A.A. 2020/21

# BASI DI DATI - SQL

SARA MIGLIORINI

FABS :)

## NOTA

Questi appunti/sbobinatura/versione “discorsiva” delle slides sono per mia utilità personale, quindi pur avendole revisionate potrebbero essere ancora presenti typos, commenti/aggiunte personali (che anzi, lascio di proposito) e nel caso peggiore qualche inesattezza!

Comunque spero siano utili! 

**Questa sbobina fa parte della mia collezione di sbobinature,**
**che è disponibile (e modificabile!) insieme ad altre in questa repo:**
https://github.com/fabfabretti/sbobinamento-seriale-uniVR

# Yet Another `psql` Cheatsheet

## Comandi PSQL

Connessione al server: `psql -h dbserver.univr.ir -U <Username GIA> <Username GIA>`

| Guida SQL | Guida psql | Elenco databases | Input file | Uscita |
|---|---|---|---|---|
| `\h` | `\?` | `\l` | `\i filename` | `\q` |

## Identificatori

Gli identificatory SQL sono stringhe che iniziano con una lettera UTF-8 o un underscore, e possono contenere cifre.

- Non sono sensibili a minuscolo/maiuscolo
- Tutti I nomi di tabella, attributo e così via sono identificatory SQL.

## Operatori

*Logica*

| AND | OR | NOT | MINORE | MAGGIORE | DIVERSO |
|---|---|---|---|---|---|
| `a AND b` | `a OR b` | `NOT a` | `a <= b` | `a >= b` | `a <> b` |

| COMPRESO TRA | CONTROLLO IDENTITA **`IS NULL`** | **`IN`** |
|---|---|---|
| `a [NOT] BETWEEN b` | `IS [NOT] {NULL \| TRUE \| FALSE}` L'uguaglianza a NULL va testata con questo operatore apposito. | Elenco di tutti i valori ammissibili; equivale a delle disuguaglianze messe in OR. |

- **Non possiamo fare controlli su `NULL`: bisogna usare `IS NULL`.**

*Stringhe*

Le stringhe si indicano con apici singoli.

| Concatenazione stringhe | Conversione in UPPERCASE | Conversione in lowercase |
|---|---|---|
| `stringa \|\| stringa` | `upper(a)` | `lower(a)` |

| | |
|---|---|
| **`LIKE`** | Confronto fra stringhe. Il LIKE fa pattern matching con caratteri speciali, dove<br>• `_` = un carattere qualsiasi<br>• `%` = 0 o più caratteri qualsiasi |
| **`SIMILAR TO`** | Come il LIKE, ma permette di usare le espressioni regolari<br>• `*` = ripetizione del match precedente da 0 a N volte<br>• `_` , `%`<br>• `+` = ripetizione del match precedente una o più volte<br>• `{n,m}` = ripetizione del precedente match almeno n e non più di m volte<br>• `[char,char..]` = elenco dei caratteri ammissibili<br>• `?` = potrebbe esserci o non esserci<br>Esempio: `'\+[0-9]+'` |

## Comandi DDL

*CREATE TABLE*

Definisce lo schema di una relazione e ne crea un'istanza vuota.

```
CREATE [TEMP] TABLE tabella (
      nomeAttributo dominio vincolo,
    nameId INTEGER NOT NULL,
    price NUMERIC DEFAULT 9.99
      CONSTRAINT pPrice NOT NULL CHECK (price > 0)

    FOREIGN KEY(nameId) REFERENCES productNames(id)
);
```

## Domini

*Caratteri*

| | |
|---|---|
| **CHARACTER** | Carattere singolo |
| **CHARACTER(20)** | Stringa di lunghezza fissa. Se si assegna una stringa di lunghezza inferiore, viene riempita di spazi. |
| **CHARACTER VARYING(20)** | Stringa di lunghezza variabile, con cap superiore |
| **TEXT** | lunghezza variabile senza limite prefissato. |

*Booleano*

| | |
|---|---|
| **BOOLEAN /BOOL** | Valore booleano. Rappresenta I valori booleani TRUE / FALSE, più lo stato *unknown* NULL. |

*Tipi numerici*

| | | |
|---|---|---|
| **INTEGER** | **SMALLINT** | **NUMERIC(precisione,scala)**<br>**DECIMAL(precisione,scala)** |
| Numeri a 4 bytes. | Numeri a 2 bytes. | Precisione = numero totale di cifre significative<br />Scala = numero di cifre dopo la virgola. |

| | |
|---|---|
| **REAL** | **DOUBLE PRECISION** |
| Precisione di 6 cifre decimali | Precisione fino a 15 cifre decimali. |

> Si tratta comunque di valori approssimati, implementati come numeri avirgola mobile. Se si devono usare importi di denaro che contengono anche decimali, SEMPRE USARE NUMERIC!

*Tempo*

| DATE | TIME [(precisione)] [WITH TIME ZONE] | TIMESTAMP [(precisione)] [WITH TIME ZONE] | INTERVAL [fields] [(p)] |
|---|---|---|---|
| Rappresenta la data come 'YYYY-MM-DD' | **precisione**: numero di cifre decimali per rappresentare le frazioni del secondo<br>**WITH TIME ZONE**: specifichiamo la differenza con l'ora di Greenwich | Salva data e tempo. | **fields** rappresenta l'estensione: year, month, year to month…<br>**p** rappresenta la precisione delle frazioni di secondo, se in fields ci sono i secondi. |
| `'2016-01-15'` | `04:05:06.789 / 04:05:06-08:00' / '12:01:01 CET'` | `'2016-01-24 00:00:00+01'` | `'3 4:05:06'`: formato SQL std per 3 giorni, 4 ore, 5 minuti e 6 secondi. `'P1Y2M3DT4H5M6S'`: 1 anno, 2 mesi, 3 giorni, 4 ore, 5minuti e 6 secondi. |

*Definizione dominio utente*

```
CREATE DOMAIN nome AS tipoBase [vincolo di attributo]
--esempio:usiamo CHECK
CREATE DOMAIN giornisettimana AS CHAR(3)
      CHECK (
        VALUE IN   ('LUN','MAR','MER','GIO','VEN','SAB','DOM')
    )
```

## Vincoli

*Vincoli di attributo / intrarelazionali*

Specificano proprietà che devono essere soddisfatte da ogni tuple di una singola relazione.

| | |
|---|---|
| **NOT NULL** | Il valore non può essere nullo. Spesso si usa insieme a DEFAULT. |
| **CHECK(exp)** | Verifica che un'espressione booleana sia soddisfatta; è soddisfatta **sia se la condizione è vera che se è nulla!!**<br>**Conviene mettere sempre NOT NULL insieme al vincolo CHECK()!** |
| **DEFAULT** | Se non viene specificato un valore, assume quello indicato nel vincolo. |
| **UNIQUE** | Non possono esistere altre tuple con lo stesso valore in questo attributo (o su un insieme di attributi). Solitamente si usa perché si vuole specificare una lista di attributi. |
| **PRIMARY KEY** | Indica che gli attributi sono chiave primaria. Impone implicitamente UNIQUE e NOT NULL. |
| **REFERENCES** | Sarebbero le chiavi esportate; è un vincolo extrarelazionale |
| **ON DELETE** / **ON UPDATE** | Posso scrivere modi in cui un attributo reagisce a certe azioni. |

*Vincoli di integrità referenziale*

Un vincolo di integrità referenziale crea un legame tra i valori di unattributo (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 diA, se diverso dal valore nullo, **sia presente tra i valori** di B nella tabella esterna.

Vanno definiti nella tabella slave.

Non è detto che sia una chiave primaria della tabella master, ma deve essere quantomeno identificante

**REFERENCES**: uso questo se il numero di attributi che fanno parte della chiave è 1.

```
CREATE TABLE tab_interna
      ...
      attributo VARCHAR(10)
            REFERENCES tab_esterna(chiave)
      ...
```

**FOREIGN KEY**: uso questo se il numero di attributi che fanno parte della chiave è > 1.

```
      FOREIGN KEY (int_column,...) REFERENCES tab_esterna (ref_column)
```

## Modifica della struttura di una tabella: ALTER TABLE

*Aggiungere colonna*

```
ALTER TABLE tabella ADD COLUMN attributo tipo;
```

*Eliminare colonna*

```
ALTER TABLE tabella DROP COLUMN attributo;
```

*Modificare valore di default*

```
ALTER TABLE tabella ALTER COLUMN attributo SET DEFAULT valore;
-- or
ALTER TABLE tabella ALTER COLUMN attributo DROP DEFAULT;
```

*Aggiunta e rimozione vincoli di integrità referenziale*

```
ALTER TABLE nome_tabella ADD CONSTRAINT
ALTER TABLE nome_tabella DROP CONSTRAINT nome_vincolo7
-- Se non si assegna un nome al vincolo, lo decide il DBMS.
```

Per conoscere il nome dei vincoli posiamo usre `\d nometabella`.

*Elimina struttura e dati tabella*

```
DROP TABLE [IF EXISTS] tabella -- una fantasia quasi senza pari
```

## Gestione dati

*Inserimento dati*

```
INSERT INTO tabella (elenco, attributi)
      VALUES('valore', 'valore'),
                  ('valore', 'valore');
-- or, sconsigliato
INSERT INTO tabella VALUES ('valore','valore');
```

*Modifica dati*

```
UPDATE tabella
      SET attributo = espressione
      [WHERE condizione];
-- esempio
UPDATE impiegato
      SET stipendio = stipendio * 100
      WHERE nome = Fabiola
```

*Elimina dati*

```
DELETE FROM tabella
      WHERE condizione;
--esempio
DELETE FROM impiegato WHERE matricola = 'A001';
```

## Query SQL base

### SELECT

Segue l'elenco degli attributi che saranno presenti nel risultato.

| | |
|---|---|
| **AS** | rinomina le colonne risultato |
| **DISTINCT** | elimina le tuple duplicate |

### FROM

Segue l'elenco delle tabelle da cui viene estratto il risultato.

- Posso anche metterci un'altra select
- Quello che fa è calcolare il prodotto cartesiano di tutte e applicare SELECT/WHERE. Posso specificare direttamente delle JOIN per ridurre la cardinalità da calcolare.

| | |
|---|---|
| **AS** | Nel caso del FROM serve a disambiguare; per esempio, facendo un prodotto cartesiano di un element con se stesso, ho bisogno dell'alias per sapere a quale delle due istanze mi sto riferendo. |

### WHERE

Seleziono e restringo le fonti specificando delle condizioni. È una condizione booleana, composta come

```
C = expr [θ {expr | const}]
```

Dove

- expr := espressione che contiene riferimenti agli attributi delle tabelle che compaiono nella clausola FROM
- **θ** appartiene a {=,<>,<=,>=}
- const := valore dei domini di base.

Ci sono diversi altri operatori comodi:

### ORDER BY

Definisce un ordinamento sul risultato rispetto ad un attributo. Di default è ascendente ma posso decidere anche di dare DESC.

### COUNT, MAX, MIN, AVG, SUM

> Dopo la SELECT con **operatori di aggregazione** non si possono costruire espressioni che usano i valori presenti nelle singole tuple, poiché il risultato è sempre una sola tupla.

#### *COUNT*

Restituisce il numero di tuple significative nel risultato dell'interrogazione.

| | |
|---|---|
| **COUNT(*)** | Ritorna il numero di tuple |
| **COUNT(expr)** | Ritorna il numere di tuple in cui expr non è nullo |
| **COUNT(DISTINCT expr)** | Ritorna il numero di tuple in cui expr non è nullo, e conta solo i valori distinti. |

#### *MAX, MIN, AVG, SUM*

SUM e AVG danno un singolo valore numerico; MAX e MIN possono essere applicati anche a stringhe-

> expr::newType converte il valore di expr a un valore del dominio newType. In SQL standard, l'istruzione di casting è `CAST (expr AS newType)`, mentre `::` è un'abbreviazione di PostreSQL.

### GROUP BY

Un raggruppamento è un insieme di tuple con i medesimi valori su uno o più attributi. Group by permette di determinare tutti i raggruppamenti delle tuple della relazione risultato. I

> Il raggruppamento è **case sensitive**. Sarebbe una buona idea usare LOWER per evitare doppioni.
>
> Nella SELECT posso mettere solo un sottoinsieme degli attributi di GROUP BY e OPERATORI DI AGGREGAZIONE

Nel GROUP BY non si possono usareespressioni con operatori di aggregazione

### HAVING

È un'espressione booleana che può usare solo gli attributi di GROUP BY o operatori di aggregazione.

> Mettendo (per esempio) un AVG con type cast nella select, esso **agisce solamente nella visualizzazione e non sul raggruppamento**; eventualmente dovrò fare un type cast anche nel raggruppamento o rischio incongruenze.

```
SELECT LOWER(città) AS "Città",
     AVG(media)::DECIMAL(5,2) AS "mediaArr."
FROM Studente
GROUP BY LOWER(città)
HAVING AVG(media)>23.47;
```

| Città | mediaArr. |
|---|---|
| padova | 28.67 |
| va | 28.67 |
| verona | 23.47 |

| WHERE | HAVING |
|---|---|
| Seleziona le **righe** che fanno parte del risultato | Seleziona i **raggruppamenti** che fano parte del risultato. |

### JOIN

| *CROSS JOIN* | *[INNER] JOIN* | *LEFT/RIGHT/FULL OUTER JOIN* |
|---|---|---|
| Prodotto cartesiano classico. | Corrisponde alla theta join dell'algebra relazionale. Combina le due tabelle e restituisce le righe per cui la condizione è vera. | Nell'outer join si fa un'inner JOIN e poi si appendono tutte le righe della tabella di destra o di sinistra per cui non è soddisfatta la condizione mettendo a NULL tutti i campi relativi all'altra tabella. **Non è simmetrico!** |

Una FULL JOIN è diversa dal normale prodotto cartesiano:

| *FULL JOIN* | *CROSS JOIN* |
|---|---|
| per le corrispondenze mancanti, mi riempe a null il resto (=**max 1 per ogni corrispondenza**). Ho una condizione che mi limita il nuemero di righe. | fa tutte le accoppiate possibili: qualsiasi riga della tabella T1 con qualsiasi riga della tabella T2. |

## Query SQL nidificata

Un'interrogazione è nidificata quando è all'interno di un'altra interrogazione. SQL permette di fare un'interrogazione nidificata sia dentro il FROM che dentro il WHERE.

Se si utilizzano interrogazioni nidificate, il confronto è tra un valore di attributo e il risultato di un'interrogazione, ovvero un insieme di valori. Quindi è necessario utilizzare dei nuovi operatori che estendono i confronti.

### EXISTS

```
EXISTS (subquery)
```

Exists ritorna falso se la subquery non contiene righe; vero altrimenti. È significativo quando nella subquery si selezionano righe usando i valori della riga della SELECT principale → databinding

```
-- 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 as I2
               WHERE I.nome <> I2.nome AND CHAR_LENGTH(I.nome)=CHAR_LENGTH(I2.nome)
              );
```

```
-- Visualizzare il nome e il cognome dei docenti, escludendo i coordinatori,che hanno
-- tenuto almeno due insegnamenti (o moduli) con più di 24crediti nel 2010/2011

→ Persone per le quali esistono almeno 2 insegnamenti con le condizioni richieste
→ Raggruppo le docenze+inserogato accettate per persona e guardo se esistono gruppi con
più di due righe per l'insegnante
SELECT P.nome, P.cognome
FROM Persona P
WHERE EXISTS ( SELECT 1
               FROM Docenza as D
                    JOIN Inserogato IE ON D.id_inserogato = IE.id
               WHERE IE.crediti > 24 AND IE.annoaccademico = '2010/2011'
                     AND D.id_persona = P.id
                     AND D.coordinatore = '0'
               GROUP BY D.id_persona
               HAVING COUNT(*) >=2
              );
```

```
-- Visualizzare il nome dei corsi di studio che nel 2006/2007 non hannoerogato
insegnamenti il cui nome contiene la sottostringa 'Info'.

→ Prendo corsi di studio per cui NOT EXISTS (insegnamenti con sottostringa Info) che
sia parte del corso di studio
SELECT CS.nome
FROM CorsoStudi AS CS
WHERE NOT EXISTS ( SELECT 1
                   FROM Insegn AS I
                        JOIN Inserogato IE ON IE.id_insegn = I.id
                   WHERE  I.nome LIKE '%Info%' AND IE.annoaccademico = '2006/2007'
                   AND IE.id_corsostudi = CS.id
                 );
```

### IN

```
(expr) IN (subquery)        con (expr) = ROW(colonna,colonna…)
```

I valori dell'espressione vengono confrontati con i valori di ciascuna riga del risultato. Il confronto ritorna vero se i valori sono uguali ad almeno una delle righe delle subquery.

- La subquery deve tornare un numero di **colonne pari al numero di espressioni** date in pasto all'IN.

### ANY/SOME

```
(expr) operator ANY (subquery) || (expr) operator SOME (subquery)
```

Ritorna vero se expression è `operator` rispetto al valore di una **qualsiasi** riga del risultato di `subquery`. `operator` è un operatore di confronto, tipo '>'

- La subquery deve restituire **una sola colonna**

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

→ Faccio una colonna dove in ogni riga ho la media dei crediti di un anno accademico
→ Outer query avrà un < ANY
SELECT I.nomeins
FROM Insegn AS I
     JOIN InsErogato AS IE ON I.id = IE.id_inserogato
WHERE
   IE.crediti < ANY ( SELECT AVG(crediti)
                      FROM InsErogato
                      WHERE modulo = 0
                      GROUP BY annoaccademico
                    );
```

### ALL

```
(expr) operator ALL (subquery)
```

Ritorna vero solo se `expr` è `operator` rispetto al valore di **ciascuna** riga di `subquery`.

Expression deve coinvolgere attributi della query principale; operator è un operatore di confronto.

- La subquery deve restituire **una sola colonna**

```
-- Trovare il nome degli insegnamenti con almeno un docente e crediti maggiori rispetto
ai crediti di ciascun insegnamento del corso di laureacon id=6

→ Faccio una colonna con i crediti di ciascun insegnamento del corso di laurea con id =
6
→ Query esterna cerca che il numero di crediti sia > di ALL subquery
SELECT I.nomeins
FROM Insegn AS I
     JOIN Inserogato AS IE ON IE.id_insegn = I.id
     JOIN Docenza D ON IE.id = D.id_inserogato
WHERE
   IE.modulo = 0 AND crediti > ALL ( SELECT InsErogato.crediti
                                     FROM InsErogato
```

```
                                    WHERE modulo = 0 AND Inserogato.corsostudi = 6
                                );

-- Trovare il nome degli insegnamenti (o moduli) conalmeno due docenti e con crediti
maggioririspetto ai crediti di ciascun insegnamento del corso di laurea con id=6

→ Nella WHERE metto una query dove metto tutti gli insegnameti con crediti > di ciascun
insegnamento con corso di laurea id=6

SELECT I.nomeins
FROM ( -- query precedente
      SELECT I.nomeins, D.id_persona
      FROM Insegn AS I
           JOIN Inserogato AS IE ON IE.id_insegn = I.id
           JOIN Docenza D ON IE.id = D.id_inserogato
      WHERE
           IE.modulo = 0 AND crediti > ALL ( SELECT InsErogato.crediti
                                             FROM InsErogato
                                             WHERE modulo = 0
                                             AND Inserogato.corsostudi = 6 )
      )
GROUP BY nomeins -- gruppando per nomeins avrò tante righe quanti sono gli insegnanti!!
HAVING COUNT(*)>=2;
```

### Query SQL insiemistiche

Si possono utilizzare solo a livello esterno di una query, operando su due SELECT.

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

| | |
|---|---|
| **UNION** | Aggiunge il risultato di $query_1$ a $query_2$ |
| **INTERSECT** | Restituisce le righe che sono presenti sia in $query_1$ che in $query_2$ |
| **EXCEPT** | Restituisce le righe di $query_1$ che non sono presenti in $query_2$; sarebbe la differenza. |

- Di default elimina i duplicati, a meno di aggiungere ALL
- Si posono utilizzare solamente se i risultati hanno lo stesso numero di colonne e sono di tipo compatibile fra loro.

### Viste

Le viste sono tabelle virtuali, il cui contenuto dipende dal contenuto delle altre tabelle delle basi 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 la vista viene utilizzata, si riesegue la query che la deinisce.

Nell'interrogazione della vista si possono inserire anche altre viste, a patto che non ci siano:

- Dipendenze **immediate** (=definire una vista in termini di se stessa)
- Dipendenze **ricorsive** (=deinire un'interrogazione di base e un'interrogazione ricorsiva)
- Dipendenze **circolari**

```
CREATE [TEMP] VIEW nome [(nome_colonna[, …])] AS query
```

**TEMP**: la lista è temporanea: quando mi sconnetto essa viene distrutta. Nella nostra base di dati sono solo temporanee.

## Indici

Un indice è una **struttura dati ausiliaria** che fa riferimento a una tabella e permettere di **accedere in maniera più efficiente.** Sono costruiti rispetto a uno o più attributi.

Una volta creato l'indice, postgres deve applicare determinati algoritmi per mantenere aggiornati gli indici a fronte di eventuali aggiornamenti appllicati alle tabelle.

Il **costo di aggiornamento** degli indici può essere **significativo**, soprattutto se sono definiti molti indici nella tabella. Quindi mettere indici dappertutto non è una buona idea 🥲

Un indice, una volta creato, viene usato solamente quando l'ottimizzatore lo ritiene opportuno (in base alle analisi delle statistiche).

Gli indici possono velocizzare anche i comandi UPDATE/DELETE, se nella clausola WHERE ci sono attriburi indicizzati.

Se non ci sono gli indici, il DBMS deve fare una scansione sequenziale della tabella insegnamenti e riportare nel risultato le righe che soddisfano la condizione.

`\timing` → attiva/disattiva la visualizzazione del tempo di **pianificazione e calcolo**.
`\di` → visualizza l'elenco degli indici di un DB
`\d nomeTabella` → visualizza l'elenco degli indici di una tabella.

`CREATE`

```
CREATE INDEX [nome] ON nomeTabella [USING method]
({nomeAttributo | (expression)} [ASC|DESC] [,...])
```

- `method` è il tipo di indice:
  - `B-tree` (default) → è scelto in quanto efficiente anche sui range
  - `Hash` → Ha un uso limitato poiché funziona bene solo con i confronti di uguaglianza stretta. È sconsigliato da postgres in quanto può dare problemi sia nella gestione che nel ripristino post-crash di sistema.
  - `GiST` → Si usa per dati geometrici e bidimensionali.
  - `SP-GiST, GIN, BRIN`
- `nomeAttr` o expression sono le espressioni di attributi sulle quali si deve creare l'indice
- `ASC`/`DESC` specifica se l'indice è ordinato in modo ascendente o discendente
- `ALTER INDEX` e `DROP INDEX` permettono di modificare e rimuovere indici creati in precedenza.

> NOTA: con la localizzazione, la comparazione fra stringhe (esclusi gli operatori '=' e LIKE) dipende anche dalla localizzazione! Ha regole diverse. Un indice da costruire su un attributo varchar (testuale) e in un DB con locale ITA deve essere dichiarato aggiungendo la parola chiave `varchar_pattern_ops`.
> Uno dei problemi del locale italiano è, ad esempio, la presenza di accenti.

```
CREATE INDEX nome ON nomeTabella(nomeAttributp varchar_pattern_ops);
```

| | |
|---|---|
| *Indici multiattributo* | È possibile creare indici multiattributo, utili se sappiamo che ci sono indici che verranno spesso usati insieme, come ad esempio nel caso di JOIN.<br>! Gli indici multiattributo non sono sempre usabili: ad esempio, definendo un indice su due attributi insieme, si può usare se la clausula è in AND ma non se è in OR. |
| *Indici di espressioni* | Query con espressioni/funzioni di uno o più attributi di una tabella possono essere ottimizzate creando indici sulle medesime espressioni/funzioni (ad esempio, su LOWER(nomeins) anziché su nomeins). |
| *Costo e regola pratica d'uso* | Gli indici costano, soprattutto perhé vanno tenuti aggiornati: per ciascuna INSERT, UPDATE, DELETE bisogna aggiornarli! Quindi definisco gli indici solo in base alle query più comuni. |

#### ANALYZE

Subito dopo aver aggiunto l'indice è necessario lanciare un `ANALYZE nometabella`, per forzare l'aggiornamento delle statistiche immediatamente.

## Explain

Permette di vedere nella pratica come funziona l'ottimizzatore di query. La corretta interpretazione di un explain è piuttosto connessa: è un albero o piano di esecuzione.

- Le **foglie** sono i **nodi di scansione** alla pagine della memoria secondaria
- I **nodi** sono le **operazioni intermedie**, come **join**, **group by**, **order by**.

L'explain ha una riga per ciascun nodo dell'albero, dove troviamo l'operazione e la stima del costo. La prima riga è la stima del costo totale, ovvero ciò che l'ottimizzatore vuole ridurre al minimo.

L'unità di misura è l'accesso a memoria.

Vanno sempre letti dal fondo andando verso l'alto.

#### Explain semplice

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

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

- `cost`:
  - **0.0** è il **costo iniziale**, ovvero il costo per produrre la prima riga. → Qui è 0 perché non deve fare JOIN o WHERE
  - **185.69** è il **costo totale** per produrre tutte le righe
- `ROWS`: 8169 è il numero totale di righe del risultato
- `width`: dimensione in byte di ciascuna riga. È basato sulle statistiche, quindi potrebbe avere un numero diverso da quello reale ottenuto con l'esecuzione.

### Explain con condizione sull'id indicizzato

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

```
                                QUERY PLAN
-----------------------------------------------------------------------
Bitmap Heap Scan ON insegn (cost=18.60..132.79 ROWS=815...)
 Recheck Cond: (id < 1000)
 -> Bitmap INDEX Scan ON i1 (cost=0..18.39 ROWS=815 w=0)
    INDEX Cond: (id < 1000)
```

1. Si esegue una scansione sull'indice, ponendovi una condizione. Tramite l'indice faccio già la scrematura sull'id
2. Faccio la selezione accedendo al dato vero e proprio

### Explain con 2 condizioni in AND

```
EXPLAIN SELECT * FROM Insegn WHERE id < 1000 AND nomeins LIKE 'A1%';
```

2 condizioni, 1 indice (id), condizione in and:

```
                                QUERY PLAN
-----------------------------------------------------------------------
Bitmap Heap Scan ON Insegn (cost=18.40..134.62 ROWS=1...)
 Recheck Cond: (id < 1000)
 Filter: ((nomeins)::TEXT ~~ 'A1%'::TEXT)
 -> Bitmap INDEX Scan ON i1 (cost=0.00..18.39 ROWS=815 width=0)
   INDEX Cond: (id < 1000)
```

1. Ho comunque una prima scansione che fa utilizzo del nome dell'indice; posso farlo perché le due condizioni sono in AND
2. Nella prima memoria carica in memoria gli insegnamenti con un indice < 1000, e bisogna anche fare un filtro su nomeins

2 condizioni, 2 indici(id,nomeins), condizione in and:

```
                                QUERY PLAN
-----------------------------------------------------------------------
Bitmap Heap Scan ON insegn  (cost=15.30..22.58 ROWS=4 width=63)
  Recheck Cond: (id < 1000)
  Filter: ((nomeins)::TEXT ~~ 'A1%'::TEXT)
  -> BitmapAnd  (cost=15.30..15.30 ROWS=4 width=0)
    -> Bitmap INDEX Scan ON nomeidx (cost=0..2.65 ROWS=37 w=0)
       INDEX Cond: (((nomeins)::TEXT ~>=~ 'A1'::TEXT) AND
    ((nomeins)::TEXT ~<~ 'Am'::TEXT))
    -> Bitmap INDEX Scan ON i1 (cost=0.00..12.40 ROWS=815 w=0)
       INDEX Cond: (id < 1000)
```

1. Ho due nodi foglia: ciascuno scansiona indipendentemente l'indice con la sua condizione
2. Esegue **BitmapAnd**, che fa l'intersezione fra le righe trovate dai due nodi foglia
3. Nodo radice: carica i risultati prodotti, un recheck della condizione.

→ Il vantaggio è che quando fa l'heap scan alla radica ha molte meno righe da caricre (avendo già fatto un filtro).

### Explain con 2 condizioni in OR

```
EXPLAIN SELECT * FROM Insegn WHERE id < 1000 OR nomeins LIKE 'A1%';
```

```
                                QUERY PLAN
-----------------------------------------------------------------------
Bitmap Heap Scan ON insegn  (cost=15.47..132.25 ROWS=850
    width=63)
  Recheck Cond: ((id < 1000) OR ((nomeins)::TEXT ~~
    'A1%'::TEXT))
  Filter: ((id < 1000) OR ((nomeins)::TEXT ~~ 'A1%'::TEXT))
  -> BitmapOr (cost=15.47..15.47 ROWS=852 w=0)
    -> Bitmap INDEX Scan ON i1 (cost=0.00..12.40 ROWS=815 w=0)
       INDEX Cond: (id < 1000)
    -> Bitmap INDEX Scan ON nomeidx (cost=0.00..2.65 ROWS=37
    w=0)
       INDEX Cond: (((nomeins)::TEXT ~>=~ 'A1'::TEXT) AND
    ((nomeins)::TEXT ~<~ 'Am'::TEXT))
```

→ È uguale a prima, ma stavolta utilizza l'unione delle due foglie con il **BitmapOr**.

### Explain con condizione LIKE

```
EXPLAIN SELECT * FROM Insegn WHERE id < 1000 AND nomeins LIKE 'A1%';
```

```
                                QUERY PLAN
-----------------------------------------------------------------
Bitmap Heap Scan ON insegn  (cost=12.40..128.62 ROWS=1
     width=63)
  Recheck Cond: (id < 1000)
  Filter: ((nomeins)::TEXT ~~ '%A1'::TEXT)
  -> Bitmap INDEX Scan ON i1 (cost=0.00..12.40 ROWS=815
     width=0)
     INDEX Cond: (id < 1000)
```

→ Come visto, l'operatore LIKE non consente di sfruttare al meglio gli indici. Di conseguenza notiamo che l'ottimizatore decide usarli **solo su ID ma non su nomeins**.

### Explain con condizione JOIN

```
EXPLAIN SELECT *
        FROM Insegn I
            JOIN Inserogato IE ON ie.id_insegn = i.id
        WHERE id < 1000 OR nomeins LIKE 'A1%';
```

No indici:

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

1. Nodo hash:Costruisce l'indice hash scorrendo sequenzialmente insegnamento
2. Scansione sequenziale su insegnamento erogato con filtro che abbiamo messo nel WHERE
3. Hash join: usa l'hash per eseguire la join; per ogni riga fornita dal primo figlio cerca nel secondo figlio (hash table) la riga da unire.

→Nonostante abbiamo degli indici disponibili NON vengono usati!

Nested loop (solo indice ID)

```
                                QUERY PLAN
-----------------------------------------------------------------
Nested Loop (cost=0.28..393703.79 ROWS=14037065 width=641)
 -> Seq Scan ON inserogato ie  (cost=0.00..5970.21 ROWS=5155
     w=578)
    Filter: ((annoaccademico)::TEXT = '2013/2014'::TEXT)
 -> INDEX Scan USING i1 ON insegn i  (cost=0.28..47.99
     ROWS=2723 w=63)
    INDEX Cond: (ie.id_insegn > id)
```

Non possiamo usare l'hash poiché non c'è condizione di uguaglianza ma '>'.

1. Il nodo esterno nested loop fa una scansione sequenziale di inserogato mettendo un filtro sull'anno accademico  e una scansione sequenziale su insegn sfruttando l'id.
    a. In questo caso, posso ottimizzare anche la prima seq scan evidenziata con un indice

### Nested loop (indice su id e annoaccademico)

```
                                QUERY PLAN
-----------------------------------------------------------------
Nested Loop  (cost=82.65..391932.20 ROWS=14037065 width=641)
 -> Bitmap Heap Scan ON inserogato ie (c=82..4198 r=5155 w=578)
    Recheck Cond: ((annoaccademico)::TEXT = '2013/2014'::TEXT)
    -> Bitmap INDEX Scan ON ie_aa (cost=0.00..81.08 ROWS=5155
     w=0)
       INDEX Cond: ((annoaccademico)::TEXT = '2013/2014'::TEXT)
 -> INDEX Scan USING i1 ON insegn i (cost=0.28..48 ROWS=2723
     w=63)
    INDEX Cond: (ie.id_insegn > id)
```

→ Costruendo un indice sull'anno accademico il numero di accessi alla memoria massimi si abbassa.

### Merge JOIN

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

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

→ Ricordiamo dalla teoria che si può eseguire solamente se gli attributi della tabella sono ordinati secondo l'attributo che ci serve.
T1 è ordinata via indice, t2 non è ordinata e quindi notiamo che il DBMS fa un sort.

### Explain analyze

Versione avanzata di EXPLAIN è EXPLAIN ANALYZE, che mostra il piano di esecuzione senza registrare eventuali modifiche e mostra una stima verosimile dei tempi di esecuzione.

Si usa con parsimonia perché costoso quanto una query.

Oltre ad avere un costo espresso come numero di accesis, abbiamo anche:

- Una stima espressa in millisecondi
- Tempo di planning e di esecuzione
- Il valore di loops indica quante volte viene eseguito il nodo interno

## Controllo della concorrenza

### Anomalie

- **Perdita di aggiornamento** (lost update)
- **Lettura sporca** (dirty read)
- **Letture inconsistenti** (non-repeatable read)
- **Aggiornamento fantasma** (ghost update)
- **Inserimento fantasma** (phantom)
- **Mancata serializzazione**. Il risultato di un gruppo di transazioni (nessun abort) è inconsistentecon tutti gli ordini di esecuzioni seriali delle stesse)

Per racchiudere più operazioni in una sola transazione basta metterle all'interno di BEGIN … COMMIT/ROLLBACK.

### Multiversion Concurrency Control

Postgres utilizza la tecnica del Multiversion Concurrency Control (MVCC). È più versatile del locking a due fasi stretto. Si basa su una multiversione della base di dati:

- Ciascuna transazione **vede un'istantanea** della base di dati
- **Le letture su questa istantanea sono sempre possibili, e non sono mai bloccate** – **nemmeno se ci sono altre transazioni che stanno modificando la base di dati**
- Le scritture possono essere sospese quando, in parallelo, un'altra transazione non ancora chiusa ha modificato la sorgente dei dati che si vuole aggiornare.
- Al commit, il sistema registra sempre l'instantanea aggiornata come nuova base di dati.

### Livelli di isolamento

Cambiare i livelli di isolamento

```
SET TRANSACTION transaction_mode
ISOLATION LEVEL {SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED}
            | READ WRITE | READ ONLY
```

Postgres offre 4 livelli possibili di isolamento:

| | | |
|---|---|---|
| `Serializable` | L'intera transazione è eseguita in ordine sequenziale. | no |
| `Repeatable read` | Garantesce che i dati letti in una transazione non cambieranno a causa di altre. È più restrittivo del livello standard di repeatable read | • Inserimento fantasma |
| `Read committed` | Garantisce che qualsiasi select vede solo i dati committati. È il default | • Lettura inconsistente<br>• Aggiornamento fantasma<br>• Inserimento fantasma<br>• Mancata serializzazione |
| `Read uncommitted` | È implementata come read committed: non si può andare più in basso di read committed. | ^ same |

| *READ COMMITTED* | *REPEATABLE READ* | *SERIALIZABLE* |
|---|---|---|
| E' il livello di isolamento di default.<br><br>Le select vedono solamente i dati per cui è stato fatto un commit.<br><br>UPDATE DELETE vedono i dati come le select. Se i dati che devono essere aggiornati sono stati modificati ma non registrati, il comando deve:<br>• Attendere COMMIT o ROLLBACK della transazione ove è stato fatto il cambio<br>• Riesaminare le righe selezionate per verificare che soddisfano ancora i criteri del comando. | Differisce da read committed in quanto i comandi di una transazione vedono sempre gli stessi dati; psql associa lla transizione una istantanea della base di dati all'esecuzione del priimo comando. (= nella precedente, se c'è una commit a metà transazione, la transazione legge subto i nuovi dati).<br><br>È più stringente di quanto richiesto dallo standard SQL.<br><br>→ Risolviamo la doppia lettura: quando ho due select / update / delete esse vedono gli stessi valori.<br><br>Se i dati da modificare sono in mano ad altre transazioni succede che bisogna attendere il commit o rollback dell'altra transazione e si riesaminano le righe selezionate:<br>• In caso di rollback, update e delete possono procedere<br>• In caso di commit, i dati sono cambiati quindi si blocca tutto: "ERROR: could not serializa access due to concurrent update". | E' il livello più restrittivo: si applica uno schedule equivalente a uno schedule seriale.<br><br>Problemi:<br>• Si perde la possibilità di abortire transazioni per aggiornamenti concorrenti (??), quindi non abbiamo più anomalie di serializzazione.<br>• Riduce in maniera drastica il livello della concorrenza della base di dati. |

Il lvello di serializzazione opportuno dipende dalle operazione da eseguire.

Arrivare al livello serializzabile è infattibile nella pratica; il default di postgres è un buon compromesso. Per accedere a livelli più alti bisogna gestire il caso in cui la transazione fallisce per livello di isolamento, ovvero con errore SQLSTATE = '40001'; è sufficiente reiterare la transazione, poiché solitamente risulta molto più efficiente reiterare la transazione che eseguire dei lock espliciti.

PostgreSQL consente anche di attivare lock espliciti, ma non li vediamo :)

# Python e psychopg

Python è un linguaggio interpretato, multi paradigma e con tipizzazione dinamica. Strutture dati ad alto livello molto versatili → Facilità ad esprimere i concetti → Più produttivo.

Si usa l'API di alto livello DB-API v2.0. Ci sono 14 implementazioni diverse: noi usiamo l'implementazione psycopg2.

Psycopg è scritta in C e maschera la libreria ufficiale C libpq.

## Conversione tipi

| Python | PostgreSQL | Python | PostgreSQL | Python | PostgreSQL |
|---|---|---|---|---|---|
| None | NULL | bool | BOOL | float | REAL<br>DOUBLE |
| int | SMALLINT<br>INTEGER<br>bigint | Decimal | NUMERIC | str | VARCHAR<br>TEXT |
| date | DATE | time | TIME<br>timetz | datetime | TIMESTAMP<br>timestamptz |

## Metodi di psycopg2

| | |
|---|---|
| `connect()` | L'accesso a un database avviene tramite il metodo connect, che ritorna un oggetto connnecton. Aprire una connessione va fatto il minimo in quanto richiede tempo.<br>`connector = psycopg2.connect(host ? "dbserver.scienze.univr.it",database = "db0",user = 2user", password = "xxx")` |
| `cursor()` | Ritorna un cursore della base di dati. Un oggetto cursor permette di inviare comandi SQL al DBMS e accedere al risultato del comando restituito. |
| `commit()` | Registra la transazione corrente.<br>!! Normalmente una connessione apre una transazione al primo invio di comandi. Se non si esegue un commit() prima di chiudere, tutte le eventuali modifiche e/o inserimenti vengono persi. |
| `rollback ()` | Abortisce la transazione corrente. |
| `close()` | Chiude la connessione corrente, e implica un rollback delle operazioni non committate yet. |

## Proprietà dell'oggetto connector

| | |
|---|---|
| `autocommit` | Default: False<br>Se true, ogni comando inviato è una transazione isolata. Altrimenti, come è di default, il primo comando inviato inizia una transazione e devo ricordare di chiuderla. |
| `readonly` | Default: False<br>Se True, nella sessione non si possono inviare comandi di modifica dati. |
| `isolation_level` | Modifica il livello di isolamento, fra 'READ UNCOMMITTED','READ COMMITTED','REPEATABLE READ', 'SERIALIZABLE', 'DEFAULT'.<br>Va assegnata subito dopo la connessione. |

### Metodi del cursore

| | |
|---|---|
| `execute(comando, parametri)` | Prepara ed esegue un comando SQL usando i parametri. I parametri devono essere un dizionario o una tupla.<br>Va fatto con la tupla perché si presta al problema della SQL INJECTION!<br>Ritorna none: si recuperano i parametri via fetch*().<br>`cur.execute("CREATE TABLE test (ID SERIAL PRIMARY KEY, num integer, data varchar)")`<br>`cur.execute("INSERT INTO test (num,data) VALUES(%s,%s)",(100,"abc") )` |
| `executemany(comando, parametri)` | Prepara ed esegue un comando SQL per ciascun valore presente nella lista parametri → istanzia un template su una lista di parametri. NON EFFICIENTE: MEGLIO FARE IL FOR LOL<br><br>`cur.execute("INSERT INTO test (num,data) VALUES(%s,%s)",(100,"abc") ,( 100,"abc"),( 100,"abc"))` |
| `fetchone()` | Ritorna una tupla della tabella del risultato. Se non ci sono tuple ritorna NONE. |
| `fetchmany(<numero>)` | Ritorna una lista di tuple della tabella di lunghezza al più <numero>. |
| `rowcount()` | Metodo di sola lettura; ritorna il numero di righe prodotte dall'ultimo comando. -1 indica che non è possibile determinare il valore. |
| `statusmessage()` | Metodo di sola lettura; ritorna il messaggio ritornato dall'ultimo comando eseguito. |

### Schema di utilizzo tipico

1. Aprire una connessione tramite `conn = psycopg2.connect(..)`
   Dalla versione 2.5 è possibile usare il `with`: quando si usa una connessione con with, all'uscita del blocco viene fatto un commit automatico e la connessione non viene chiusa, mentre il cursore viene chiuso automaticamente.
2. Eventualmente modificare livello di isolamento, autocommit e readonly
3. Creare cursore tramite `curs = conn.cursor()`
4. Eseguire le operazioni
5. Se la sessione NON è in autocommit, eseguire un `conn.commit()` o `conn.rollback()`
6. Chiudere il corsore con `curs.close()` e la connessione con `conn.close()`

Con un oggetto connessione posso creare più cursori, che condivideranno la connessione.
Psychopg2 garantisce solo che le istruzioni inviate sono sequenzializzate, ma non si possono gestire transazioni concorrenti usando cursori diversi sulla stessa connessione.

→ Regoletta: usare più cursori sulla medesima connessione quando si fanno transazioni in auto-commit o tansazioni in sola lettura.

*Esempio*

```
import psycopg2
conn = psychopg2.connect(host = host, database = db, user=user, password=pg)
with conn:
      with conn.cursor() as cursore:
            cursore.execute(
                  """CREATE TABLE IF NOT EXISTS spese( id SERIAL PRIMARY KEY, data DATE NOT NULL,
                  voce VARCHAR NOT NULL,importo NUMERIC NOT NULL)"""
            )
      print('Esito della creazione della tabella spese: {:s}\n Eventuali
modifiche:{:s}'.format(cursore.statusmessage,conn.notices[-1]))
conn.close()
```