Breve guida a PostgreSQL (versione per Windows)

Gianluca Cima

13/07/2017

Indice

Capitolo 1

Introduzione

Per le esercitazioni del corso di basi di dati, si fa uso di un DBMS open-source, `PostgreSQL 9.6` che gira sotto Windows, sotto Linux e sotto Mac OS X. Tale DBMS è disponibile gratuitamente su `https://www.postgresql.org/download/`.

Questa breve guida può essere utilizzata come introduzione ad un utilizzo base del sistema, per maggiori dettagli si faccia riferimento alla documentazione ufficiale `https://www.postgresql.org/docs/9.6/static/index.html`.

Capitolo 2

Installazione

Una volta arrivati alla pagina `https://www.postgresql.org/download/`, selezionare Windows tra i vari "binary packages", e successivamente cliccare su "Download the installer" nella sezione "Interactive installer by EnterpriseDB".

Quindi, dalla pagina `https://www.postgresql.org/download/windows/`, selezionare una versione Postgres 9.6.x ed il proprio sistema operativo. Una volta concluso il download, è necessario un doppio click sul file di installazione per iniziare il processo di installazione.

Durante il processo di installazione, vi verrà chiesto di inserire una password (che va ricordata) per il *superuser* di PostgreSQL. D'ora in poi quando parleremo di password, ci riferiremo implicitamente alla password del super di PostgreSQL. Il superuser di PostgreSQL è un utente con particolari privilegi di amministrazione su tutti le basi di dati. Riguardo tutte le altre impostazioni, si consiglia di lasciare quelle di default.

In particolare, l'installazione comprende:

- Il server PostgreSQL 9.6;
- pgAdmin 4: un tool grafico per gestire le varie basi di dati;
- StackBuilder: un pacchetto che può essere usato per scaricare ed installare driver e tool aggiuntivi per PostgreSQL.

In qualsiasi momento possiamo iniziare a lavorare con PostgreSQL utilizzando la utility `psql` per la linea di comando, oppure attraverso la GUI `pgAdmin`. In questa guida, ci occuperemo soltanto della linea di comando, siccome la GUI offre un'interfaccia intuitiva e facilmente utilizzabile. Sia la linea di comando (`SQL Shell (psql)`), che l'interfaccia grafica (`pgAdmin 4`) possono essere aperti cercando tra i programmi installati nel pc. Quando

viene aperta la linea di comando premere sempre *Invio* fino a quando non viene mostrato il messaggio:

```
postgres=#
```

Per uscire basta digitare: `\q`

Capitolo 3

Creare una base di dati

Riapriamo la linea di comando e premiamo *invio* fino all'ottenimento del messaggio:

```
postgres=#
```

Ora creiamo la base di dati di nome `primodb`:

```
postgres=# create database primodb;
```

Dovrebbe apparire il messaggio: `CREATE DATABASE`

Possiamo controllare tutte le nostre basi di dati tramite il comando:

```
postgres=# \l
```

Per cominciare ad usare la base di dati appena creata digitare:

```
postgres=# \c primodb
```

Dopodichè inserire la password, e avremo il seguente messaggio: `You are now connected to database "primodb" as user "postgres".`
`primodb=#`

Per eliminare una base di dati è necessario eseguire il comando:

```
drop database nome-database ;
```

Ovviamente, il comando non va lanciato dalla stessa base di dati che si vuole eliminare, in questo caso apparirà il messaggio: `ERROR: cannot drop the currently open database`

Invece, se l'eliminazione è andata a buon fine, apparirà il seguente messaggio: `DROP DATABASE`

Capitolo 4

Gestione delle tabelle

4.1 Creare tabelle

In PostgreSQL possiamo eseguire qualsiasi comando SQL. Per creare una tabella si usa il comando SQL standard `create table` avente la forma seguente:

```
create table nome-tabella
(<lista di attributi e tipi ad essi associati>);
```

Nello scrivere un comando si può andare a capo. Se il comando occupa più linee, ad ogni *invio* viene dato un nuovo prompt fino a che non si digita il punto e virgola. Per esempio creiamo una tabella di nome *film* all'interno della base di dati *primodb* con il seguente comando:

```
primodb=# create table film (
primodb(# id integer,
primodb(# titolo varchar(255) NOT NULL,
primodb=# regista varchar(255),
primodb=# data_produzione date,
primodb=# CONSTRAINT pk PRIMARY KEY (id)
primodb=# );
```

A questo punto dovrebbe comparire il seguente messaggio: `CREATE TABLE`.

N.B.: Il punto e virgola serve ad indicare all'interprete dei comandi che il comando corrente è terminato.

Possiamo controllare tutte le tabelle all'interno di una base di dati tramite il comando:

```
\dt
```

Inoltre, per ciascuna tabella possiamo ottenere i suoi attributi (sia nome che tipo) attraverso il comando:

```
\d nome-tabella
```

4.2 Eliminare tabelle

Per eliminare una tabella dalla base di dati, si esegue il comando:

```
drop table nome-tabella;
```

Se l'eliminazione è andata a buon fine, apparirà il seguente messaggio: `DROP TABLE`.

Capitolo 5

Inserire tuple

Dopo aver creato una tabella possiamo inserirvi delle tuple utilizzando il comando *insert*. La maniera più semplice è inserire direttamente i valori:

`insert into` *`nome-tabella`*
`values`(<lista ordinata dei valori da inserire negli attributi>);

Per esempio, scrivere:

```
primodb=# insert into film values
(1, '2001:  Odissea nello spazio', 'Stanley Kubrick', '1968-04-04');
```

A questo punto dovrebbe comparire il seguente messaggio: `INSERT 0 1`.

Analogamente, possiamo inserire altre tuple come ad esempio:

```
primodb=# insert into film values
(2, 'Otto e mezzo', 'Federico Fellini', '1963-02-14');
```

Si ricorda che SQL è *case-insensitive*, vale a dire che **non** distingue tra maiscuolo e minuscolo, tranne ovviamente nelle stringhe.

Infine, per cancellare ed aggiornare tuple si utilizzano i comandi standard `UPDATE` e `DELETE`.

Capitolo 6

Interrogare una base di dati

Interroghiamo la tabella *film*, facendoci restituire ad esempio tutti i film presenti in questa tabella girati dal regista *Fellini*

```
select *
from film
where regista='Federico Fellini';
```

A questo punto dovrebbe comparire una tabella che rappresenta il risultato della query, seguito da un'indicazione riguardo il numero di tuple restituite.

Capitolo 7

Eseguire comandi memorizzati su file

Anzichè eseguire comandi SQL digitandoli su terminale è spesso più conveniente scriverli in un file di testo e poi richiamarli dall'interprete di comandi PostgreSQL. Il comando per far ciò è:

`\i` *`percorso-file`*

Usciamo (se già dentro psql) tramite il comando `\q`, e rientriamo in `psql`. Supponiamo di avere un file di nome *script.sql* sul Desktop con le seguente istruzioni SQL:

```
drop database if exists primodb;
create database primodb;
\c primodb;
create table film (
id integer,
titolo varchar(255) NOT NULL,
regista varchar(255),
data_produzione date,
CONSTRAINT pk PRIMARY KEY (id)
);
insert into film values
(1, '2001:  Odissea nello spazio', 'Stanley Kubrick', '1968-04-04');
insert into film values
(2, 'Otto e mezzo', 'Federico Fellini', '1963-02-14');
```

Possiamo caricare le seguenti istruzioni SQL direttamente da PostgreSQL con il comando:

```
postgres=# \i C:/Users/user/Desktop/script.sql
```

A questo punto dovrebbero comparire tutti i messaggi relativi alla creazione della base di dati, alla creazione delle tabella, e infine all'inserimento delle due tuple nella tabella.

Capitolo 8

Supporto transazionale in PostgreSQL

Quando la variabile `AUTOCOMMIT` è settata a `ON` (lo è così di default) si rende permanente in modo automatico qualsiasi operazione viene eseguita.

Ad esempio, creiamo una tabella di prova:

```
primodb=# create table Customer (a integer, b varchar(20));
CREATE TABLE
```

E disabilitiamo la variabile `AUTOCOMMIT` ponendola al valore `OFF`

```
primodb=# \set AUTOCOMMIT OFF
```

Ora inseriamo una tupla:

```
primodb=# insert into Customer values (10, 'Heikki');
INSERT 0 1
```

e rendiamo la modifica permanente in modo esplicito tramite il comando `COMMIT`:

```
primodb=# COMMIT;
COMMIT
```

Inseriamo ora un'altra tupla:

```
insert into Customer values (15, 'John');
INSERT 0 1
```

Controllando tutte le tuple all'interno della tabella Customer:

```
primodb=# select * from Customer;
```

dovremmo avere come risposte alla query le seguenti tuple:

```
10, 'Heikki'
15, 'John'
```

Tuttavia, le modifiche effettuate dall'ultimo `COMMIT` non sono ancora permanenti. Difatti, dando l'istruzione di `ROLLBACK`:

```
primodb=# ROLLBACK;
ROLLBACK
```

e ripondendo la stessa query:

```
primodb=# select * from Customer;
```

dovremmo avere come risposte alla query solamente la tupla:

```
10, 'Heikki'
```

Infine, rimettiamo la variabile `AUTOCOMMIT` al suo valore di deafult:

```
primodb=# \set AUTOCOMMIT ON
```

In qualsiasi momento possiamo verificare il valore attuale di questa variabile digitando il comando:

```
\echo :AUTOCOMMIT
```

N.B. Il supporto transazionale di PostgreSQL aderisce in pieno allo standard SQL, differentemente dallo *store engine* `InnoDB`, il quale offre il pieno supporto transazionale a MySQL. Infatti, ad esempio, in `InnoDB` quando si esegue un rollback non vengono ritratti gli statement `create table` e `drop table` (cosa che invece avviene in PostgreSQL).