente di collegarsi ad un database ed eseguire operazioni. Le fasi principali per eseguire una qualsiasi operazione sono:

1. Caricare il driver `org.postgresql.Driver`;
2. Collegarsi al database sul quale si intende operare;
3. Creare gli statement da eseguire;
4. Eseguire gli statement e il commit;
5. Chiudere le risorse utilizzate.

## 6.1 Caricamento del driver

Questa fase consente di caricare i moduli che servono per collegarsi al database. Si effettua eseguendo l'opportuna chiamata:

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

## 6.2 Collegamento al database

Per collegarsi al database si effettua una chiamata che, passando gli opportuni parametri, ritorna un oggetto di tipo `Connection`, che verrà usato per eseguire gli statement nelle prossime fasi.

```
try {
    Connection connection = DriverManager.getConnection(
        [url], [utente], [password]);
} catch (SQLException e) {
    System.out.println(e.getMessage());
}
```

## 6.3 Creazione ed esecuzione di statement

Gli statement sono le istruzioni SQL da eseguire sul database. Per statement semplici si può usare questo snippet:

```
try {
    Statement ps = connection.createStatement();
    ResultSet rs = ps.executeQuery("select * from Spese");
} catch (SQLException e) {
    e.printStackTrace();
}
```

dove `connection` è un'istanza della classe `Connection`. Se lo statement è un comando di aggiornamento (`insert`, `update`) si usa il metodo `executeUpdate` o `executeLargeUpdate` il quale *ritorna il **numero di righe che sono state modificate***.

Se gli statement sono complessi e hanno magari una clausola `where` con vari confronti, bisogna prima creare un oggetto `PreparedStatement`, con la seguente sintassi:

```
try {
    PreparedStatement ps = connection.prepareStatement(
        "insert into Spese(data, voce, importo) values(?, ?, ?)");
    ps.setDate(1, date);
    ps.setString(2, descr);
    ps.setFloat(3, price);
    ps.executeUpdate();
} catch (SQLException e) {
```

```
    e.printStackTrace();
}
```

dove `connection` è un'istanza della classe `Connection`. I punti interrogativi sono usati come segnaposto da rimpiazzare con con i metodi `setDate()`, `setString()` ecc. I metodi set funzionano passando come parametro un indice (da 1), e il valore che si vuole sostituire nella query. In questo modo i punti interrogativi saranno rimpiazzati dai campi passati nei metodi set.

**Nota: le connessioni sono auto-commit, il commit viene eseguito al termine dell'esecuzione di ogni comando.**

## 6.4 Accesso ai campi e chiusura risorse

**Accesso ai dati.** La query viene eseguita con il metodo `executeQuery()` che ritorna un oggetto di tipo `ResultSet` il quale contiene lo stato e l'eventuale tabella risultato, alla quale si può accedere tramite metodi get. Supponendo di aver eseguito la query:

```
...
ResultSet rs = ps.executeQuery("select * from Spese");
...
```

si accede ai dati contenuti in `rs` tramite chiamate del tipo:

```
rs.get[tipoDato]([nomeCampo]);
```

Ad esempio, avendo l'attributo `Descrizione` nella tabella `Spese` di tipo `varchar`:

```
rs.getString("descrizione");
```

**Chiusura risorse.** Quando gli statement da eseguire sono terminati, va chiusa la connessione utilizzata con:

```
connection.close();
```

## 6.5 Modulo JDBC completo

Qui di seguito viene inserito un modulo jdbc per il controllo della tabella `Spese` utilizzata negli esempi precedenti.

```
import java.sql.*;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Collections;
import java.util.Scanner;

public class Main {
  public static void main(String args[]) {
    printMenu();
    callFunction();

  }

  private static void displayData() {
    Connection connection = getConnection();
    try {
      Statement ps = connection.createStatement();
      ResultSet rs = ps.executeQuery("select * from Spese");
    printData(rs);
    } catch (SQLException e) {
```

```
      e.printStackTrace();
    }
  }

  private static void insertData() {
    Connection connection = getConnection();
    Scanner scan = new Scanner(System.in);
    System.out.print("Data:");
    String dateString = scan.next();
    SimpleDateFormat format = new SimpleDateFormat("dd/MM/yyyy");
    Date date = null;
    try {
      date = new Date(format.parse(dateString).getTime());
    } catch (ParseException e) {
      System.out.println(e.getMessage());
    }
    System.out.print("Prezzo:");
    float price = Float.parseFloat(scan.next());

    System.out.print("Descrizione:");
    String descr = scan.next();

    try {
      PreparedStatement ps = connection.prepareStatement(
      "insert into Spese(data, voce, importo) values(?, ?, ?)");
      ps.setDate(1, date);
      ps.setString(2, descr);
      ps.setFloat(3, price);
      ps.executeUpdate();
      System.out.println("Update completed.");
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }

  private static void removeData() {
    Connection connection = getConnection();
    Scanner scan = new Scanner(System.in);
    System.out.print("Data:");
    String dateString = scan.next();
    SimpleDateFormat format = new SimpleDateFormat("dd/MM/yyyy");
    Date date = null;
    try {
      date = new Date(format.parse(dateString).getTime());
    } catch (ParseException e) {
      System.out.println(e.getMessage());
    }
    System.out.print("Prezzo:");
    float price = Float.parseFloat(scan.next());

    System.out.print("Descrizione:");
    String descr = scan.next();

    try {
      PreparedStatement ps = connection.prepareStatement(
      "delete from Spese where data=? and voce=? and importo=?");
```

```
      ps.setDate(1, date);
      ps.setString(2, descr);
      ps.setFloat(3, price);
      ps.executeUpdate();
      System.out.println("Update completed.");
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }

  private static void printData(ResultSet rs) {
    try {
      System.out.println(
      String.join("", Collections.nCopies(50, "=")));
      SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
      while (rs.next()) {
        System.out.println(
        String.format("| %2s | %10s | %-20s | %10.2f |",
         rs.getInt("id"), sdf.format(rs.getDate("data")),
         rs.getString("voce"), rs.getFloat("importo")));
      }
      System.out.println(String.join("",
          Collections.nCopies(50, "=")));

    } catch (SQLException e) {
      e.printStackTrace();
    }
  }

  private static void printMenu() {
    System.out.println(String.join("", Collections.nCopies(50, "=")));
    System.out.println("1. Display data");
    System.out.println("2. Insert new outlay");
    System.out.println("3. Remove outlay");
    System.out.println(String.join("", Collections.nCopies(50, "=")));
  }

  private static Connection getConnection() {
    try {
      Class.forName("org.postgresql.Driver");
      return DriverManager.getConnection(
      "jdbc:postgresql://dbserver.scienze.univr.it:5432/id864ghl",
      "id864ghl", "perzona-falza");
    } catch (ClassNotFoundException | SQLException e) {
      System.out.println(e.getMessage());
    }

  return null;
  }

  private static void callFunction() {
    Scanner scan = new Scanner(System.in);
    System.out.print("Choose action:");
    int option = Integer.parseInt(scan.next());
    if (option == 0) {
      System.exit(0);
```

```
    } else if (option == 1) {
      displayData();
    } else if (option == 2) {
      insertData();
    } else if (option == 3) {
      removeData();
    } else {
      System.out.println("Invalid option");
    }
  }
}
```

# 7 Credits

Davide Bianchi (mail: davideb1912@gmail.com)
Matteo Danzi (mail: matteodanziguitarman@hotmail.it)