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 esatti, 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)`: equivalente a `NUMERIC`;
- `REAL`: valori in virgola mobile approssimati 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` e `DECIMAL` (non approssimano la parte decimale)!

1.1.2 Domini di caratteri

- `CHAR`/`CHARACTER`: singoli caratteri;
- `CHAR(n)`/`CHARACTER(n)`:stringa di caratteri di lunghezza n;
- `VARCHAR(n)`: stringhe di caratteri di lunghezza variabile con limite n;
- `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;
- `TIMESTAMP`: corrispondente a `DATE` + `TIME`;
- `TIME/TIMESTAMP WITH TIME ZONE`: aggiunta di indicazioni sul fuso orario.

```
DATE : '2016-01-15'
TIME(3) : '04:05:06.789'
INTERVAL : '3 hours 25 minutes'
TIME WITH TIME ZONE : '04:05:06-08:00' o '12:01:01 CET'
TIMESTAMP WITH TIME ZONE : '2016-01-24 00:00:00+01'
```

Funzioni e operazioni: `https://www.postgresql.org/docs/9.1/functions-datetime.html`

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.

```
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 i vincoli `NOT NULL` e `UNIQUE`.

  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. Un vincolo `CHECK` è soddisfatto se la sua espressione è vera o nulla.
  In molti casi, un'espressione è nulla se uno degli operandi è nullo. Conviene quindi mettere sempre `NOT NULL` insieme al vincolo `CHECK()`!

```
CREATE TABLE Impiegato (
    ...
    stipendio NUMERIC (8 ,2) DEFAULT 500.00 NOT NULL
        CHECK ( stipendio >= 0.0) , -- check di attributo
    UNIQUE ( cognome , nome ) ,
    CHECK ( nome <> cognome ) -- check di tabella
);
```

1.3.1 Vincoli di integrità referenziale

Un vincolo di integrità referenziale crea un legame tra i valori di un attributo (o di un insieme di attributi) A della tabella corrente (detta interna/slave) e i valori di un attributo (o di un insieme di attributi) B di un'altra tabella (detta esterna/master):

- Impone che, in ogni tupla della tabella interna, il valore di A, se diverso dal valore nullo, sia presente tra i valori di B nella tabella esterna.
- L'attributo B della tabella esterna deve essere soggetto a un vincolo `UNIQUE` o `PRIMARY KEY`.

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';
```

In assenza di una condizione vengono eliminate tutte le tuple della tabella.

1.8 Comando DROP TABLE

Una tabella viene cancellata con il comando `DROP TABLE`.

```
DROP TABLE impiegato;
```

1.9 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.

Si possono aggiungere/rimuovere anche dopo la creazione della tabella:

- Per aggiungere:

  ```
  ALTER TABLE nome_tabella ADD [ CONSTRAINT nome_vincolo ]
      CHECK (def_vincolo);
  ```

  dove `def_vincolo` = dichiarazione vincolo di tabella.

  Esempio:

  ```
  ALTER TABLE museo ADD CONSTRAINT cons_prezzo
      CHECK (costo > 0.0);
  ```

  Se non si definisce un nome del constraint, il DBMS ne assegna uno.

- Per rimovere:

  ```
  ALTER TABLE nome_tabella DROP CONSTRAINT nome_vincolo;
  ```

  dove `nome_vincolo` è il nome scelto durante la dichiarazione (o definito dal DBMS).

1.10 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.

1.10.1 Operatore LIKE e SIMILAR TO

Nella clausola `WHERE` può apparire l'operatore `LIKE` per il confronto di stringhe (`ILIKE` se il confronto è case-insensitive). `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}` = ripetizione del precedente match n volte (**nè più nè meno**).
- `{n,m}` = ripetizione del precedente match almeno n e non più di m volte.
- `[...]` = ... è un elenco di caratteri ammissibili. Con '-' si possono usare intervalli (ad esempio con `[A-Z]` si selezionano tutte le lettere maiuscole dell'alfabeto).

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';
```

1.10.2 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';
```

1.10.3 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');
```

1.10.4 Operatore IS NULL

Nella clausola `WHERE` può apparire l'operatore `IS [ NOT ] 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!

1.10.5 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;
```

1.10.6 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.
Questi operatori si possono usare solo in `SELECT` e `HAVING`; il loro uso in una SELECT che richiede **altri valori**, oltre all'operatore in questione, necessita la presenza del campo `GROUP BY` (che NON può contenere operatori di aggregazione!) nella query (esempi più avanti).

L'operatore `COUNT` restituisce il numero di tuple significative nel risultato dell'interrogazione:

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

dove `expr` è un'espressione che usa attributi e funzioni di attributi ma non operatori di aggregazione (per quel caso è necessario l'uso di una vista).
Tre casi comuni:

- `COUNT(*)` ritorna il numero di tuple nel risultato dell'interrogazione.
- `COUNT([ALL] expr)` ritorna il numero di tuple in ciascuna delle quali il valore expr è non nullo.
- `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;
```

1.10.7 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 (=no duplicati) 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 (guardare il primo esempio per un confronto con `WHERE`).
- 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 (la query ritorna solo i singoli valori delle città che iniziano con 'V', mentre la stessa condizione in `WHERE` ritorna tutte le tuple che contengono queste città).

```
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;
```

N.B: **le funzioni di aggregazione non sono ammesse in WHERE, solo in HAVING!**

1.10.8 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 Θ 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`. `INNER` è opzionale.

```
SELECT I.cognome, R.nomeRep, R.sede
FROM Impiegato I 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`!

1.10.9 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** 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.

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.

Esempio: 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`.

Operatore IN

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

- `expr` è un'espressione costruita con un attributo della query principale. Ci possono essere 1 o più espressioni.
- `(subquery)` deve restituire un numero di colonne = numero di espressioni in `(expr [,...])`.
- I valori delle 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
);
```

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 '>='.
- Il confronto ritorna vero se `expression` è `operator` rispetto al valore di una qualsiasi riga del risultato di `(subquery)`.

`SOME` è un sinonimo di `ANY`.

Esempio: 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
);
```

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 '>='.
- Il confronto ritorna vero se `expression` è `operator` rispetto al valore di ciascuna riga del risultato di `(subquery)`.

Esempio: 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
);
```

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`

query_1 { `UNION or INTERSECT or EXCEPT` } [`ALL`] query_2

- Gli operatori si possono applicare solo quando query_1 e 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 query_2 a quello di query_1.
- `INTERSECT` restituisce le righe che sono presenti sia nel risultato di query_1 sia in quello di query_2.
- `EXCEPT` restituisce le righe di query_1 che non sono presenti nel risultato di 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 ;
```

1.11 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.

Esempio:
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`.

Visualizzare gli indici

Per visualizzare gli indici di una tabella si può utilizzare psql, l'applicativo front-end su terminale per interfacciarsi ai database Postgres. Per collegarsi si scrive:

```
psql -h dbserver database utente
```

Inserita la password richiesta si accederà al prompt di psql. Con il comando

```
=> \d nomeTabella
```

vengono riportati tutti i dettagli della tabella `nomeTabella`, tra i quali gli eventuali indici.

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;
```

2.1 Comando CREATE INDEX

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

dove:

- `UNIQUE` si usa se si vuole che l'indice imponga anche l'unicità dei valori dei dati;
- `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'indice è ordinato in modo 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.

Esempio:

```
CREATE INDEX nomeins_index ON Insegn(nomeins) ;
```

Nota: PostgreSQL crea in automatico indici (B-tree) per gli attributi dichiarati come chiave primaria, quindi è inutile indicizzarli.
Essendo indici B-tree, un indice (a,b,c,d) permette a query che filtrano ad esempio su (a), (a,b), o (a,b,c) di usare l'indice.

2.2 Comando ANALYZE

Il comando `ANALYZE [nomeTabella]` aggiorna le statistiche circa il contenuto delle tabelle (e indici), statistiche poi usate dall'ottimizzatore di query per decidere quando usare gli indici.
Con il comando `ANALYZE` dopo la creazione di uno o più nuovi indici si forza il DBMS ad aggiornare le statistiche di tutte le tabelle nel database corrente:

```
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;
```

2.3 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 2.1 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** abilita l'ottimizzatore a considerare l'indice anche quando ci sono pattern del tipo `LIKE 'stringa%'`.

2.4 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`.

2.5 Indici di espressioni

Le query con condizioni su espressioni/funzioni di uno o più attributi di una tabella possono essere velocizzate creando indici sulle medesime espressioni/funzioni.
Sintassi: `CREATE INDEX nome ON nomeTabella(expression);`
dove `expression` è una espressione su uno o più attributi. Come nel caso degli indici su attributi, si considera la creazione per espressioni che sono frequenti nelle interrogazioni usate.

Esempio. Indice sull'espressione `LOWER(nomeins)`, da usare anche con pattern matching:

```
CREATE INDEX ins_lower_nonins ON
    Insegn ( LOWER ( nomeins ) varchar_pattern_ops );
```

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)
```

- 0.0 è il costo iniziale, per produrre la prima riga
- 185.69 è il costo totale, per produrre tutte le righe
- 8169 è il numero totale di righe del risultato
- 63 è la dimensione, in byte, di ogni riga

Il costo è in termini di numero di accessi alla memoria secondaria (page disk). Il numero totale di righe non è sempre il numero totale di righe valutate dall'esecutore.

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 B-tree, permette di ritornare un vettore di bit che marca il sottoinsieme di righe da considerare (in un B-tree se la chiave cercata non è esplicita in un nodo, esiste un sottoalbero con chiavi maggiori e minori della chiave cercata). 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. `t1` è già ordinata via indice. `t2` non è ordinata, quindi c'è un nodo `Sort` .

Esempio 4. Considerare questa query:

```
EXPLAIN SELECT * FROM Inserogato WHERE id <1000 AND
    id_discriminante >100;
```

Il piano corrispondente è:

```
Bitmap Heap Scan ON Inserogato(cost=743.92..2178.36 ROWS=480..)
    Recheck Cond : (( id < 1000) AND ( id_discriminante > 100) )
    -> BitmapAnd (cost=743.92..743.92 ROWS=480 width=0)
        -> Bitmap INDEX Scan ON inserogato_pkey
                    (cost=0.00..18.57 ROWS=837 width=0)
           INDEX Cond : ( id < 1000)
        -> Bitmap INDEX Scan ON inserogato_discriminante_index
                    (cost=0.00..724.86 ROWS=39009 width=0)
           INDEX Cond : ( id_discriminante > 100)
```

- I 2 nodi foglia restituiscono un vettore di bit: 1 per ogni riga chè può soddisfare il criterio;
- Il nodo padre esegue l'AND dei due vettori di bit;
- Il nodo root recupera le righe e fa la selezione finale.

Esempio 5. Considerare questa query, leggermente diversa dalla precendente:

```
EXPLAIN SELECT * FROM Inserogato WHERE id <100 AND
    id_discriminante >100;
```

Il piano corrispondente è:

```
Bitmap Heap Scan ON Inserogato(cost=4.95..317.09 ROWS=50..)
    Recheck Cond : (id < 100)
    Filter: (id_discriminante > 100)
    ->  Bitmap INDEX Scan ON inserogato_pkey
                (cost=0.00..4.94 ROWS=86 width=0)
        INDEX Cond: (id < 100)
```

La condizione `id < 100` è così forte che conviene fare una lettura sull'indice della chiave e poi fare una selezione sulle righe indicate dal vettore di bit prodotto.

EXPLAIN ANALYZE

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.
Esempio: `https://robots.thoughtbot.com/reading-an-explain-analyze-query-plan`

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 Uncommitted (in PostgreSQL 9.6 implementato come Read Committed)
- Read Committed
- Repeatable Read
- Serializable

Durante l'esecuzione, in base al livello di isolamento si possono presentare delle anomalie, tra le quali:

- **Nonrepeatable reads**: la transazione legge un `UPDATE` di cui è stato fatto il commit in un'altra transazione. La stessa riga ha quindi diversi valori rispetto all'inizio della transazione;
- **Phantom reads**: la transazione legge un `INSERT` o un `DELETE` di cui è stato fatto il commit in un'altra transazione. Ci sono quindi nuove righe o righe mancanti rispetto all'inizio della transazione.

I prossimi esempi si basano sulla seguente tabella, `Web`:

id	hits
1	9
2	10

3.1 Read Committed

È il **livello di default**, basta scrivere '`BEGIN;`'. Nonrepeatable reads e Phantom reads possibili.

- `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`. Inoltre 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 soddisfino ancora i criteri del comando.

Esempio di anomalia Nonrepeatable 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, anche se esiste un hits=10 sia prima sia dopo l'update: questo perchè `SELECT`, che vede solo i valori registrati, ha saltato id=1 in fase di valutazione, perciò quando il `DELETE` verrà sbloccato, ricontrollerà solamente se il valore corrispondente a id=2 è ancora valido.

Esempio di anomalia Phantom reads:

T_1:

```
BEGIN;
SELECT id FROM Web WHERE hits=10;


SELECT id FROM Web WHERE hits=10;

COMMIT;
```

T_2:

```
BEGIN;

DELETE FROM Web WHERE hits=10;
COMMIT;
```

Le due SELECT della T_1 sono identiche ma danno due risultati diversi. Nessuna tupla è stata modificata nei suoi valori. Delle tuple sono state cancellate.

3.2 Repeatable Read

Differisce da Read Committed per il fatto che una query in una transazione Repeatable Read vede i dati come erano prima dell'inizio della transazione, che perciò non cambiano all'interno della transazione a causa di commits in altre transazioni, come nel caso di Read Committed. Due SELECT identiche all'interno di una singola transazione 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.

Esempio 1: Cattura dell'anomalia Nonrepeatable reads

T_1:

```


BEGIN;
UPDATE Web SET hits=hits+1;

COMMIT;
```

T_2:

```
BEGIN TRANSACTION ISOLATION LEVEL
  REPEATABLE READ;

DELETE FROM Web WHERE hits=10;

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

La T_2 legge l'UPDATE concorrente della T_1 e annulla.

Esempio 2: anomalia 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 Python e Database

La API ufficiale di Python è la DB-API v2.0, che descrive come un modulo Python deve accedere a una base di dati esterna. Esistono diversi moduli (librerie) DB-API, si utilizzerà **psycopg2**.

4.1 Connection

L'accesso ad un database avviene tramite un oggetto di tipo **Connection**. Il metodo **connect()** accetta i parametri necessari per la connessione e ritorna un oggetto *Connection*.
Metodi principali dell'oggetto:

- `cursor()`: ritorna un cursore della base di dati. Un oggetto cursore permette di inviare comandi SQL al DBMS.
- `commit()`: registra la transazione corrente. Normalmente una connessione apre una transazione al primo invio di comandi. Se non si esegue un *commit()* prima di chiudere, tutte le eventuali modifiche/inserimenti vengono persi.
- `rollback()`: abortisce la transazione corrente.
- `close()`: chiude la connessione corrente. Implica un *rollback()* automatico delle operazioni non registrate.
- `autocommit`: proprietà r/w. Se *True* , ogni comando inviato è una transazione isolata. Se *False* (default) il primo comando inviato inizia una transazione, che deve essere chiusa con *commit()* o *rollback()*.
- `readonly`: proprietà r/w. Se *True* , nella sessione non si possono inviare comandi di modifica dati. Il default è *False*.
- `isolation_level`: proprietà r/w. Modifica il livello di isolamento per la prossima transazione. Valori leciti: `'READ UNCOMMITTED'`, `'READ COMMITTED'`, `'REPEATABLE READ'`, `'SERIALIZABLE'` e `'DEFAULT'` (di solito corrispondente a `REPEATABLE READ`).
  Meglio assegnare questa variabile subito dopo la creazione della connessione.

```
connector = psycopg2.connect(...)
connector.isolation_level = 'REPEATABLE READ'
```

4.2 Cursor

Un **cursore** gestisce l'interazione con la base di dati: mediante un cursore è possibile inviare un comando SQL e accedere all'esito e ai dati di risposta del comando.
Metodi principali:

- `execute(comando, parametri)`: prepara ed esegue un *comando* SQL usando i *parametri*. I parametri devono essere passati come **tupla** o come **dict**.
  Il comando ritorna *None*. Eventuali risultati si devono recuperare con il `fetch*()`.

```
cur.execute("CREATE TABLE test (id SERIAL PRIMARY KEY, \
    num integer, data varchar)")
cur.execute("INSERT INTO test (num, data) VALUES (%s, %s)",
    (100, "abc'def"))   #psyconpg2 fa le conversioni!
```

- `executemany(comando, parametri)`: prepara ed esegue un *comando* SQL per ciascun valore presente nella lista *parametri*. Per come è attualmente implementato è meno efficiente di un unico insert con più tuple:

```
cur.execute("INSERT INTO test (num, data)
    VALUES (%s, %s), (%s, %s), (%s, %s)",
    (100, "abc'def", None, 'dada', 42, 'bar'))
```

- `fetchone()`: ritorna una tupla della tabella risultato.
  Si può usare dopo un `execute("SELECT ...")`. Se non ci sono tuple ritorna *None*.

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

- `fetchmany(<numero>)`: ritorna una lista di tuple della tabella risultato di lunghezza massima *<numero>*.
  Si può usare dopo un `execute("SELECT ...")`. Se non ci sono tuple ritorna una lista vuota.

```
>>> 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()`: ritorna l'intero risultato come lista di tuple. Se non ci sono tuple ritorna una lista vuota.

- Dopo un `execute("SELECT ...")`, il cursore è un iterabile sulla tabella risultato. È possibile quindi accedere alle tuple del risultato anche con un ciclo:

```
>>> 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'),
```

- `rowcount`: di sola lettura, uguale al numero di righe prodotte dall'ultimo comando. -1 indica che non è possibile determinare il valore.

- `statusmessage`: di sola lettura, uguale al messaggio ritornato dall'ultimo comando eseguito.

4.3 Particolarità di psycopg2

`cursor.execute*` accetta solo %s come indicatore di posizione parametro. La conversione dal tipo Python al dominio SQL è automatica per tutti i tipi fondamentali.
Esempio:

```
>>> cur.execute("INSERT INTO test1(id, date_val, item)
    VALUES (%s, %s, %s)", (42, datetime.date(2005, 11, 18),
    "O' Reilly ") )
```

è convertita in SQL:

```
INSERT INTO test1(id, date_val, item) VALUES
    (42, '2005-11-18', 'O''Reilly');
```

5 Uso di 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`.

- Eventualmente modificare le proprietà di livello di isolamento, autocommit, readonly.

- 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:

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

  Se non ci sono tuple, viene ritornato `None` nel primo caso, una lista vuota negli altri due casi. Altro metodo per leggere è attraverso l'uso di un ciclo for:

```
for record in cur :
    print(record)
```

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

La gestione dei **close** e dei **commit** è semplificata se si usa il costrutto **with**:

```
conn = psycopg2.connect(...)
with conn :
    with conn.cursor() as cur:
        cur.execute([statement],[params])
        ...
```

- Quando si usa una connessione con il with, all'uscita dal blocco viene fatto un **commit** automatico, mentre la connessione **non viene chiusa**.

- Quando si usa/crea un cursore con il with, all'uscita dal blocco viene fatto un **close** automatico del cursore (conviene usare sempre lo stesso cursore se possibile).

5.1 Connessioni e cursori

Si deve porre attenzione alla combinazione di cursori sulla medesima connessione:

- Aprire una connessione costa in tempo (e spazio). Meglio aprire/chiudere poche connessioni in un'esecuzione.
- Con un oggetto connessione si possono creare più cursori. Questi cursori condividono la connessione.
- Psycopg2 garantisce solo che le istruzioni inviate dai cursori vengono sequenzializzate, quindi non si possono gestire transazioni concorrenti usando diversi cursori sulla medesima connessione.
- Regola pratica: usare più cursori sulla medesima connessione quando si fanno transazioni in auto-commit o solo transazioni di sola lettura.

5.2 Esempio

```
from datetime import date
from decimal import Decimal
import psycopg2

conn = psycopg2.connect(host = "dbserver.scienze.univr.it", \
    database = "psnrrt07", user = "psnrrt07")

with conn :
    with conn.cursor() as cur :
        cur.execute("CREATE TABLE Spese(id SERIAL PRIMARY KEY, \
            data DATE, voce VARCHAR, importo NUMERIC)" )
        print("Esito creazione tabella : ", cur.statusmessage)
        cur.execute("INSERT INTO Spese(data, voce, importo) \
            VALUES (%s,%s,%s),(%s,%s,%s),(%s,%s,%s),(%s,%s,%s)", \
            (date(2016,2,24), "Stipendio", Decimal("0.1"), \
            date(2016,2,24), 'Stipendio "Bis"', Decimal("0.1"), \
            date(2016,2,24), 'Stipendio "Tris"', Decimal("0.1"), \
            date(2016,2,27), "Affitto", Decimal("-0.3")) )
        print("Esito inserimento tabella: ", cur.statusmessage )
# In questo punto il with precedente è chiuso: cursore chiuso e fatto un commit
with conn :
    with conn.cursor() as cur :
        cur.execute("SELECT id, data, voce, importo FROM Spese")
        print('=' * 55)
        print("| {:>2s} | {:10s } | {:<20} | {:>10s} |".format( \
            "N", "Data", "Voce", "Importo") )
        print('-' * 55)
        tot = Decimal(0)
        for riga in cur :
            print("| {:>2d} | {:10s} | {:<20} | {:>10.2f} |" \
                .format(riga[0], str(riga[1]), riga[2], riga[3]) )
            tot += riga[3]
        print('-' * 55)
# In questo punto cur è chiuso ed è stato fatto un commit !
# Si può ora chiudere la connessione e stampare il totale degli importi
conn.close()
print(" {:>40s}      {:10.2f}".format("Totale", tot) )
```

6 Flask

6.1 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.

6.2 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 ...
```

6.3 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'])
```

6.4 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('nessunPrestito0Errore.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('nessunPrestito0Errore.html')

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

6.5 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>
```

7 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.

7.1 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");
```

7.2 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());
}
```

Esempio di url:

```
"jdbc:postgresql://dbserver.scienze.univr.it/db0"
```

La classse `Connection` contiene i seguenti metodi fondamentali:

- `createStatement()`: ritorna un oggetto `Statement`, che permette di invare query statiche.
- `prepareStatement(query)`: ritorna un oggetto `PreparedStatement`, che rappresenta la query ma che permette di reinviare la stessa più volte ma con parametri diversi.
- `commit()`: registra la transazione corrente (normalmente le connessioni in JDBC sono in auto-commit).
- `rollback()`: abortisce la transazione corrente.
- `close`: chiude la connessione corrente

7.2.1 Differenza tra Statement e PreparedStatement

- un oggetto di tipo **Statement** è sufficiente per inviare query semplici senza parametri.
- un oggetto di tipo **PreparedStatement** è da preferire quando una stessa query deve essere riusata più volte con parametri diversi o anche non si vuole fare la conversione esplicita dei valori dei parametri da Java nei corrispondenti SQL.

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

dove `connection` è un'istanza della classe `Connection`.
Se lo statement è un comando di `select` (come in questo caso) si usa il metodo `executeQuery`, altrimenti nel caso di comandi 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 i metodi `setDate()`, `setString()`, ecc. I metodi set funzionano passando come parametro un **indice** (partendo 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.

7.4 Accesso ai campi

La query eseguita con il metodo `executeQuery()` ritorna un oggetto di tipo `ResultSet`, il quale contiene lo stato e l'eventuale cursore sulla tabella risultato.

- Il metodo `next()` di ResultSet posiziona il cursore alla prossima riga non letta della tabella e restituisce vero se esiste, falso altrimenti.
- I metodi `get<tipo>(<index>)` e `get<tipo>(<nome>)` di ResultSet permettono di recuperare il valore della colonna `<index>`/`<nome>` della riga corrente.

Esempio:

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

while(rs.next()) {
    System.out.print("Data : " + rs.getData(1) );
    System.out.print("Voce : " + rs.getString("descrizione") );
    System.out.print("Importo : " + rs.getBigDecimal(3) );
}
```

7.5 Chiusura risorse

Quando gli statement da eseguire sono terminati, va chiusa la connessione utilizzata con:

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

7.6 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");
    }
  }
}
```

8 Appello del 19/09/2018

Si consideri il seguente schema relazionale (chiavi primarie sottolineate) contenente le informazioni relative ai voli:

VOLO(iataCompagnia, numero, orarioPartenza, aeropPartenza, aeropArrivo, durata)
AEROPORTO(iata, città)
COMPAGNIA(iata, nome)
BIGLIETTO(numero, dataAcquisto, prezzo)
VOLIBIGLIETTO(numeroBiglietto, iataCompagnia, numero, orarioPartenza, ordinale)

dove **iataCompagnia** è il codice di 2 lettere della compagnia aerea, **VOLO.numero** è un intero positivo, **VOLO.orarioPartenza** è la data e l'orario con fuso orario della partenza, **VOLO.aeropPartenza/aeropArrivo** sono i codici IATA degli aeroporti (4 lettere). Per questioni di semplicità si assume che i codici IATA delle compagnie e degli aeroporti siano già definiti come domini di nome, rispettivamente, **IATACompagnia** e **IATAAeroporto**. Un biglietto comprende uno o più voli. **BIGLIETTO.numero** è una stringa di 13 cifre. **VOLIBIGLIETTO.ordinale** indica l'ordine del volo ed è = 0 se il biglietto è composto da un solo volo.

Domanda 1

Determinare i vincoli di integrità che si possono desumere usando la notazione '→'.
Scrivere il codice PostgreSQL che generi TUTTE le tabelle per rappresentare lo schema relazionale specificando tutti i possibili controlli di integrità referenziale e almeno 3 controlli di correttezza dei dati.

Soluzione

VOLO.iataCompagnia→COMPAGNIA.iata
VOLO.aeropPartenza→AEROPORTO.iata
VOLO.aeropArrivo→AEROPORTO.iata
VOLIBIGLIETTO.numeroBiglietto→BIGLIETTO.numero
VB.iataCompagnia,VB.numero, VB.orarioPartenza→V.iataCompagnia,V.numero, V.orarioPartenza

Codice SQL:

```
CREATE DOMAIN numBiglietto AS CHAR(13)
    CHECK (VALUE SIMILAR TO '[0-9]{13}' );

CREATE TABLE AEROPORTO(
    iata IATAAeroporto PRIMARY KEY,
    città TEXT NOT NULL
);

CREATE TABLE COMPAGNIA(
    iata IATACompagnia PRIMARY KEY,
    nome TEXT NOT NULL
);

CREATE TABLE BIGLIETTO(
    numero numBiglietto PRIMARY KEY,
    dataAcquisto DATE NOT NULL,
    prezzo DECIMAL(6,2) NOT NULL
);

CREATE TABLE VOLO(
    iataCompagnia IATACompagnia
        REFERENCES COMPAGNIA(iata),
    numero INTEGER CHECK (numero > 0),
    orarioPartenza TIMESTAMP WITH TIME ZONE,
```

```
    aeropPartenza IATAAeroporto
        REFERENCES AEROPORTO(iata),
    aeropArrivo IATAAeroporto
        REFERENCES AEROPORTO(iata),
    durata INTERVAL,
    PRIMARY KEY (iataCompagnia, numero, orarioPartenza)
);

CREATE TABLE VOLIBIGLIETTO(
    numeroBiglietto numBiglietto,
    iataCompagnia IATACompagnia,
    numero INTEGER,
    orarioPartenza TIMESTAMP WITH TIME ZONE,
    ordinale INTEGER NOT NULL CHECK (ordinale >= 0),
    PRIMARY KEY (numeroBiglietto, iataCompagnia, numero,
        orarioPartenza),
    FOREIGN KEY (iataCompagnia, numero, orarioPartenza)
        REFERENCES VOLO(iataCompagnia, numero, orarioPartenza)
);
```

Domanda 2

Scrivere il codice PostgreSQL che determina le soluzioni alle seguenti due interrogazioni usando il minor numero di JOIN:

(a) Per ogni biglietto che ha più di un volo, visualizzare il numero del biglietto e tutti i suoi voli in ordine temporale. Ciascun volo deve essere completato con aeroporto di partenza, di arrivo e orarioPartenza.

(b) Dato l'aeroporto di partenza 'X' e quello di arrivo 'Y', visualizzare i biglietti che hanno almeno un volo fra i due aeroporti riportando, per ciascun biglietto, il numero del biglietto, il numero e la durata complessiva dei voli del biglietto.

Soluzione

(a)

```
SELECT VB.numeroBiglietto, V.numero, V.aeropPartenza,
    V.aeropArrivo, V.orarioPartenza
FROM VOLIBIGLIETTO VB JOIN VOLO V ON
    VB.iataCompagnia = V.iataCompagnia AND
    VB.numero = V.numero AND
    VB.orarioPartenza = V.orarioPartenza
WHERE VB.ordinale >= 1
ORDER BY VB.numeroBiglietto, V.orarioPartenza;
```

(b) (forse troppi join usati)

```
SELECT VB.numeroBiglietto, COUNT(VB.ordinale), SUM(V.durata)
FROM VOLIBIGLIETTO VB JOIN VOLO V on
    VB.iataCompagnia = V.iataCompagnia and
    VB.numero = V.numero and
    VB.orarioPartenza = V.orarioPartenza
WHERE VB.numeroBiglietto = ANY(
    SELECT VB2.numeroBiglietto
    FROM VOLIBIGLIETTO VB2 JOIN VOLO V2 on
        VB2.iataCompagnia = V2.iataCompagnia and
        VB2.numero = V2.numero and
        VB2.orarioPartenza = V2.orarioPartenza
    WHERE VB.numeroBiglietto = VB2.numeroBiglietto AND
```

```
            V2.aeropPartenza = 'X' AND V2.aeropArrivo = 'Y'
    )
    GROUP BY VB.numeroBiglietto      --necessario per count/sum
    ORDER BY VB.numeroBiglietto;
```

Domanda 3

Data una base di dati PostgreSQL che contenga le tabelle della Domanda 1, scrivere un programma Python che, leggendo il numero di biglietto da console, visualizzi tutti i voli del biglietto, in ordine, con la sosta tra un volo e l'altro (se il biglietto ha più voli, altrimenti la sosta ha valore 0). Se il biglietto non esiste, il programma deve richiedere il numero del biglietto.
Suggerimento: ***datetime.timedelta*** *rappresenta un interval.* ***durataSosta = timedelta(seconds=0)****...*

Soluzione

```
from datetime import date, timedelta
from decimal import Decimal
from getpass import getpass
import psycopg2

bigliettoEsiste = 0
psw = getpass("Inserire la password: ")

conn = psycopg2.connect(host="dbserver.scienze.univr.it", \
    database="id468wfl", user="id468wfl", password=psw)

with conn:
    with conn.cursor() as cur:
        while(not bigliettoEsiste):

            codice = input("Inserire il codice del biglietto: ")
            cur.execute("SELECT numero, orarioPartenza, ordinale
                FROM VOLIBIGLIETTO WHERE numeroBiglietto =  %s
                ORDER BY orarioPartenza", (codice,) )

            voli = cur.fetchall()

            if not voli:    #oppure usare cur.rowcount ?
                print("Il biglietto non esiste!")
            else:
                bigliettoEsiste = 1

oraPrecedente = timedelta(seconds = 0)  #non so come usarlo
for riga in voli:

    #dopo il volo con ordinale 1 stampa la sosta
    if riga[2] > 1:
        #timestamp - timestamp = interval
        durataSosta = riga[1] - oraPrecedente
        print("--sosta: " + str(durataSosta))

    print("numero: {:<6s} | orarioPartenza: {:22s}" \
        .format(str(riga[0]) , str(riga[1])) )

    oraPrecedente =  riga[1]
conn.close()
```

Domanda 4

Si consideri la tabella `BIGLIETTO` della Domanda 1. Si scrivano due transazioni in cui: 1) nella prima transazione si aumenta del 5% il prezzo a tutti i biglietti che partono dopo il '2018-11-01 00:00:00 CET', mentre 2) nell'altra si abbassa il prezzo del 10% a tutti i biglietti che hanno prezzo pari a 1050 euro. Che tipo di errore si può verificare se le due transazioni sono concorrenti? Che tipo di isolamento si deve dichiarare per eliminare l'errore? In quale transazione?

Soluzione

Transazione 1:

```
BEGIN;

UPDATE BIGLIETTO
SET prezzo = prezzo * 1.05
WHERE dataAcquisto >= '2018-11-01';

COMMIT;
```

Transazione 2:

```
BEGIN TRANSACTION ISOLATION LEVEL
  REPEATABLE READ;

UPDATE BIGLIETTO
SET prezzo = prezzo * 0.9
WHERE prezzo = 1050;
...
```

La seconda transazione (T2) è stata scritta con livello di isolamento REPEATABLE READ per evitare anomalie Nonrepeatable reads; se infatti le due transazioni sono concorrenti, è possibile che il controllo della seconda transazione (T2) sul valore di 'prezzo' (se è = 1050) non legga determinati valori corretti (= T2 si ferma quando legge l'UPDATE concorrente di T1).

Esempio con READ COMMITTED: ci sono 2 biglietti, il primo con prezzo 1050, l'altro 1000. T1 e T2 iniziano contemporaneamente. T1 aumenta il prezzo dei due biglietti (il secondo diventa 1050), T2 controlla i biglietti da modificare; T2 vede che ci sono state modifiche in T1 e aspetta il commit. Nel momento del commit di T1, T2 ricontrolla SOLO il biglietto che aveva prezzo 1050, valore ora diverso, e l'UPDATE non fa modifiche, senza vedere che il biglietto che aveva prezzo 1000 ora lo ha 1050, e sarebbe quindi da aggiornare.

Con isolamento REPEATABLE READ invece, T2 fallisce quando legge le modifiche di T1.

Domanda 5

Si consideri il seguente risultato del comando `ANALYZE` su una query inerenti a 2 tabelle:

```
Hash JOIN (cost=17.68..36.19 ROWS=3 width=84
    Hash Cond: (((v.iatacompagnia)::TEXT = (b.iatacompagnia)::TEXT) AND
        (v.numero = b.numerovolo) AND
        (v.orarioPartenza=b.orarioPartenza))
    -> Seq Scan ON volo v (cost=0.00..14.00 ROWS=400 width=60)
    -> Hash (cost=17.62..17.62 ROWS=3 width=104)
        -> Seq scan ON volibiglietto b (cost=0.00..17.62 ROWS=3 width=104)
            Filter: (numerobiglietto = '1234567890123'::bpchar)
```

Desumere il testo della query e indicare un indice che potrebbe migliorare l'esecuzione della query. *Suggerimento: la query inizia con* ***SELECT*** *b.numeroBiglietto, b.ordinale, b.orarioPartenza, v.durata* ***FROM****...*

Soluzione

Query:

```
SELECT b.numeroBiglietto,b.ordinale,b.orarioPartenza,v.durata
FROM VOLIBIGLIETTO b JOIN VOLO v ON
    b.iataCompagnia = v.iataCompagnia AND
    b.numero = v.numero AND
    b.orarioPartenza = v.orarioPartenza
WHERE b.numeroBiglietto = '1234567890123';
```

La query filtra su chiavi primarie quindi non dovrebbero essere necessari nuovi indici.

9 Appello del 16/02/2018 (Da sistemare)

Si consideri il seguente schema relazionale parziale e semplificato (chiavi primarie sottolineate) contenente le informazioni relative alla programmazione di convegni:

CONVEGNO(nome, dataInizio, dataFine, numeroSessioni, tipo, luogo)
INTERVENTO_IN_CONVEGNO(nomeConvegno, idIntervento, nomeSessione, orarioInizio)
SESSIONE(nome, nomeConvegno, data, orarioInizio, orarioFine)
INTERVENTO(id, titolo, relatore, durata)

dove **numeroSessioni** è un intero, **tipo** può assumere valori (seminario, simposio, conferenza), **luogo** è generico (VARCHAR è sufficiente), INTERVENTO_IN_CONVEGNO.**orarioInizio** è l'orario di inizio dell'intervento all'interno della sessione. Gli attributi inerenti al tempo devono avere il fuso orario.

Domanda 1
Scrivere il codice PostgreSQL che generi i domini e le tabelle per rappresentare lo schema relazionale specificando tutti i possibili controlli di integrità e di correttezza dei dati opportuni.

Soluzione

```
CREATE DOMAIN tipo_domain AS TEXT
    CHECK (VALUE IN ('seminario', 'simposio', 'conferenza') );

CREATE TABLE CONVEGNO(
    nome VARCHAR PRIMARY KEY,
    dataInizio DATE NOT NULL, -- o timestamp with timezone ??
    dataFine DATE NOT NULL,
    numeroSessioni INTEGER, -- check > 0 ??
    tipo tipo_domain,
    luogo VARCHAR
);

CREATE TABLE INTERVENTO(
    id VARCHAR PRIMARY KEY,
    titolo VARCHAR NOT NULL,
    relatore VARCHAR NOT NULL,
    durata INTERVAL
);

CREATE TABLE SESSIONE(
    nome VARCHAR,
    nomeConvegno VARCHAR REFERENCES CONVEGNO(nome),
    data DATE,
    orarioInizio TIME WITH TIME ZONE,
    orarioFine TIME WITH TIME ZONE,
    PRIMARY KEY(nome, nomeConvegno)
);

CREATE TABLE INTERVENTO_IN_CONVEGNO(
    nomeConvegno VARCHAR REFERENCES CONVEGNO (nome),
    idIntervento VARCHAR REFERENCES INTERVENTO (id),
    nomeSessione VARCHAR,
    orarioInizio TIME WITH TIME ZONE,
    PRIMARY KEY(nomeConvegno, idIntervento, nomeSessione),
    FOREIGN KEY(nomeSessione, nomeConvegno)
        REFERENCES SESSIONE(nome, nomeConvegno)
);
```

Domanda 2

Dato il nome 'X' di un convegno, scrivere una query che visualizzi il programma del convegno. Il programma è costituito da un elenco ordinato per giorno e ora degli interventi: sessione, titoloIntervento, relatore e orario inizio intervento.

Soluzione

```
SELECT S.nome, I.titolo, I.relatore, S.data, IiC.orarioInizio
FROM INTERVENTO_IN_CONVEGNO IiC JOIN INTERVENTO I ON
    IiC.idIntervento = I.id JOIN SESSIONE S ON
    IiC.nomeSessione = S.nome
    -- AND IiC.nomeConvegno = S.nomeConvegno ?? serve??
WHERE IiC.nomeConvegno = 'X'
ORDER BY S.data, IiC.orarioInizio;
```

Domanda 3

Assumendo di avere una base di dati PostgreSQL che contenga le tabelle di questo tema d'esame, scrivere un programma Python che, leggendo una o più tuple del tipo 'nomeConvegno', 'nomeSessione', 'idIntervento', 'orarioInizio' da console, inserisca una o più tuple nella tabella INTERVENTO_IN_CONVEGNO. L'inserimento deve garantire la correttezza della base di dati: controllo preventivo che le eventuali dipendenze siano rispettate e che l'orario dell'intervento sia compatibile con la sua durata e la presenza di altri interventi. Se una dipendenza non è rispettata, il programma deve chiedere di reinserire il dato associato alla dipendenza prima di procedere ad inserire la tupla. Il programma deve visualizzare l'esito di ogni singolo inserimento. È richiesto che il programma suggerisca il tipo di dati da inserire, che non ammetta possibilità di SQL Injection e che sia eseguibile in concorrenza con altre istanze del programma stesso.

Soluzione (incompleta)

```
from datetime import date, timedelta
from decimal import Decimal
from getpass import getpass
import psycopg2

fine = 0
psw = getpass("Inserire la password: ")

conn = psycopg2.connect(host="dbserver.scienze.univr.it", database="id468wfl",
    user="id468wfl", password=psw)

with conn:
  with conn.cursor() as cur:
    while(not fine):

      check = 0
      nomeConvegno = input("Inserire il nome del convegno: ")
      #check esistenza convegno
      while not check:
        cur.execute("SELECT * FROM Convegno WHERE nome = %s", (nomeConvegno,))
        if not cur.fetchall():
          nomeConvegno = input("Nome di convegno inesistente. Reinserire: ")
        else:
          check = 1

      check = 0
      nomeSessione = input("Inserire il nome della sessione: ")
      #check esistenza sessione
      while not check:
        cur.execute("SELECT * FROM Sessione WHERE nome = %s AND nomeConvegno = %s",
          (nomeSessione, nomeConvegno))
        if not cur.fetchall():
          nomeSessione = input("Nome di sessione inesistente. Reinserire: ")
        else:
          check = 1
```

```
        check = 0
        idIntervento = input("Inserire l'id dell'intervento: ")
        #check esistenza intervento
        while not check:
          cur.execute("SELECT * FROM Intervento WHERE id = %s", (idIntervento,))
          if not cur.fetchall():
            idIntervento = input("ID di intervento inesistente. Reinserire: ")
          else:
            check = 1

        orarioInizio = input("Inserire l'orario di inizio: ")
        #TODO: check orarioInizio Intervento > orarioInizio e < orarioFine sessione
        #TODO: check orarioInizio Intervento + durata <= orarioFine sessione

        #inserimento nuova tupla
        cur.execute("INSERT INTO INTERVENTO_IN_CONVEGNO ( nomeConvegno, \
            idIntervento, nomeSessione, orarioInizio) values (%s,%s,%s,%s)",
            (nomeConvegno, idIntervento, nomeSessione, orarioInizio) )

        print("Esito: " + cur.statusmessage)

        scelta = ""
        while(scelta != "S" and scelta != "n"):
          scelta = input("(S/n) Continuare inserimento? ")
        if(scelta == "n"):
          fine = 1

conn.close()
```

Domanda 4 .....

Scrivere il codice PostgreSQL, definendo anche eventuali viste, per rispondere alle seguenti due interrogazioni nel modo più efficace:

(a) Trovare per ogni convegno il numero totale e la durata totale degli interventi per ciascuna sessione di convegno. Il risultato deve visualizzare nome convegno, giorno e i conteggi richiesti.

(b) Trovare per ogni convegno distribuito su almeno 3 giorni il numero totale di interventi e la durata totale degli interventi per ciascun giorno di convegno. Il risultato deve visualizzare nome convegno, giorno e i conteggi richiesti.

Soluzione .....

(a)

```
SELECT S.nomeConvegno, S.nome, S.data, COUNT(I.id) as
    "n_interventi", SUM(I.durata) as "durata_totale"
FROM INTERVENTO_IN_CONVEGNO IiC JOIN SESSIONE S ON
    S.nome = IiC.nomeSessione JOIN INTERVENTO I ON
    I.id = IiC.idIntervento
GROUP BY S.nomeConvegno, S.nome
ORDER BY S.nomeConvegno, S.data;
```

(b)

```
SELECT C.nome, S.data, COUNT(I.id) as
    "n_interventi", SUM(I.durata) as "durata_totale"
FROM INTERVENTO_IN_CONVEGNO IiC JOIN Convegno C on
    IiC.nomeConvegno = C.nome JOIN SESSIONE S on
    IiC.nomeSessione = S.nome JOIN INTERVENTO I on
    IiC.idIntervento = I.id
WHERE C.dataFine::date - C.dataInizio::date >= 2
    -- date-date=integer (es: '2018-01-03'-'2018-01-01' = 2)
GROUP BY C.nome, S.data;
```

Domanda 5

Si consideri il seguente piano di esecuzione:

```
HashAggregate (cost=1859.67..1866.24 ROWS=657 width=60
  GROUP KEY: i.nomeins, d.descrizione
  ->  Hash JOIN (cost=319.54..1856.39 ROWS=657 width=60
      Hash Cond: (ie.id_insegn = i.id)
      ->  Hash JOIN (cost=31.74..1559.55 ROWS=657 width=25)
          Hash Cond: (ie.id_discriminante = d.id)
          ->  Bitmap Heap Scan ON inserogato ie (cost=26.68..1543.97 ROWS=1053 width=8)
              Recheck Cond: (((annoaccademico)::TEXT='2009/2010'::TEXT) AND (crediti=ANY
  ('{3,5,12}'::NUMERIC[])) AND (modulo = '0'::NUMERIC))
                ->  Bitmap Index Scan ON ins_aa (cost=0.00..26.42 ROWS=1053 width=0)
                    INDEX Cond: (((annoaccademico)::TEXT='2009/2010'::TEXT) AND (crediti=ANY
  ('{3,5,12}'::NUMERIC[])) AND (modulo = '0'::NUMERIC))
          -> Hash (cost=3.36..3.36 ROWS=136 width=25)
            ->  Seq Scan ON discriminante d (cost=0.0..3.36 ROWS=136 width=25)

      -> Hash (cost=185.69..185.69 ROWS=8169 width=43)
        ->  Seq Scan ON insegn i (cost=0.00..185.69 ROWS=8169 width=43)
```

(a) Indicare quanti e quali indici sono stati usati per risolvere la query. Per indicare quali sono gli indici, scrivere il codice SQL per crearli.

(b) Supponendo che i dati non varino nel tempo e che le chiavi primarie hanno indice, in base al piano di esecuzione conviene creare degli altri indici? Se sì, quali?

Soluzione

(a) Un solo indice utilizzato, `ins_aa`:

```
CREATE INDEX ins_aa ON InsErogato (annoaccademico,
    crediti, modulo)    --???
```

(b)

```
HashAggregate
└── Hash Join
    ├── Hash Join
    │   ├── Bitmap Heap Scan
    │   │   └── Bitmap Index Scan
    │   └── Hash
    │       └── Seq Scan
    └── Hash
        └── Seq Scan
```

Secondo il piano di esecuzione la query fa un `Hash JOIN` tra un ulteriore `JOIN` e un `Seq Scan` di un'intera tabella (`insegn`); la condizione di uguaglianza è su due chiavi (`ie.id_insegn` e `i.id`), che hanno già di default un indice, perciò non è necessaria qui la creazione di nuovi indici.
Il `JOIN` interno è tra un `Bitmap Scan` che utilizza già un indice e un `Seq Scan` di un'intera tabella (`discriminante`); la condizione di uguaglianza è sulla chiave di `discriminante` (`d.id`) e su `ie.id_discriminante`, che non è una chiave (nella tabella InsErogato). In questo caso un indice su quest'ultimo potrebbe essere utile a ridurre il costo della query, che è abbastanza significativo.

```
CREATE INDEX ie_iddiscr ON Inserogato (id_discriminante);
```

N.B. `ie.id_discriminante` non è chiave di InsErogato, ma il database did2014 contiene già un indice "`occorrenzains_discriminante_index`". Questa cosa non è possibile saperla durante la prova perciò, dato che le *reference keys* non creano indici, credo sia comunque corretta questa soluzione.

10 Credits

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

Documento modificato da Edoardo Righi nell'Anno Accademico 2017/2018. Aggiunte due prove di esame da me risolte (mi auguro correttamente) e modificate alcune sezioni. Lascio un collegamento qui sotto al già ottimo documento originale.

`https://github.com/davbianchi/dispense-info-univr/tree/master/triennale/basi-di-dati-lab`