PostgreSQL 10

il database Open Source piu' avanzato del pianeta

ing. Luca Ferrari, PhD

fluca1978
https://fluca1978.github.io

<2018-05-22 mar>

Attribution-NonCommercial-ShareAlike 4.0

Cos'è?

PostgreSQL è un Object Relational Database Management System (ORDBMS).
Object non è da intendersi relativamente al paradigma *OOP* quanto al fatto che un utente *puo' estendere il database con i propri "oggetti"*.
Ad esempio, PostgreSQL non supporta i *query-hints* tanto famosi in altri sistemi commerciali: per gli sviluppatori questa funzionalità non ha senso e non è mai stata (e mai verrà) implementata.

Approccio simile a quello di OpenBSD!

PostgreSQL è il naturale discendente di Ingres (dopo, *post*, gres), un database accademico/sperimentale inventato dal prof. Michael Stonebreaker e commercializzato indicativamente nello stesso spazio temporale di Oracle (1989 circa).
Tutto inizia alla Berkely University of California.

Quand'è?

Nome	Anno di produzione	Note
POSTGRES	1989	successore di Ingres
POSTGRES95	1994	viene aggiunto un interprete SQL
Postgres 1	1995	
PostgreSQL 6	1997	
PostgreSQL 7	2000	foreign keys, join e no-crash
PostgreSQL 8	2005	port Windows nativo
PostgreSQL 9	2010	Replication
PostgreSQL 10	2017	...

The copyright of Postgres 1.0 has been loosened to be freely modifiable and modifiable for any purpose. Please read the COPYRIGHT file. Thanks to Professor Michael Stonebraker for making this possible. — Release 1.0

Di chi è?

PostgreSQL non è guidato da nessun vendor e di conseguenza non ha una lista di clienti da soddisfare. Questo significa che una feature sarà implementata in PostgreSQL solo se ha senso dal punto di vista *tecnico* e *scientifico*.
PostgreSQL is a non-commercial, all volunteer, free software project, and as such there is no formal list of feature requirements required for development. We really do follow the mantra of letting developers scratch their own itches.
Ad esempio, PostgreSQL non supporta i *query-hints* tanto famosi in altri sistemi commerciali: per gli sviluppatori questa funzionalità non ha senso e non è mai stata (e mai verrà) implementata.

Approccio simile a quello di OpenBSD!

Per chi è?

Licenza BSD (anche per il logo):

```
PostgreSQL Database Management System
(formerly known as Postgres, then as Postgres95)

Portions Copyright (c) 1996-2017, PostgreSQL Global Development Group

Portions Copyright (c) 1994, The Regents of the University of California

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose, without fee, and without a written agreement
is hereby granted, provided that the above copyright notice and this
paragraph and the following two paragraphs appear in all copies.
```

Quanto costa?

Il progetto è *Open Source* ed è *Free*.
Esistono diverse varianti commerciali che si differenziano dalla versione *mainstream* per funzionalità (es. replica multi-master, query multi-nodo, ecc.) e per un *costo* che dipende direttamente dal vendor.
Molti degli sviluppatori della versione mainstream sono in realtà anche sviluppatori di un qualche vendor.

Qual'era?

In PostgreSQL i numeri di versione *erano* a tre cifre separati da punto:

- *release brand* (es. 7, no-crash, 8 MS Windows portability, 9 Replication)
- *year release* (da quanti anni si ha questo brand)
- *minor release* (rilasciata circa ogni quattro mesi o in presenza di gravi problemi di sicurezza o consistenza)
  - 9.5.1 # major version 9.5, minor version 1
    - 9.5.1 compatibile con 9.5.2, 9.5.3, ...
  - 9.6.2 # major version 9.6, minor version 2
    - incompatibile con 9.5.x!

Le prime due cifre rappresentavano una major version e quindi erano segnale di possibile incompatibilità.

Qual'é?

Dalla versione 10 la numerazione è diventata a due sole cifre.
La cifra prima del punto rappresenta la *major version*.
Questo rappresenta una incompatibilità semantica con il passato: gli script che facevano affidamento alla versione devono modificare la propria logica!
L'unico metodo affidabile è quello di considerare che ogni cifra viene usata con un formato `printf(2)` pari a `%02d` e che il numero dell'anno da ora in avanti è sempre zero.

Versione	Major	Minor	Internal
9.5.1	9.5	1	090501
9.6.4	9.6	4	090604
10.0	10	0	100000
10.1	10	1	100001

Quanto dura?

Ogni *major release* di PostgreSQL viene manutenuta per 5 anni dalla data di primo rilascio. Una volta che una release raggiunge la End Of Life nessun pacchetto binario sarà piu' rilasciato ma potrebbe essere aggiornato (in retrocompatibilità) l'albero dei sorgenti (a discrezione degli sviluppatori e senza alcuna garanzia).
Ecco un esempio delle prossime "scadenze":

Version	First release date	EOL date
9.6	September 2016	September 2021
9.5	January 2016	January 2021
9.4	December 2014	December 2019
9.3	September 2013	September 2018
9.2	September 2012	September 2017

Quanto è?

E' difficile *spaventare* una istanza PostgreSQL!

Dato	Limite massimo
Maximum Database Size	Unlimited
Maximum Table Size	32 TB
Maximum Row Size	1.6 TB
Maximum Field Size	1 GB
Maximum Rows per Table	Unlimited
Maximum Columns per Table	250 - 1600 depending on column types
Maximum Indexes per Table	Unlimited

Chi lo sviluppa?

Chiunque, anche tu! Non esiste un *benevolent dictator*!

Tre livelli principali di sviluppatori:

1. core team: 5 membri storici che si occupano di gestire il ciclo di rilascio e tutte le questioni "spinose" (mancanza di consenso, disciplina, ecc)
   - Peter Eisentraut
   - Magnus Hagander
   - Tom Lane
   - Bruce Momjian
   - Dave Page
2. major contributors: *sviluppatori fidati* (commit access) che lavorano abitualmente alle funzionalità del progetto
3. contributor: chiunque fornisca una patch, una proposta, una traduzione, ...

- hacker emeritus: chi ha lavorato in passato al progetto con particolare successo (Josh Berkus, Marc G. Fournier, Thomas G. Lockhart, Vadim B. Mikheev, Jan Wieck)

Come si sviluppa?

Si utilizza `git` (migrato da CVS intorno al 2009).

```
% git clone git://git.postgresql.org/git/postgresql.git

Cloning into 'postgresql'...
...
% du -hs postgresql
356M    postgresql
% git rev-list --all --count
59672
```

- Linguaggio di programmazione principale: `C`, stile BSD (`style(9)`).
- Strumenti di sviluppo ben noti: `gcc`, `gmake`, `autoconf`, ecc.
- Strumenti (anche Perl) ad-hoc per mantenere il codice: `pgindent`, `git_changelog`, `make_ctags`, ecc.

Da quanto si sviluppa?

Da molto tempo (oltre 30 anni), ma non si torna prima della versione 1.01 di Postgres95, ramo di sviluppo 6 del Postgres "attuale":

```
% git log `git rev-list --max-parents=0 HEAD`

commit d31084e9d1118b25fd16580d9d8c2924b5740dff
Author: Marc G. Fournier <scrappy@hub.org>
Date:   Tue Jul 9 06:22:35 1996 +0000

    Postgres95 1.01 Distribution - Virgin Sources
```

Terminologia

- Ogni istanza di PostgreSQL gestisce un cluster.
- Un cluster è formato da uno o piu' database, ogni database può essere a sua volta scomposto in uno o piu' schema (*namespace logico*), che a sua volta può contenere uno o piu' oggetti (*tabelle, trigger, indici, funzioni*, ...). Ogni database è totalmente isolato dall'altro.
- Ogni oggetto può appartenere ad uno e un solo tablespace (*spazio fisico*).
- Il cluster mantiene anche le informazioni relative agli utenti e ai permessi. Gli utenti vengono chiamati ruoli e rappresentano sia singole utenze che gruppi (quindi un ruolo può contenere altri ruoli).

In linea con la filosofia Unix, PostgreSQL vuole svolgere un compito solo nel miglior modo possibile: gestire i dati. E' compito del DBA documentarsi e aggiungere le estensioni necessarie a seconda del caso d'uso (es. pooling).

Cluster di Database

Un singolo cluster quindi può gestire un albero di oggetti a granularità molto specifica:

- *database 1*
  - *schema* `public` (default)
  - *schema 1*
    - *tabelle, trigger, indici -> tablespace disco SSD*
    - *tabelle, trigger, indici -> tablespace disco lento*
  - *schema 2*
    - *tabelle, trigger, indici -> tablespace disco SSD*
    - *tabelle, trigger, indici -> tablespace disco lento*
- *database 2*
  - *schema* `public` (default)
  - *schema 1*
    - *tabelle, trigger, indici -> tablespace disco SSD*
    - *tabelle, trigger, indici -> tablespace disco lento*
  - *schema 2*
    - *tabelle, trigger, indici -> tablespace disco SSD*
    - *tabelle, trigger, indici -> tablespace disco lento*

Analogia tra Cluster e OS

L'isolamento e la gestione di un cluster ricorda molto quella di un sistema operativo:

PostgreSQL	Unix
cluster	OS
ruolo	utente
tablespace	mount point
database	home directory
schema	sottodirectory (es `$HOME/bin`)
oggetto (tabella, trigger, funzione, ...)	file, script

Schema a Processi: Connessioni

PostgreSQL utilizza uno schema a processi: *ogni connessione viene gestita da un sottoprocesso creato ad-hoc*.

Processi vs Thread

Ci sono svariate ragioni per preferire uno schema a processi rispetto ad uno a thread: isolamento e portabilità sono le principali.

Il processo principale è denominato postmaster; ogni volta che questo riceve una richiesta di connessione si effettua una *fork* di un processo postgres (denominato anche backend) delegato a gestire la connessione. Una connessione può essere TCP/IP oppure su socket locale.

Schema a Processi: IPC

Siccome ogni processo è *fisicamente* isolato, ma piu' connessioni possono dover condividere i dati, PostgreSQL utilizza un'area shared memory ove mantiene i dati. Tale zona di memoria è visibile a tutti i processi postgres. La shared memory viene organizzata in *pagine dati* che rappresentano la copia in memoria dei dati persistenti su disco. Vi sono una serie di processi di utilità che si occupano di scaricare/caricare i dati dalla *shared memory* e dal disco.

File System

PostgreSQL si appoggia al filesystem per lo stoccaggio dei dati in maniera persistente..
Questo offre diversi vantaggi, fra i quali la possibilità di un tuning molto raffinato circa le opzioni di funzionamento del filesystem (replica, journaling, ecc.). Dall'altra parte, il filesystem deve essere affidabile, pena il rischio di perdita dati.

Tipologie di installazione

E' possibile installare PostgreSQL:

- mediante installer ufficiale
- mediante pacchetti binari della propria installazione
- compilando l'albero dei sorgenti

Tutte le prove qui mostrate sono state effettuate su una macchina virtuale con quattro dischi da 2GB utilizzati come spazio dati:

```
$ uname -a
FreeBSD olivia 11.1-RELEASE FreeBSD 11.1-RELEASE
$ mount
/dev/ada1p1 on /mnt/data1 (ufs, local, soft-updates)
/dev/ada2p1 on /mnt/data2 (ufs, local, soft-updates)
/dev/ada3p1 on /mnt/data3 (ufs, local, soft-updates)
/dev/ada4p1 on /mnt/data4 (ufs, local, soft-updates)
```

Installazione (FreeBSD)

Compilazione dai ports:

```
# cd /usr/ports/databases/postgresql10-server
# make PREFIX=/opt/postgresql-10 BATCH=yes install clean
```

oppure il pacchetto binario:

```
# pkg install postgresql10-server-10.1_1
```

Precedenti versioni vengono rimosse dall'installazione binaria! I moduli `contrib` vanno rimossi a mano!

In FreeBSD il database viene gestito dall'utente di sistema `postgres`:

```
# id postgres
uid=770(postgres) gid=770(postgres) groups=770(postgres)
```

altri sistemi operativi creano utenti simili (`psql`, `pgsql`, ecc.).

Configurazione avvio servizio (OS)

I parametri di configurazione dipendono ovviamente dal sistema operativo, ad esempio su FreeBSD le variabili di `rc.conf` sono visibili da `/usr/local/etc/rc.d/postgresql`:

```
#  postgresql_enable="YES"
#  postgresql_data="/var/db/postgres/data96"
#  postgresql_flags="-w -s -m fast"
#  postgresql_initdb_flags="--encoding=utf-8 --lc-collate=C"
#  postgresql_class="default"
#  postgresql_profiles=""
```

Avvio tramite pg_ctl

Il comando pg_ctl consente di avviare una istanza in modo *controllato* senza dover dipendere da script del sistema operativo. L'unico parametro obbligatorio è la $PGDATA.

```
% sudo -u postgres  pg_ctl -D /mnt/data3/pgdata start -l /tmp/postgresql.log
```

Analogamente, per fermare lo stesso cluster si usa:

```
% sudo -u postgres  pg_ctl -D /mnt/data3/pgdata stop
waiting for server to shut down.... done
server stopped
```

Arresto del server

In generale *non si può rompere una istanza PostgreSQL*, anche in caso di un arresto forzato questa saprà come riavviarsi.
Le modalità di arresto "controllato" di una istanza sono tre:

- `smart` (default) impone che il cluster attenda la disconessione volontaria di tutti i client e dei processi di backup, dopodiché si spegnerà;
- `fast` forza una disconnessione di tutti i client e dei processi di backup, e si spegne;
- `immediate` è l'equivalente di un `SIGKILL`.

Solo il caso `immediate` produce un *crash* del sistema, e quindi al riavvio il sistema dovrà ripercorrere i WAL per riportarsi ad uno stato coerente.

Arresto e connesioni in ingresso

Nei meccanismi di spegimento `smart` e `fast` il cluster non accetterà nessuna ulteriore connessione in ingresso attendento invece il termine di quelle già attive.

```
% sudo -u postgres  pg_ctl -D /mnt/data1/pgdata stop -m smart
waiting for server to shut down.....................
pg_ctl: server does not shut down
HINT: The "-m fast" option immediately disconnects sessions rather than
waiting for session-initiated disconnection.
```

e se si tenta di collegarsi:

```
% psql -h localhost -U luca testdb
psql: FATAL:  the database system is shutting down
```

Processi

```
% pstree
\-+- 02057 postgres /usr/local/bin/postgres -D /mnt/data1/pgdata
 |--= 02058 postgres postgres: pg9.6.5: logger process    (postgres)
 |--= 02060 postgres postgres: pg9.6.5: checkpointer process    (postgres)
 |--= 02061 postgres postgres: pg9.6.5: writer process    (postgres)
 |--= 02062 postgres postgres: pg9.6.5: wal writer process    (postgres)
 |--= 02063 postgres postgres: pg9.6.5: autovacuum launcher process    (postgres)
 |--= 02064 postgres postgres: pg9.6.5: stats collector process    (postgres)
```

- `postgres` (denominato anche postmaster), accetta le connessioni;
- `wal_writer` scrive i record WAL;
- `writer` (*bgwriter*) scrive le pagine dati;
- `logger` gestisce i log testuali del server;
- `checkpointer` schedula e gestisce i checkpoint;
- `autovacuum launcher` gestisce i processi *autovacuum*;
- `stats collector` raccoglie le statistiche dinamiche.

PGDATA

PostgreSQL utilizza il filesystem del sistema operativo per salvare i dati in modo persistente.
In particolare una directory specifica, denominata PGDATA, viene usata per contenere tutti gli oggetti PostgreSQL. Tale directory deve essere inizializzata opportunamente (creazione struttura directory, impostazione dei permessi, ecc.) tramite il programma `initdb`.
Un cluster può servire una sola PGDATA alla volta. La directory PGDATA deve essere protetta opportunamente da accessi involontari di altri utenti del sistema.

initdb

Creazione di una directory per la memorizzazione del database (alcuni sistemi operativi lo fanno automaticamente al momento dell'installazione binaria):

```
# mkdir /mnt/data1/pgdata
  && chown postgres:postgres /mnt/data1/pgdata
$ initdb --data-checksum --encoding="UTF-8" \
         --pwprompt                         \
         -D /mnt/data1/pgdata/
```

`initdb` deve essere eseguito da un utente non privilegiato, le opzioni indicano:

- `--data-checksum`: abilita il controllo sulle pagine dati del database;
- `--encoding`: default encoding di ogni database se non sovrascritto;
- `--pwprompt`: richiede la password del superutente di PostgreSQL (comodo per non impostarla dopo);
- `-D`: l'opzione principale, indica dove si troveranno i dati.

initdb: errore comune

Se la directory specificata come `PGDATA` per il cluster non è vuota `initdb` si rifiuta di continuare (si noti bene che non esiste un'opzione *force*):

```
...
initdb: directory "/mnt/data1/pgdata" exists but is not empty
If you want to create a new database system, either remove or empty
the directory "/mnt/data1/pgdata" or run initdb
with an argument other than "/mnt/data1/pgdata".
```

initdb: risultato

Se il comando `initdb` completa con successo, viene riportato un modo per avviare il cluster con la directory `PGDATA` appena specificata:

```
$ initdb
...
Success. You can now start the database server using:

    pg_ctl -D /mnt/data1/pgdata/ -l logfile start
```

Struttura di PGDATA

La directory `$PGDATA` contiene diversi file e directory, in particolare:

- `PG_VERSION`: file di testo con la versione che serve il cluster;
- `postgresql.conf`: configurazione principale del cluster;
- `pg_hba.conf`: file di accesso al database;
- `base`: directory sotto la quale si troveranno tutti i database;
- `global`: directory che contiene dati inter-database (es. cataloghi di sistema);
- `pg_stat` e `pg_stat_tmp`: informazioni per le statistiche di sistema;
- `pg_tblscp`: link ai vari tablespace (oggetti fuori da `base`);
- `pg_wal` e `pg_xact`: rispettivamente WAL e commit log.

Per altre informazioni vedere Storage File System Layout.

File fisici su disco

Ogni *oggetto* con dei dati (es. tabella) viene memorizzato su disco in un file con nome pari al suo *Object IDentifier* (`OID`) numerico. Questo ha il vantaggio di:

- essere indipendente dal nome *logico* e dalla relativa codifica e charset;
- essere *univoco* indipendentemente da quante volte si cambia nome all'oggetto nel database.

Ogni file dato viene *spezzato* in chunk da `1 GB` massimo, quindi dello stesso oggetto si possono avere piu' file nominati con `oid`, `oid.1`, `oid.2`, ecc. Solitamente i file crescono in dimensione di `8 kB` alla volta (ossia della dimensione di una pagina dati).

oid2name

L'utility `oid2name` (modulo *contrib*) consente di esplorare la struttura dati su disco.

```
% oid2name -H localhost -U postgres
All databases:
    Oid  Database Name  Tablespace
----------------------------------
  12646       postgres  pg_default
  12645      template0  pg_default
      1      template1  pg_default
```

Gli *oid* visualizzati in questo caso corrispondono al nome fisico su disco dei database (che a sua volta corrisponde al nome della directory ove sono contenuti i dati):

```
% sudo ls -l /mnt/data1/pgdata/base
drwx------  2 postgres  ... 1
drwx------  2 postgres  ... 12645
drwx------  2 postgres  ... 12646
```

oid2name (2)

Esploriamo il database `template1` su disco e cerchiamo di capire cosa contiene:

```
% sudo stat /mnt/data1/pgdata/base/1/1259
104 65579 -rw------- 1 postgres postgres ...
```

A cosa corrisponde l'oggetto file 1259?

```
% oid2name -H localhost -U postgres -d template1 -o 1259
From database "template1":
  Filenode  Table Name
----------------------
      1259    pg_class
```

ATTENZIONE: si deve specificare a quale database si fa riferimento, poiché gli stessi oid possono essere riciclati in database differenti

oid2name (3)

E se si vuole trovare una tabella dato il suo nome?

```
% oid2name -H localhost -U postgres -d template1 -t pg_class
From database "template1":
  Filenode  Table Name
----------------------
      1259    pg_class
```

oid2name va ad interrogare il catalogo di sistema per trovare le informazioni necessarie.

pg_relation_filepath()

La funzione di sistema `pg_relation_filepath()` permette di trovare un file partendo dal nome di una tabella considerando il nome della tabella:

```
# SELECT pg_relation_filepath( 'pg_class'::regclass );
 pg_relation_filepath
----------------------
 base/1/1259
```

Il percorso è *relativo a $PGDATA*.

Connessione al servizio

Il file `pg_hba.conf` contiene le informazioni su quali metodi di autenticazione, quali utenti, quali host remoti e quali database sono accessibili per la connessione. Si può editare questo file prima di avviare il servizio (se si è impostata una password per `postgres` superuser) o anche in seguito.

```
# tipo  database    utente    da dove         metodo
local   all         all                       trust
host    all         all       127.0.0.1/32      md5
```

$pg_{hba.conf}$ vs sudoers

Il file `pg_hba.conf` è simile al file `sudoers`, e come tale va gestito scrupolosamente. La parola `all` indica tutti gli utenti/database (a seconda di dove è messa). Il metodo `trust` non richiede autenticazione e non va usato!

Il modulo $auth_{delay}$ permette di mitigare attacchi a forza bruta ritardando le risposte di errore del server in caso di fallita autenticazione.

Avvio del servizio

Una volta che i primi pezzi sono al loro posto, è possibile avviare il servizio:

```
# service postgresql start
```

e se tutto va a buon fine...

```
# psql -h localhost -U postgres -l
Password for user postgres:
   Name     |  Owner   | Encoding |
------------+----------+----------+
 postgres   | postgres | UTF8     |
 template0  | postgres | UTF8     |
 template1  | postgres | UTF8     |
(3 rows)
```

Database template

Quando viene inizializzata `PGDATA` il sistema crea due database chaimati *template*:

- `template0`: la copia principale del template;
- `template1`: la copia usata in default.

Ogni votla che viene creato un nuovo database le impostazioni di base sono copiate da `template1` (che funge da *skel* directory).
E' facoltà del DBA impostare `template1` opportunamente per far si che la creazione di nuovi database abbia una base comune riconosciuta (es. schemi, linguaggi, ecc.).
`template0` è la copia di sicurezza del template, qualora si "sporchi" troppo `template1`.

Database templating

I due database template non svolgono alcuna funzione particolare se non quella di essere usati come possibili punti di origine di un nuovo database. In default, se non specificato, PostgreSQL copia `template1`, mentre `template0` dovrebbe essere lasciato *vergine* per operazioni particolari quali restore (può servire un database vuoto a cui collegarsi).
E' possibile creare quanti database template si vuole e istruire il comando `CREATE DATABASE` per usare altri template oltre `template1`. Si noti però che un database template non accetta connessioni durante la creazione, quindi questo non è un meccanismo di *clonazione* dei database!

psql

Il pacchetto *client* contiene un interprete da riga di comando, denominato `psql` che consente di collegarsi al database e svolgere *tutti* i compiti necessari.

```
% psql -h localhost -U postgres template1
Password for user postgres:
psql (10.1)
Type "help" for help.

template1=#
template1=# \q
%
```

I parametri di linea comando sono:

- `-h`: host a cui collegarsi (hostname, indirizzo ip);
- `-U`: utente con cui collegarsi (*ruolo* PostgreSQL);
- database a cui collegarsi (es. `template1`).

psql: connection URI

Oltre a specificare ogni singolo parametro della conessione tramite opzioni di comando, psql consente di utilizzare un URI per la connessione, ad esempio:

```
% psql postgresql://postgres@localhost:5432/template1
Password:
psql (10.1)
Type "help" for help.

template1=#
```

Parametri ulteriori possono essere specificati nell'URL (dopo ?), come ad esempio:

```
postgresql://postgres@localhost/template1?sslmode=require
```

psql: prompt

In modo simile alla shell, il prompt di `psql` mostra:

- il database al quale si è collegati (`template1`);
- un # se si è superuser o > se si è utenti normali.

Si esce da `psql` con `\q`.

Quale versione del server?

`psql` mostra all'avvio la propria versione (client) ma con una query è possibile capire anche la versione del server:

```
template1=# SELECT version();
 PostgreSQL 10.1 on amd64-portbld-freebsd11.1, ...
(1 row)
```

La funzione speciale `version()` viene compilata al momento del build del pacchetto binario.

psql, ruoli, database

In default `psql` cerca di collegarsi a un database che ha lo stesso nome utente dell'utente che esegue il comando stesso, con un ruolo che ha lo stesso nome. In altre parole:

```
% id -p
uid     luca

% psql
psql: FATAL:  role "luca" does not exist
```

corrisponde a:

```
% psql -h localhost -U luca luca
```

psql: ruoli, database e variabili di ambiente

Quando non viene specificato un utente e/o un database `psql` cerca di collegarsi a quanto stabilito dalle variabili di ambiente `PGUSER` e `PGDATABASE` (e le relative `PGHOST` e `PGPORT`):

```
% export PGUSER=foo PGDATABASE=myDB
% psql
psql: FATAL:  no pg_hba.conf entry for host "[local]",
              user "foo",
              database "myDB", SSL off
```

psql: aiuto

All'interno di `psql` ci sono due tipologie di aiuto:

- *aiuto sugli statement SQL*: si ottiene con `\h`
  - `\h` senza argomenti mostra tutti i comandi SQL disponibili;
  - `\h STATEMENT` mostra l'aiuto dello statement SQL specificato;
- *aiuto su `psql`*: si ottiene con `\?` e mostra tutti i comandi speciali di `psql`. Tutti i comandi `psql` iniziano con backslash (es. `\d`).

psql: evitare la password

`psql` consente di impostare un file di credenziali (`$HOME/.pgpass`) per collegarsi a uno specifico database/host con uno specifico utente senza dover digitare una password. Ogni riga nel file contiene:

- host e porta a cui collegarsi;
- database a cui collegarsi (* per tutti);
- username e password con cui collegarsi.

Il file non deve essere leggibile da altri utenti (es. permessi Unix 600).

```
% cat ~/.pgpass
127.0.0.1:5432:template1:postgres:xxxxxxxx
```

Il file deve avere permessi `rw` per il solo proprietario. Inoltre si deve specificare la porta a cui collegarsi dopo l'hostname!

psql: configurazione utente

`psql` consente di specificare delle configurazioni utente nel file `$HOME/.psqlrc`. Tutti i comandi (`psql` compatibili) specificati in tale file vengono eseguiti prima di fornire il prompt all'utente.
Utile per impostare variabili, prompt, formati di output, ecc.

psql: ~/.psqlrc di esempio

```
\set HISTFILE ~/.psql_history_ :DBNAME
\set ON_ERROR_ROLLBACK on
\set ON_ERROR_STOP     on
\x
\set PROMPT1 '[%n @ %/ on %m] %l %x %# '
```

Il prompt corrisponde a: *username* (%n), *database* (%/), *hostname* (%m), *linea* (%l), *stato transazione* (%x) e *prompt superutente o utente normale* (%#).
A differenza di `ON_ERROR_STOP` che fa quello che il nome suggerisce, il parametro `ON_ERROR_ROLLBACK` fa il suo opposto: » When set to on, if a statement in a transaction block generates an error, the error is ignored and the transaction continues.

psql: variabili

All'interno di una sessione `psql` si possono impostare delle variabili da usare per una sostituzione letterale:

```
> \set my_limit 10
> \set my_table foo

> SELECT *
  FROM :my_table
  LIMIT :my_limit
```

L'interpolazione di una variabile avviene usando il carattere `:` prima del nome della variabile stessa. Se occorre fare una interpolazione di stringa si deve usare il carattere `:` e il nome della variabile va fra apici (doppi o semplici):

```
> \set my_string blah
> INSERT INTO foo VALUES( :'my_string' );
```

psql: variabili di visualizzazione

Le variabili che controllano la visualizzazione sono gestite tramite \pset, alcuni esempi:

```
\set QUIET 1          -- non visualizzare i comandi \

\pset expanded on     -- abilita versione estesa
\pset title 'ITEMS'   -- titolo della query
...query...
\pset title           -- disabilita titolo per query successive
\pset expanded off    -- disabilita versione estesa
```

psql: comandi della shell

In maniera simile ai principali editor il carattere ! passa il comando ad una shell:

```
> \! uname -a
  FreeBSD olivia 11.1-RELEASE FreeBSD 11.1-RELEASE #0 r321309: ...
```

psql: NULL

E' possibile visualizzare i campi con valori NULL (che altrimenti non sono visualizzati) usando la variabile speciale `null` (siccome è una opzione di visualizzazione va impostata con \pset):

```
> \pset null 'Null Value'
Null display is "Null Value".

> SELECT 1, NULL, 2;
 ?column? |  ?column?  | ?column?
----------+------------+----------
        1 | Null Value |        2
```

La funzione quote_nullable è maggiormente portabile (client differenti):

```
> SELECT 1, NULL, 2, quote_nullable( NULL );
 ?column? |  ?column?  | ?column? | quote_nullable
----------+------------+----------+----------------
        1 | Null Value |        2 | NULL
```

Ruoli e Utenti

Dalla versione 8.1 in poi PostgreSQL non distingue piu' fra utenti e gruppi ma usa il concetto di ruolo che rappresenta entrambi:

- un ruolo può rappresentare un utente;
- ad un ruolo si possono aggiungere altri ruoli (e quindi rappresenta un gruppo).

Per collegarsi ad un database occorre sempre un ruolo, ossia un utente PostgreSQL (che è logicamente separato da quello del sistema operativo). Quando viene inizializzato un cluster viene creato un ruolo superutente per permetterne la gestione (negli esempi `postgres`).

Ruoli e connessioni

L'accesso al sistema viene garantito dalla configurazione del file `pg_hba.conf`, che rappresenta il listato di quali utenti, database e host possono ottenere una connessione.
E' sufficiente un `SIGHUP` per far riconoscere al cluster le modifiche ai permessi di connessione!

Vedere i ruoli

Il catalogo pg_roles contiene le informazioni sui ruoli e le loro proprietà:

```
# SELECT rolname, rolsuper, rolcreatedb, rolcanlogin
  FROM pg_roles;
      rolname         | rolsuper | rolcreatedb | rolcanlogin
----------------------+----------+-------------+-------------
 pg_monitor           | f        | f           | f
 pg_read_all_settings | f        | f           | f
 pg_read_all_stats    | f        | f           | f
 pg_stat_scan_tables  | f        | f           | f
 pg_signal_backend    | f        | f           | f
 postgres             | t        | t           | t
```

E' possibile creare utenti/gruppi/ruoli con privilegi di super utente, possibilità di creare nuovi database e di collegarsi o no al cluster.

Creare i ruoli

Il comand SQL `CREATE ROLE` (o da terminale `createuser`) consente di creare un nuovo utente/gruppo. Ad esempio si supponga di voler gestire un database di una applicazione definendo due utenti: uno applicativo e uno amministrativo/interattivo:

```
# CREATE ROLE my_app
  WITH NOLOGIN
  CONNECTION LIMIT 1
  PASSWORD 'xxx';
CREATE ROLE

# ALTER ROLE my_app WITH LOGIN;

# CREATE ROLE luca
  WITH CREATEDB LOGIN PASSWORD 'xxxxx'
  IN ROLE my_app;
CREATE ROLE
```

Ora il ruolo `my_app` funge sia da utente che da gruppo a cui `luca` appartiene. Si noti l'uso di `ALTER ROLE` per correggere un errore.

Creare i ruoli (2)

I ruoli appena creati risultano ora:

```
# SELECT rolname, rolsuper, rolcreatedb, rolcanlogin, rolconnlimit
  FROM pg_roles;
     rolname      | rolsuper | rolcreatedb | rolcanlogin | rolconnlimit
------------------+----------+-------------+-------------+--------------
 postgres         | t        | t           | t           |           -1
 my_app           | f        | f           | t           |            1
 luca             | f        | t           | t           |           -1
 ...
```

Creare i ruoli (3)

E' ora possibile configurare il file .pgpass per i nuovi ruoli:

```
% cat ~/.pgpass
127.0.0.1:5432_template1:postgres:postgres
localhost:5432:template1:luca:xxxxxx
localhost:5432:template1:my_app:xxxxx
```

Si ricordi che è possibile usare * per host, porta e database. Questo semplifica il deployment di nuovi database, ma dall'altro lato rende piu' complesso censire e controllare i database a cui si accede.

IN ROLE, INHERITS, ADMIN

I ruoli possono essere annidati con la clausola `IN ROLE`.
La clausola `INHERITS` eredita (dinamicamente) i permessi del ruolo da cui si eredita, ma è necessario usare `SET ROLE` per *switchare* al ruolo voluto.
La clausola `ADMIN` identifica il ruolo come *amministratore del gruppo* e gli consente di aggiungere altri ruoli a questo gruppo.

Gruppi & login

- INHERITS non aggiunge il LOGIN!

```
# CREATE ROLE developers WITH LOGIN;
-- ok to login
# CREATE ROLE dev1 IN ROLE developers LOGIN;
-- FATAL:  role "dev2" is not permitted to log in
# CREATE ROLE dev2 IN ROLE developers;
```

- il gruppo non include gli altri utenti quando usato in pg_hba.conf:

```
# pg_hba.conf
host    all   developers   127.0.0.1/32   trus

% psql -h localhost -U dev1 testdb
FATAL:  no pg_hba.conf entry for host "127.0.0.1", user "dev1", database "testdb",
```

Gruppi & login (2)

- occorre usare i gruppi con un +:

```
# pg_hba.conf
host    all   +developers   127.0.0.1/32   trus
```

- per disabiltiare un gruppo e abilitare gli altri utenti al login:

```
host    all     +developers     127.0.0.1/32     reject
host    all     all             127.0.0.1/32     trust
```

La vista pg_shadow

In analogia con i sistemi Unix, è disponibuile una vista `pg_shadow` che riassume i dati degli account utente (ruoli) comprensivi di password. Se la password inizia con `md5` la password è stata cifrata con tale algoritmo. Ovviamente occorre essere superutente per poter interrogare la vista.

```
# SELECT usename, usecreatedb, usesuper,
         userepl, passwd
  FROM pg_shadow;
-[ RECORD 1 ]-----------------------------------
usename     | postgres
usecreatedb | t
usesuper    | t
userepl     | t
passwd      | md53175bce1d3201d16594cebf9d7eb3f9d
```

La vista pg_hba_file_rules

La vista pg_hba_file_rules consente di effettuare il *debugging* delle impostazione di autenticazione e di capire chi può collegarsi al database:

```
# SELECT line_number, type, database, user_name, auth_method
  FROM pg_hba_file_rules;
line_number| type  | database |   user_name    | auth_method
-----------+-------+----------+----------------+-------------
        84 | local | {all}    | {all}          | trust
        86 | host  | {all}    | {+developers}  | reject
        87 | host  | {all}    | {+stocker_app} | trust
```

Creare un database

Usando il comando SQL `CREATE DATABASE` è possibile aggiungere un nuovo database (e *opzionalmente* aggiungere un commento per indicare lo scopo del database):

```
# CREATE DATABASE testdb
  WITH OWNER 'luca';

# COMMENT ON DATABASE testdb IS 'A test database';
```

Alternativamente si può usare il comando shell `createdb`:

```
% createdb --owner='luca' -e -h localhost -U postgres testdb 'A test database'
CREATE DATABASE testdb OWNER luca;
COMMENT ON DATABASE testdb IS 'A test database';
```

Vedere i database disponibili

Il catalogo pg_database contiene le informazioni circa i database presenti nel sistema:

```
# SELECT datname FROM pg_database;
  datname
-----------
 postgres
 testdb
 template1
 template0
```

Alternativamente si può usare l'opzione -l in psql:

```
% psql -h localhost -U postgres -l
...
Name                | testdb
Owner               | luca
Encoding            | UTF8
Collate             | C
Ctype               | C
Access privileges   |
```

Eliminare un database

Avendo i privilegi corretti, si può usare il comando SQL `DROP DATABASE` o il comando shell `dropdb`:

```
% psql -h localhost -U postgres template1
# DROP DATABASE testdb;
DROP DATABASE
```

```
% dropdb -h localhost -U postgres testdb
```

Creare una tabella

L'istruzione SQL è `CREATE TABLE`.

```
% psql -h localhost testdb
> CREATE TABLE persona(
 pk serial,
 nome varchar(20),
 cognome varchar(20),
 codice_fiscale varchar(16),
 PRIMARY KEY(pk),
 UNIQUE(codice_fiscale)
 );
```

Modificare una tabella

Ci sono varie istruzioni, in particolare `ALTER TABLE` che consente di agire sulla tabella e sui relativi vincoli:

```
> ALTER TABLE persona ADD COLUMN eta integer;
> ALTER TABLE persona ADD CHECK ( eta > 0 and eta < 120 );
```

Come è fatta la tabella?

Il comando speciale di psql \d consente di ispezionare una tabella e i suoi vincoli:

```
> \d persona
                                  Table "public.persona"
     Column      |          Type           |                       Modifiers
-----------------+-------------------------+-------------------------------------------
 pk              | integer                 | not null default nextval('persona_pk_seq'
 nome            | character varying(20)   |
 cognome         | character varying(20)   |
 codice_fiscale | character varying(16)   |
 eta             | integer                 |
Indexes:
    "persona_pkey" PRIMARY KEY, btree (pk)
    "persona_codice_fiscale_key" UNIQUE CONSTRAINT, btree (codice_fiscale)
Check constraints:
    "persona_eta_check" CHECK (eta > 0 AND eta < 120)
```

SELECT not dual !

In PostgreSQL non esiste la "famosa" tabella `dual` e lo statement `SELECT` fa esattamente quello che ci si aspetta:

```
> SELECT 1 FROM dual;
ERROR:  relation "dual" does not exist

> SELECT 1;
 ?column?
----------
        1
```

NULL (breve ripasso)

Tutte le seguenti ritornano NULL:

```
> SELECT 1 + NULL AS sum_null,
         NULL = NULL AS eq_null,
         NULL <> NULL AS ne_null,
         '' || NULL AS cat_null,
         '' = NULL as empty_string;
```

quello che spesso si vuole è IS NULL:

```
> SELECT NULL IS NULL as eq_null,
      ( '' || NULL ) IS NULL AS str_null;

eq_null  | t
str_null | t
```

Dollar quoting

PostgreSQL permette l'uso del dollar quoting (simile all'operatore qq di Perl):

- si può usare un tag con nome arbitrario purché racchiuso fra due simboli $;
- il tag va usato per l'apertura e la chiusura;
- la stringa fra tag viene sottoposta ad escape automatico.

```
> SELECT $qq$Perche' l'hai scritto?$qq$;
        ?column?
------------------------
 Perche' l'hai scritto?
```

Si può usare anche uno statement SQL come tag (es. $SELECT$).

Non dollar quoting

Sono supportate altre sintassi per il quoting:

- doppio apice per figurarne uno solo (no, \' non fa quello che ci si aspetterebbe);
- `quote_literal` per inserire il corretto quoting di una stringa.

```
> SELECT 'E'' arrivata la primavera!',
quote_literal( 'E'' arrivata la primavera!' ); --non interpolato!

?column?      | E' arrivata la primavera!
quote_literal | 'E'' arrivata la primavera!'
```

Camel Case

Lo standard SQL non ammette il camel case, e chiede l'uso di UPPERCASE. PostgreSQL utilizza il lowercase per tutti gli identificatori (il risultato non cambia).

```
> CREATE TABLE Foo( Bar int, bAz int );
-- diventa
-- create table foo( bar int, baz int);
```

Se si vuole usare il cAmeLcAsE si deve indicare ogni operatore *sempre* fra doppi apici (sconsigliato):

```
> CREATE TABLE "Foo"( "Bar" int, "bAz" int );
> SELECT "Bar", "bAz" FROM "Foo";
```

Camel Case: quote_ident

La funzione speciale quote_ident sa cosa deve fare con gli identificativi che non siano minuscoli, e quindi è bene usarla ogni volta che ad esempio si generano statement in automatico:

```
> SELECT quote_ident( 'foo' ),
         quote_ident( 'FOO' ),
         quote_ident( 'Foo' );
-[ RECORD 1 ]------
quote_ident | foo
quote_ident | "FOO"
quote_ident | "Foo"
```

NULL quoting

Esiste una funzione quote_nullable che si comporta come quote_literal ma restituisce la stringa speciale NULL qualora il suo argomento sia NULL. ATTENZIONE: restituisce la *stringa* NULL, non il valore NULL!

```
> SELECT quote_literal( '' || NULL ),
         quote_ident( '' || NULL ),
         quote_nullable( '' || NULL );

quote_literal  |
quote_ident    |
quote_nullable | NULL

> SELECT quote_nullable( '' || NULL ) = 'NULL';
 ?column?
----------
 t
```

NULL concatenation

Ogni cosa *aggiunta* a NULL fornisce come risultato un (diverso) NULL. La funzione concat permette di concatenare molte cose, ignorando eventuali NULL.

```
> SELECT 'before' || NULL || 'after' AS no_ignoring,
testdb-> concat( 'before', NULL, 'after' ) AS ignoring;
 no_ignoring |  ignoring
-------------+-------------
             | beforeafter
```

concat_ws()

Oltre all'operatore || e alla funzione concat() vi è la funzione concat_ws() che accetta come primo argomento un *separatore* da interporre ciclicamente fra gli argomenti:

```
 > SELECT concat_ws( '-', 'Here', 'is', 'a', 'text', NULL, '.' );
    concat_ws
-----------------
 Here-is-a-text-.
```

Il valore NULL viene igonrato (come in concat()) a meno che non sia usato come *separatore*, nel qual caso tutto il risultato si annulla.

Camel Case vs Text

ATTENZIONE: ogni apice doppio " viene considerato come delimitatore di nome camel-case di colonna, non di testo!

SHOW e SET

Il comando speciale SHOW mostra il valore di alcuni parametri *run-time*, e analogamente il parametro SET consente di impostarne il valore.

```
# SHOW enable_seqscan;
 enable_seqscan
----------------
 off

# SET enable_seqscan TO 'on';
```

Il comando SET modifica solo il comportamento della sessione, a meno che non sia specificato LOCAL che allora persiste solo alla fine (positiva o meno) di una transazione (SET normale svanisce se la transazione abortisce).

current_setting e set_config

Le due funzioni speciali current_setting() e set_config() corrispondo al comando SHOW e SET e consentono di ottenere e impostare il valore di un parametro di configurazione:

```
> SELECT current_setting( 'enable_seqscan' );
 current_setting
-----------------
 on

> SELECT set_config( 'enable_seqscan', 'off', false );
-- terzo parametro true se valido solo per la transazione corrente!
 set_config
------------
 off
```

Interagire con gli altri processi

Ci sono un serie di funzioni per inviare dei segnali specifici:

- `pg_cancel_backend()` e `pg_terminate_backend()` inviano rispettivamente un `SIGTERM` e `SIGKILL` ad un altro processo backend;
- `pg_rotate_logfile()` invia un `SIGHUP` al processo logger collector per indicare di creare un nuovo file di log;
- `pg_reload_conf()` invia un `SIGHUP` al postmaster per rileggere il file di configurazione;
- `pg_xlog_switch()` forza la creazione di un nuovo segmento di WAL;
- `pg_current_xlog_flush_location()` fornisce il *LSN* attuale;
- `pg_is_in_recovery()` e `pg_is_in_backup()` indicano se il cluster è in recovery o backup mode.

Addormentarsi: pg_sleep

La funzione pg_sleep permette di sospendere il backend corrente per *circa* il numero di secondi specificato (accetta anche le frazioni di secondo):

```
> \timing
Timing is on.
> SELECT pg_sleep( 2.5 );

Time: 2573.297 ms
```

Generare dati random (suggerimenti)

E' possibile generare delle sequenze di dati con:

- `generate_series( a, b )` che genera una serie di numeri interi compresi fra `a` e `b` (ma anche una serie di `timestamp`);
- `random()` che generare un `float` casuale;
- `md5( t )` che calcola l'hash della stringa `t`.

Ad esempio per generare una chiave *univoca* lunga 16 caratteri:

```
SELECT substring(
         md5(
           ( random() * random() )::text )
        from 1 for 16 );
```

mentre per generare dei giorni

```
> SELECT day::date
  FROM generate_series( '2018-01-01', '2018-03-01',
                        '1 day'::interval ) day;
```

Numerici

Sono presenti i tipi interni `integer`, `smallint`, `bigint`.
I tipi esatti con virgola sono `decimal` e `numeric`, da definirsci con una *precisione* (cifre significative) e una *scala* (cifre dopo la virgola). Ad esempio `123.456789` ha precisione 9 e scala 6.
I tipi inesatti `real` e `real` sono a virgola mobile con arrotondamenti.
I tipi `serial`, `bigserial`, `smallserial` sono collegati ai relativi tipi interi e alle sequenze.

Numerici precisi e imprecisi

I tipi di dato `real` e `real` sono imprecisi, i tipi `numeric` e `decimal` sono precisi (dipendentemente dalla scala):

```
> SELECT 1 - 0.0000000000000001::float AS float_result,
  1 - 0.0000000000000001::numeric AS numeric_result,
  1 - 0.0000000000000001::decimal AS decimal_result,
  1 - 0.0000000000000001::decimal( 10, 8 ) AS decimal_fix_result;
-[ RECORD 1 ]------+------------------
float_result       | 1
numeric_result     | 0.9999999999999999
decimal_result     | 0.9999999999999999
decimal_fix_result | 1.00000000
```

Numeric: scala e precisione

I tipi numerici sono definibili con due vincoli: *scala* e *precisione*. Il primo valore (precisione) indica *quante cifre totali comporranno il numero*, il secondo (scala) quante cifre saranno presenti dopo la virgola.
Ne consegue che scala e precisione vincolano il massimo valore del dato stesso:

```
#     numeric(p, s) --> 10^(p-s)
> SELECT 39::numeric(2,1);
ERROR:  numeric field overflow
DETAIL:  A field with precision 2,
         scale 1 must round to an
         absolute value less than 10^1.
```

Valute

Il tipo `money` rappresenta una valuta con cifre dopo la virgole. Può essere convertito in `numeric` senza perdita di precisione, ma convertirlo in altri formati (intero, reale) produce perdita di precisione.

```
> SELECT 10.25::money::numeric(5, 2);

numeric | 10.25
```

Stringhe

I tipi principali sono:

- `char(n)` stringa di lunghezza fissa con blank a destra, tutto ciò che supera `n` viene troncato (standard SQL);
- `varchar(n)` memorizza fino a `n` caratteri ma senza padding (lunghezza variabile);
- `text` stringa illimitata (fino a 1GB)
- `name` stringa di 64 byte ad uso interno (es. nomi degli identificatori).

Da notare che il `char(n)` non viene interpretato dagli operatori come ci si potrebbe aspettare:

```
> SELECT length( 'abc'::char(10) );
 length
--------
      3
```

Gli spazi a destra nei `char` non sono considerati significativi, mentre lo sono per i dati a lunghezza variabile.

I blank (a destra) non sono significativi!

Si ricordi che solo nei tipi a lunghezza variabile i *rigth-blank* sono considerati significativi!

Tuttavia un `char(n)` viene effettivamente memorizzato (e restituito all'esterno) come un *blank-padded* di lunghezza *n*.

```
> SELECT
    length( rpad( 'abc', 10 ) ) AS text_length,
    length( rpad( 'abc', 10 )::char( 10 ) ) AS char_length;

text_length | 10
char_length | 3
```

Funzioni basilari per la manipolazione di stringhe

```
> SELECT length( 'Perl' );
   -- 4, le stringhe iniziano da 1!!!!

> SELECT upper( 'luca' ), -- LUCA
         lower( 'LUCA' ); -- luca

> SELECT 'Perl' || ' v5.27' AS concat,
   replace( 'Perl 5', '5', '6') AS replace,
   substr( 'Perl v5.27', 1, 4 ) AS substr,
   regexp_match( 'Perl v5.27',
                 '\d+\.\d+' ) AS version_array,
   regexp_replace( 'Perl v5.27',
                   '\d+\.\d+',
                   '6' );

concat         | Perl v5.27
replace        | Perl 6
substr         | Perl
version_array  | {5.27}
regexp_replace | Perl v6
```

format(): il ritorno della sprintf(3)

La funzione format() permette la costruzione di stringhe in modalità analoga a quanto fatto da psrintf(3), tuttavia i formati hanno placeholder limitati:

- %s stringa e valori numerici;
- %I identificatore SQL;
- %L valore SQL.

```
> SELECT format( 'SELECT %I FROM %I WHERE x = %L ORDER BY %s',
                          'y', 'Coordinates', 30, 'distance' );
                            format
-----------------------------------------------------------
 SELECT y FROM "Coordinates"
 WHERE x = '30' ORDER BY distance

> SELECT format( '%3$s, %2$s, %1$s', 10, 20, 30 );
   format
------------
 30, 20, 10
```

Bytea

bytea è una stringa binaria illimitata con un header variabile da 1 a 4 byte.
Usa generalmente il formato esadecimale per input e output.

Tempo

Ci sono tipi di `date`, `time`, `timestamp` con e senza timezone. Il tipo `timestampz` è un sinonimo di `timestamp with time zone`.
Si noti che, nonostante lo standard SQL lo permetta, il timezone ha senso solo quando è presente sia la data che l'ora.

datestyle

PostgreSQL fa del suo meglio per interpretare le stringhe come date, ma qualora non sia sufficiente occorre variare il parametro datestyle:

```
> SELECT '19/7/2018'::date;
ERROR:  date/time field value out of range: "19/7/2018"
LINE 2: select '19/7/2018'::date;

> show datestyle
testdb-> ;
 DateStyle
-----------
 ISO, MDY

> set datestyle TO 'ISO,DMY';
> SELECT '19/7/2018'::date;
    date
------------
 2018-07-19
```

Manipolazione del tempo

Ci sono alcune regole:

1. la differenza fra due `timestamp` è sempre un `interval`, e di conseguenza aggiungere un `interval` ad un `timestamp` produce un altro `timestamp`;
2. la somma/differenza di due `interval` è sempre un `interval` (non un `timestamp`);
3. non si sommano/sottraggono due `timestamp`;
4. la differenza fra due `date` è sempre un valore intero, ovvero il numero di giorni di differenza, e di conseguenza aggiungere un intero ad una data produce un'altra data;

Esempi di manipolazione del tempo

```
> SELECT '2017-11-15T22:00'::timestamp - interval '3 days';
      ?column?
---------------------
 2017-11-12 22:00:00

> SELECT interval '1 month' - interval '3 days';
   ?column?
---------------
 1 mon -3 days

> SELECT '2017-11-15'::date + 3 - '2017-10-12'::date;
 ?column?
----------
       37
```

Funzioni per manipolare il tempo

- `age()` calcola la differenza fra due `timestamp`, con un solo argomento la differenza dal momento corrente (`now()`);
- `extract()` e l'equivalente `date_part()` estraggono una informazione da un `date` o `timestamp`, quali ad esempio:
  - `century`, `millenium`, `epoch`;
  - `year`, `month`, `quarter`, `day`, `dow` (day of week), `doy` (day of year);
  - `hour`, `minute`, `seconds`, `milliseconds`, `microseconds`;
- `date_trunc()` aggiusta la precisione del `date` o del `timestamp` alla parte specificata, mettendo a zero ogni campo piu' preciso;
- `justify_days`, `justify_hours` aggiusta il periodo temporale a frazioni intere di 30 giorni e 24 ore, `adjust_interval` applica entrambi.

Esempio di funzioni di manipolazione del tempo

```
> SELECT age( current_timestamp, '2017-12-25T23:59:59'::timestamp ) as to_xmas,
        extract( 'year' from current_date ),
        date_part( 'hours', current_time ),
        justify_days( '35 days'::interval );
-[ RECORD 1 ]+---------------------------------
to_xmas      | -1 mons -4 days -05:20:00.051659
date_part    | 2017
date_part    | 18
justify_days | 1 mon 5 days
```

extract vs date_part

Le due funzioni `extract()` e `date_part()` svolgono sostanzialmente la stessa funzione ma:

- in `estract` la parte da estrarre può non essere quotata, mentre in `date_part` deve essere una stringa;
- in `extract` si usa la parola chiave `from`, in `date_part` gli argomenti sono separati da `,`.

```
> SELECT
  date_part( 'year', '2018-01-31'::date ) AS anno,
  extract( 'month' from '2018-01-31'::date ) AS mese, -- stringa
  extract( day from '2018-01-31'::date ) AS giorno;   -- non stringa
-[ RECORD 1 ]
anno   | 2018
mese   | 1
giorno | 31
```

date_part non solo per le date ma anche per il tempo!

```
> SELECT extract( hour from current_time ),
         date_part( 'hour', current_time ),
         date_trunc( 'hour', current_timestamp );

date_part  | 18
date_part  | 18
date_trunc | 2018-02-17 18:00:00+01
```

Costruttori del tempo

Oltre al cast esplicito di stringhe, sono disponibili i costruttori `make_date()`, `make_time()` e `make_timestamp()`. Esiste anche un `make_interval()` che ha una serie di valori di default che possono quindi essere specificati uno alla volta:

```
> SELECT make_date( 2017, 12, 25 ) as xmas_date,
         make_time( 23, 59, 59 ) as santa_arrives,
         make_interval( days => 25 );
-[ RECORD 1 ]-+-----------
xmas_date     | 2017-12-25
santa_arrives | 23:59:59
make_interval | 25 days
```

Costruttori speciali del tempo

Esistono delle *stringhe particolari* per specificare date e tempo che possono essere usate come input per un campo date e timestamp:

```
> SELECT 'epoch'::timestamp AS epoch,
         'now'::timestamp AS now,
         'yesterday'::timestamp AS yesterday,
         'today'::timestamp AS today,
         'tomorrow'::timestamp AS tomorrow,
         'allballs'::time AS allballs;  -- solo time!

epoch     | 1970-01-01 00:00:00
now       | 2018-02-19 18:44:55.015914
yesterday | 2018-02-18 00:00:00
today     | 2018-02-19 00:00:00
tomorrow  | 2018-02-20 00:00:00
allballs  | 00:00:00
```

Si noti che a parte now tutti gli altri campi sono *troncati* alla data!

Convertire il tempo

Le funzioni `to_char()` convertono un tempo (`timestamp`, `date`, `interval`) in una stringa, mentre `to_date()` e `to_timestamp()` convertono una stringa in un tempo. I parametri di formattazione del tempo solitamente usati sono:

- `HH`, `HH24`, `MI`, `SS`, `MS`, `US` per il tempo da ore fino a microsecondi;
- `Y`, `mon` (uppercase, lowercase, upperfirst), `MM`, `day` (uppercase, lowercase, uppperfirst), `DD` per anno, mese, giorno del mese.

```
> SELECT to_char( current_date, 'YYYY-Mon-DD' ),
         to_char( current_date, 'DD/MM/YYYY' ),
         to_date( '25/12/2017', 'DD/MM/YYYY' );
-[ RECORD 1 ]--------
to_char | 2017-Nov-21
to_char | 21/11/2017
to_date | 2017-12-25
```

Intervalli: sintassi ammesse

Il tipo `interval` ammette almeno tre sintassi differenti:

```
> SELECT
   current_timestamp + '1 month 3 days 4 hours 25 minutes 12 seconds' AS pg_interval,
   current_timestamp + '0-1 3 4:25:12' AS sql_interval,
   current_timestamp + 'P0Y1M3DT4H25M12S' AS iso_interval;

pg_interval  | 2018-03-22 22:59:42.618705+01
sql_interval | 2018-03-22 22:59:42.618705+01
iso_interval | 2018-03-22 22:59:42.618705+01
```

Booleani

Il tipo di dato `boolean` rappresenta un booleano che accetta come input:

- `TRUE` oppure una delle stringhe `'t'`, `'true'`, `'on'`, `'yes'`;
- `FALSE` oppure una delle stringhe `'f'`, `'false'`, `'off'`, `'no'`;

Bit

Sono supportati due tipi di stringhe bit: a lunghezza fissa `bit(n)` e variabile `bit varying(n)`.
Gli operatori logici sulle stringhe di bit sono definiti.

```
> SELECT 35::bit(8);
   bit
----------
 00100011

> SELECT ( 35::bit(8) | 12::bit(8) )::int;
 int4
------
   47
```

Le stringhe di bit devono avere la stessa lunghezza per essere utilizzate con gli operatori!

Bit vs Boolean

Le stringhe di bit possono essere usate per memorizzare piu' valori booleani in un solo campo. Le funzioni speciali `set_bit` e `get_bit` consentono di agire sui singoli bit specificandone l'offset (e l'eventuale valore):

```
> SELECT 10::bit(4) AS bit_array,
   get_bit( 10::bit(4), 2 ) AS second_bit,
   set_bit( 10::bit(4), 2, 0 ) AS result_second_bit;

bit_array          | 1010
second_bit         | 1
result_second_bit | 1000
```

Indirizzi di rete

PostgreSQL fornisce tre tipi di dato:

- `inet` memorizza un indirizzo IPv4 o IPv6 con relativa rete (ammetta valori sballati di rete!);
- `cidr` analogo, ma non memorizza la maschera di rete (solo valori corretti);
- `macaddr` memorizza un indirizzo hardware.

Effettuano la matematica degli indirizzi (con interi)!

```
> SELECT '192.168.222.100/24'::inet + 300;
      ?column?
--------------------
 192.168.223.144/24

> SELECT '192.168.222.0/24'::cidr + 300;
     ?column?
-------------------
 192.168.223.44/24
```

Punti nel piano

Il tipo speciale `point` consente di specificare delle coordinate di un punto nel piano. Esistono degli operatori particolari che consentono di stabilire la distanza fra due punti:

```
-- ATTENZIONE: uso di apici!
> SELECT '(10, 20)'::point <-> '(30, 40)'::point AS distance;
     distance
------------------
 28.2842712474619
```

Box, Circle, Path & friends...

Esistono tipi costruiti *sopra* al concetto di `point`: `box` (rettangolo), `circle` (circonferenza), `path` (unione di punti). Altri costruttori consentono di costruire poligoni, segmenti, linee, ecc.

```
> SELECT
  box( '(10,20)'::point, '(20, 30)'::point ) AS rettangolo,
  circle( '(50,50)'::point, 10 ) AS circoferenza_r10,
  line( '(1,1)'::point, '(-1,-1)'::point ) AS linea,
  lseg( '(1,1)'::point, '(-1,-1)'::point ) AS segmento,
  polygon( path '( (1,1), (2,2), (3,3) )' ) AS triangolo;
-[ RECORD 1 ]----+--------------------
rettangolo       | (20,30),(10,20)
circoferenza_r10 | <(50,50),10>
linea            | {1,-1,0}
segmento         | [(1,1),(-1,-1)]
triangolo        | ((1,1),(2,2),(3,3))
```

I cast a `point` non sono obbligatori per molte funzioni!

Sequenze

Le sequenze vengono create con uno statement di `CREATE SEQUENCE`, possono essere agganciate ai tipi di dato interi (`smallint`, `int`, `bigint`) ma `bigint` è il default.
Alla sequenza si possono assegnare valori *minimi* e *massimi*, se non specificati si prendono quelli di default del tipo di dato della sequenza.
La sequenza può essere *agganciata* ad una colonna di tabella con `OWNED BY`, nel qual caso il drop della tabella/colonna implica anche il drop della sequenza.
Le sequenze possono ciclare i valori (in caso di overflow) con `CYCLE`.

Sequenze: esempio

```
> CREATE SEQUENCE my_seq
   AS int
   INCREMENT BY -1    -- si muove all'indietro
   MINVALUE 0         -- fino a zero
   MAXVALUE 100       -- da 100
   START WITH 100     -- partendo da 100
   CYCLE;             -- e ricomincia da 100 quando arriva a 0
```

Sequenze: funzioni

- `nextval( 'my_seq' )` fornisce il prossimo valore della sequenza senza rispettare eventuali ROLLBACK;
- `currval( 'my_seq' )` fornisce il valore corrente senza alterare la sequenza;
- `setval( 'my_seq', 55 )` imposta a 55 il valore corrente della sequenza.

```
> SELECT setval( 'my_seq', 1 ); --imposta a 1 il valore della sequenza

> SELECT currval( 'my_seq' ), currval( 'my_seq' );
 currval | currval
---------+---------
       1 |       1

> SELECT nextval( 'my_seq' ), nextval( 'my_seq' );
 nextval | nextval
---------+---------
       0 |     100  -- la sequenza ha ciclato!
```

Restart di una sequenza

```
> ALTER SEQUENCE my_seq RESTART;
```

serial

Il tipo di dato speciale `serial` (e `bigserial`, `smallserial`) corrisponde ad una generazione automatica di sequenza:

```
> CREATE TABLE foo( pk serial );
```

```
> CREATE TABLE foo( pk integer );
> CREATE SEQUENCE foo_seq OWNED BY foo.pk;
> ALTER TABLE foo ALTER COLUMN pk SET DEFAULT nextval( 'foo_seq' );
```

Colonne IDENTITY

Le colonne IDENTITY agganciano automaticamente una sequenza di generazione alla colonna stessa.
La differenza con il tipo SERIAL è che questo è solo un *collante sintattico* per generare una sequenza e agganciare il valore di default di una colonna al prossimo risultato di tale sequenza.
La gestione però rimane a carico del DBA: sequenza e colonna sono gestite separatamente.
Con il tipo IDENTITY il database sa, semanticamente, che la colonna è agganciata ad una sequenza e ne tiene traccia. Non occorre ad esempio conoscere il nome della sequenza, ma solo lavorare sulla colonna.
Inoltre questa sintassi è aderente agli standard SQL.

Colonne IDENTITY: generazione dei valori

A differenza del tipo `serial` le colonne `identity` consentono due tipologie di generazione dei valori:

- `ALWAYS` scarta sempre eventuali valori esplicitati durante una `INSERT` e li sostituisce con quelli generati dalla sequenza;
- `BY DEFAULT` permette l'inserimento di valori esplicitati (si comporta come `serial` in questo senso).

Da notare che `ALWAYS` può essere superato se l'`INSERT` specifica `OVERRIDING SYSTEM VALUE`.

Esempio complessivo: creazione della tabella

```
> CREATE TABLE foo(
  i integer GENERATED ALWAYS AS IDENTITY,
  j integer GENERATED BY DEFAULT AS IDENTITY,
  k serial,
  t text );
```

Vengono generate le sequenze:

```
public | foo_i_seq                | sequence | luca
public | foo_j_seq                | sequence | luca
public | foo_k_seq                | sequence | luca
```

Esempio complessivo: popolamento

```
> INSERT INTO foo( t ) VALUES( 'Tutti autogenerati' );

> INSERT INTO foo( i, j, k, t ) VALUES( 1,2,3, 'i=1, j=2, k=3' );
ERROR:  cannot insert into column "i"
DETAIL:  Column "i" is an identity column defined as GENERATED ALWAYS.

> INSERT INTO foo( i, j, k, t )
  OVERRIDING SYSTEM VALUE
  VALUES( 1,2,3, 'i=1, j=2, k=3' );

> INSERT INTO foo( j, k, t )
  VALUES( 4,5, 'i=auto, j=4, k=5' );
```

che produce il risultato

```
> SELECT * FROM foo;
 i | j | k |         t
---+---+---+-------------------
 1 | 1 | 1 | Tutti autogenerati
 1 | 2 | 3 | i=1, j=2, k=3
 2 | 4 | 5 | i=auto, j=4, k=5
```

Restart di una colonna IDENTITY

```
> ALTER TABLE foo ALTER COLUMN i RESTART;
```

Temporary Tables

Le *Temporary Tables* sono tabelle che persistono (in memoria) solo per la durata della sessione (o della transazione).
Analogamente, ogni indice costruito su una tabella temporanea viene automaticamente eliminato assieme alla tabella.
Rappresentano l'analogo dei file temporanei di un sistema operativo!

Temporary Tables: caratteristiche

La tabella appartiene ad uno schema particolare, quindi non può essere creata in uno schema imposto dall'utente.
Una tabella temporanea con lo stesso nome di una persistente nasconde quest'ultima (a meno che non sia referenziata completamente) per tutta la vita della tabella temporanea.
Il processo `autovacuum` non può analizzare una tabella temporanea, quindi i comandi `ANALYZE` e/o `VACUUM` devono essere tutti impartiti manualmente!

Creare una tabella temporanea

```
> CREATE TEMPORARY TABLE persona( nome text );
-- nasconde una 'public.persona' se esiste!
```

Manipolare una tabella temporanea in transazione

Una tabella temporanea può essere *agganciata* ad una transazione con la clausola `ON COMMIT`:

- `PRESERVE ROWS` è il default, la tabella si comporta come una normale tabella (eccetto che sparirà con la sessione);
- `DELETE ROWS` al commit della transazione la tabella è troncata automaticamente;
- `DROP` al commit della transazione la tabella viene distrutta senza attendere che la sessione finisca.

Tabella temporanea e transazione: esempio

```
> BEGIN;
> CREATE TEMPORARY TABLE foo( f text ) ON COMMIT DELETE ROWS;
> INSERT INTO foo VALUES( 'hello' );
> INSERT INTO foo VALUES( 'world' );
> SELECT COUNT(*) FROM foo;
 count
-------
     2
(1 row)

> COMMIT;
> SELECT COUNT(*) FROM foo;
 count
-------
     0
```

Tabella temporanea e transazioni: esempio (2)

```
> BEGIN;
> CREATE TEMPORARY TABLE foo(
                pk serial PRIMARY KEY,
                t text )
    ON COMMIT DROP;
> INSERT INTO foo(t) SELECT 'foo' || generate_series(1, 1000) ;
> SELECT COUNT(*) FROM foo;

count | 1000

> COMMIT;
> SELECT COUNT(*) FROM foo;
ERROR:  relation "foo" does not exist
LINE 2: SELECT COUNT(*) FROM foo;
```

Unlogged Tables

Una tabella può essere marcata come *unlogged*: essa sarà persistente ma i suoi dati no. Di fatto una tabella *unlogged* non viene memorizzata nei WAL, e quindi non sopravvive ad un crash e non viene nemmeno replicata. Di conseguenza PostgreSQL effettua un truncate di una *unlogged* table ogni volta che il sistema riparte dopo un crash.
Il vantaggio è che, non dovendo sincronizzarsi con i WAL, la tabella risulta piu' veloce ma al tempo stesso insicura rispetto ai crash.
Gli indici creati su una tabella *unlogged* sono anch'essi automaticamente unlogged.

Unlogged table: creazione

```
> CREATE UNLOGGED TABLE baz( b text );
```

Estrarre un sottoinsieme dei dati

Una `SELECT` permette di specificare un meccanismo di *sampling* mediante la clausola `TABLESAMPLE` al quale si specifica l'algoritmo e la percentuale di tabella da campionare. Il risultato è l'applicazione dei vincoli della query sul campionamento della tabella. Gli algoritmi al momento disponibili sono:

- `SYSTEM` campionamento basato sui blocchi di dati (tutte le tuple della pagina dati selezionata sono ritornate);
- `BERNOULLI` campionamento sulle tuple.

In entrambi i casi la percentuale di tuple ritornate è una approssimazione del valore specificato.

Esempio di tablesample

```
> SELECT count(*)
  FROM evento
  TABLESAMPLE SYSTEM ( 0.2 )
  WHERE mod( pk, 2 ) = 0;
```

Si seleziona il 20% delle pagine dati della relazione, e poi si filtra per valore di pk pari. Il sampling è applicato prima delle clausole di filtro!

Operatore LATERAL

In unq SELECT l'uso di LATERAL consente di aggregare una sottoquery in modo che sia valutata per ogni tupla della query principale.

Esempio di utilizzo: senza LATERAL

Ad esempio, in questa query:

```
> SELECT * FROM evento
  JOIN (
      SELECT pk
      FROM evento
      WHERE mod( pk, 2 ) <> 0 ) evs
  ON evs.pk = evento.pk;
```

viene prima valutata la query innestata (subquery) e poi la query principale. Questo viene confermato dai nodi di esecuzione della query:

```
  QUERY PLAN
-------------------------------------------------------------------------------
 Merge Join  (cost=0.86..178533.01 rows=2099132 width=22)
   Merge Cond: (evento.pk = evento_1.pk)
   ->  Index Scan using evento_pkey on evento  (cost=0.43..68235.63 rows=2109680 wi
   ->  Index Only Scan using evento_pkey on evento evento_1  (cost=0.43..78784.03 r
         Filter: (mod(pk, 2) <> 0)
```

Esempio di utilizzo: con LATERAL

La stessa query riscritta con LATERAL (attenzione ci vuole una condizione di join, in questo caso fittizia):

```
> SELECT * FROM evento
  JOIN LATERAL (
            SELECT pk FROM evento
            WHERE mod( pk, 2 ) <> 0 ) evs
  ON TRUE;
```

In questo caso per ogni tupla della tabella esterna si valuta la query annidata:

```
 QUERY PLAN
-------------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..72655670831.83 rows=4428496797760 width=22)
   ->  Seq Scan on evento  (cost=0.00..34531.80 rows=2109680 width=18)
   ->  Materialize  (cost=0.00..63775.86 rows=2099132 width=4)
         ->  Seq Scan on evento evento_1  (cost=0.00..45080.20 rows=2099132 width=4
               Filter: (mod(pk, 2) <> 0)
```

Foreign Keys

Vegono specificate con `REFERENCES` o `FOREIGN KEY` e supportano piu' colonne. Entrambe le sintassi permettono di specificare una strategia di `NATCH`:

- `MATCH FULL` non sono tollerati valori nulli nella tabella corrente se non sull'intero insieme di valori che referenziano un'altra tabella;
- `MATCH SIMPLE` (default) se una colonna locale è nulla allora non si pretende che ci sia corrispondenza con i valori dell'altra colonna.

e anche una strategia di modifica alla tabella referenziata `ON DELETE` e `ON UPDATE`:

- `NO ACTION` produce un errore e blocca la `DELETE` o la `UPDATE` perchè spezzerebbero la chiave esterna;
- `RESTRICT` uguale a `NO ACTION` ma non posticipabile;
- `CASCADE` propaga l'aggiornamento/cancellazione anche alla tabella referenziante;
- `SET NULL` imposta a null i valori delle colonne referenzianti;
- `SET DEFAULT` imposta al valore di default le colonne referenzianti (ma deve esistere una tupla referenziata con tali valori).

Foreign Keys: esempio

```
> CREATE TABLE a(
   pk SERIAL PRIMARY KEY,
   a1 int,
   a2 int,
   UNIQUE (a1),
   UNIQUE(a2) );

> CREATE TABLE b( pk SERIAL PRIMARY KEY,
    b1 int,
    b2 int,
    FOREIGN KEY (b1) REFERENCES a(a1) ON DELETE CASCADE ON UPDATE SET NULL,
    FOREIGN KEY (b2) REFERENCES a(a2) ON DELETE CASCADE ON UPDATE SET NULL
    );
```

Foreign key: effetti

Si inseriscono un po' di tuple e le si cancellano:

```
> INSERT INTO a( a1, a2 ) VALUES (1,1), (2,2);
> INSERT INTO b( b1, b2 ) VALUES (1,1), (2,2);

> UPDATE a SET a1 = 3 WHERE a1 = 1;
  -- forza a nulla b(b1)!

> SELECT * FROM b;
 pk | b1 | b2
----+----+----
  2 |  2 |  2
  1 |    |  1

> DELETE FROM a WHERE a2=2;
  -- forza cancellazione anche di b
> SELECT * FROM b;
 pk | b1 | b2
----+----+----
  1 |    |  1
```

Vincoli posticipabili

Tutti i vincoli, con l'essclusione di `NOT NULL` e `CHECK` sono posticipabili: un vincolo posticipabile viene controllato non ad ogni statement ma alla fine della transazione.

```
> CREATE TABLE c( val int NOT NULL,
                  UNIQUE(val) DEFERRABLE );

> BEGIN;
> SET CONSTRAINTS ALL DEFERRED;
> INSERT INTO c VALUES (1), (1);
  -- le due tuple sembrano essere passate!
> COMMIT;
ERROR:  duplicate key value violates unique constraint "c_val_key"
DETAIL:  Key (val)=(1) already exists.
```

RENAME

Il comando `ALTER TABLE RENAME` consente di rinominare una tabella.
Il comando mantiene gli eventuali vincoli di integrità referenziale:

```
> CREATE TABLE a( pk serial PRIMARY KEY);
> CREATE TABLE b( pk serial PRIMARY KEY,
                  a  int REFERENCES a(pk) );

> ALTER TABLE a RENAME TO z;
> \d b
                            Table "public.b"
...
Foreign-key constraints:
    "b_a_fkey" FOREIGN KEY (a) REFERENCES z(pk)
```

Confronto fra tuple

PostgreSQL supporta un vincolo di tipo `EXCLUDE` da definire in fase di creazione della tabella o successivamente.
Il vincolo controlla le tuple sulle colonne specificate con l'operatore specificato: se si ha match l'inserimento non è consentito.

Esempio di EXCLUDE

```
> CREATE TABLE v (
    pk serial PRIMARY KEY,
    name text, value int );

> ALTER TABLE v
   ADD CONSTRAINT abs_value_exclude
   EXCLUDE ( abs( value ) WITH = );
```

ovvero si escludono tuple che hanno un valore assoluto (abs) della colonna value identico (operatore =):

```
> INSERT INTO v( name, value ) VALUES ( 'v1', 100 );
> INSERT INTO v( name, value ) VALUES ( 'v2', -100 );
ERROR:  conflicting key value violates exclusion constraint "abs_value_exclude"
DETAIL:  Key (abs(value))=(100) conflicts with existing key (abs(value))=(100)
```

EXCLUDE: effetti collaterali

Il vincolo EXCLUDE utilizza un indice (e quindi l'operatore deve essere applicatibile ad un indice) per effettuare il controllo:

```
> \d v
...
Indexes:
    "v_pkey" PRIMARY KEY, btree (pk)
    "abs_value_exclude" EXCLUDE USING btree (abs(value) WITH =)
```

In un certo senso, EXCLUDE è la negazione di un vincolo UNIQUE.

Commenti

L'istruzione speciale `COMMENT` permette di agganciare dei commenti agli oggetti definiti nel database. La sintassi è `COMMENT ON <type> IS 'BLAH'`.

```
> COMMENT ON TABLE persona IS 'Tabella anagrafica';
> COMMENT ON COLUMN persona.pk IS 'Chiave primari surrogata';
```

Vedere i commenti

Non esiste un modo "comodo" per vedere i commenti, i tool di amminitrazione li mostrano, ma in generale occorre andare sulla tabella `pg_description`:

```
> SELECT description
     FROM pg_description
     WHERE objoid = 'public.persona'::regclass;
       description
-------------------------
 Tabella anagrafica
 Chiave primari surrogata
```

In `psql` usare:

- `\d+` sul nome della tabella per vedere anche la descrizione delle colonne;
- `\dt+` sul nome della tabella per vedere anche la descrizione della tabella.

Valori di uscita di una INSERT

In default una istruzione di `INSERT` restituisce due valori:

- `oid` della tupla inserita (se la tabella ha gli oid);
- *numero di tuple inserite*.

E' possibile specificare una clausola `RETURNING` che restituisca dei valori basati su una *select-like* di ogni tupla inserita (caso banale: la chiave automatica).

INSERT RETURNING: un primo esempio

```
> INSERT INTO persona( cognome, nome, codice_fiscale )
  VALUES( 'Luca', 'Ferrari', 'FRRLCU78L19F257T' )
  RETURNING pk;
 pk
----
 13
```

INSERT RETURNING: un esempio piu' complesso

```
> INSERT INTO persona( cognome, nome, codice_fiscale )
  VALUES( 'Luca', 'Ferrari', 'FRRLCU78L19F257Z' )
  RETURNING upper(codice_fiscale), pk;
      upper        | pk
-------------------+----
 FRRLCU78L19F257Z | 14
```

INSERT or UPDATE?

Una *UPSERT* è una `INSERT` che, in caso di conflitto (vincolo di univocità violato) esegue una `UPDATE` automaticamente.
UPSERT è una modifica della sintassi di `INSERT`! UPSERT non può funzionare se si sono delle rules definite sulla tabella!
Si specifica cosa fare in caso di conflitto, ed eventualmente come risolvere tale conflitto.
In default viene ritornato il numero di tuple inserite o aggiornate in caso di conflitto (come da return di un `INSERT`).

Funzionamento di UPSERT

Occorre che sia specificato un `CONSTRAINT` che indica il conflitto (o una colonna su cui esiste un constraint di univocità).
La tupla in conflitto viene nominata `EXCLUDED`.

Senza UPSERT...

```
> INSERT INTO persona( codice_fiscale, nome, cognome )
  VALUES( 'FRRLCU78L19F257B', 'Ferrari', 'Luca' );

-- no upsert!
> INSERT INTO persona( codice_fiscale, nome, cognome )
  VALUES( 'FRRLCU78L19F257B', 'Luca', 'Ferrari' );
ERROR:  duplicate key value violates
  unique constraint "persona_codice_fiscale_key"
```

UPSERT in azione

```
-- upsert!
> INSERT INTO persona( codice_fiscale, nome, cognome )
  VALUES( 'FRRLCU78L19F257B', 'Luca', 'Ferrari' )
  ON CONFLICT(codice_fiscale)
  DO UPDATE SET nome = EXCLUDED.nome,
                cognome = EXCLUDED.cognome;
```

UPSERT in aborto (controllato)

Se si specifica la risoluzione del conflitto come `DO NOTHING` allora la query non effettua l'inserimento della tupla e fallisce silenziosamente (cioè con successo).

```
> INSERT INTO persona( codice_fiscale, nome, cognome )
  VALUES( 'FRRLCU78L19F257B', 'Luca', 'Ferrari' )
  ON CONFLICT(codice_fiscale)
  DO NOTHING;

INSERT 0 0
```

UPSERT RETURNING

E' possibile usare la clausola `RETURNING` anche nel caso di un *UPSERT*:

```
> INSERT INTO persona( codice_fiscale, nome, cognome )
  VALUES( 'FRRLCU78L19F257B', 'Luca', 'Ferrari' )
  ON CONFLICT(codice_fiscale)
  DO UPDATE SET nome = EXCLUDED.nome,
                cognome = EXCLUDED.cognome
RETURNING nome, cognome;

 nome | cognome
------+---------
 Luca | Ferrari
```

Cos'è il search path?

Ogni utente definisce un *percorso* (simile al `PATH` di una shell) che rappresenta dove trovare gli oggetti del database (tabelle, funzioni, ecc.). In default il `search_path` contiene uno schema che si chiama come l'utente, identificato dalla stringa speciale `"$user"` e `public`. Se l'utente modifica il `search_path` ogni oggetto non qualificato non sarà piu' visibile a meno che non si trovi in un path specificato.

```
>  SHOW search_path;
   search_path
-----------------
 "$user", public

>  SELECT count(*) FROM persona;
  -- ok !
> SELECT count(*) FROM public.persona;
  -- ditto
```

A cosa server "$user"?

L'idea è quella di permettere ad ogni utente di avere oggetti omonimi ma con valori differenti.
Ad esempio tabelle e funzioni di configurazione, che piazzati opportunamente nello schema del nome utente diventano "visibili" a lui senza andare a interferire con quelli degli altri utenti.

Modificare il search path

Il comando SET permette di impostare il valore del search_path:

```
> SET search_path="$user", private;
> SHOW search_path;
   search_path
------------------
 "$user", private


> SELECT count(*) FROM persona;
ERROR:  relation "persona" does not exist
LINE 1: SELECT count(*) FROM persona;
                             ^
> SELECT count(*) FROM public.persona;
  -- ok !
```

Considerazioni di sicurezza sul search_path e public

Sostanzialmente si applicano tutte le considerazioni di sicurezza relative al PATH di Unix.
Inoltre PostgreSQL consente la creazione di nuovi oggetti a ogni utente (in default) che abbia accesso al database. Questo significa che è possibuile "mascherare" oggetti e funzioni maliziosamente nello schema public. A tal fine è opportuno rimuovere il permesso di creazione sullo schema public:

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

Modalità di cast

In PostgreSQL il cast può essere effettuato in quattro modi:

- specificando il tipo prima del valore (es. `int '10'`);
- specificando il tipo dopo il valore con l'operatore `::` (es. `'10'::int`);
- usando l'operatore `CAST`;
- usando il tipo come fosse una funzione (es. `text( 10 )`).

```
> SELECT '10'::int + 3;
> SELECT int '10' + 3;
> SELECT CAST( '10' AS integer ) + 3;
```

Considerazioni sul cast

I modi consigliati sono `CAST()` (conforme SQL standard) o `::`, gli altri hanno delle limitazioni (es. la modalità funzione va usata solo per nomi di tipi che sono anche nomi validi di funzione, mentre il tipo prima del valore funziona solo per i letterali).
Qualora il cast non sia ambiguo, è possibile ometterlo, ovvero:

```
> SELECT '10' + 3;
```

pg_typeof

La funzione pg_typeof può essere usata per vedere come il sistema cerca di interpretare i tipi di dato:

```
> SELECT pg_typeof( 1234 ) AS first,
         pg_typeof( 1234.56 ) AS second,
         pg_typeof( 'Hello' || 'World' ) AS third;

  first  | second  | third
---------+---------+-------
 integer | numeric | text
```

CASE

Rappresento lo *switch* del linguaggio C, o il *given* di Perl:

```
> SELECT nome,
  CASE
     WHEN eta >= 18 THEN 'maggiorenne'
     ELSE 'minorenne'
  END
  FROM persona;
 nome  |     case
-------+-------------
 Luca  | maggiorenne
 Diego | minorenne
```

CASE con equals

Si può usare la versione che effettua una comparazione secca sul valore da analizzare:

```
> SELECT nome,
  CASE eta
   WHEN 39 THEN 'maggiorenne'
   WHEN 10 THEN 'minorenne'
   ELSE 'non saprei'
  END
  FROM persona;
```

COALESCE

Ritorna il primo valore non nullo degli argomenti specificati:

```
> SELECT COALESCE( NULL, 'MIRIAM',
                   'LUCA', NULL );
 coalesce
----------
 MIRIAM
```

E' utile per estrarre informazioni che possono trovarsi in campi multipli.

NULLIF

Ritorna un valore NULL se gli operatori sono uguali, altrimenti ritorna il primo valore:

```
> SELECT NULLIF('luca', 'LUCA');
 nullif
--------
 luca
> SELECT NULLIF( 10, 10 );
 nullif
--------
```

E' una sorta di xor del poveraccio!

Row

L'operatore `ROW` permette di costruire una *tupla* al volo. Accetta l'uso di costanti e di espressioni da valutare al volo:

```
> SELECT ROW( 1, 2, 'Foo', 'Fighters' );
        row
--------------------
 (1,2,Foo,Fighters)

> SELECT ROW( p.codice_fiscale, length( p.codice_fiscale ) ) FROM persona p;
          row
-----------------------
 (FRRLCU71L19F257B,16)
```

Costruttore di un array

Gli array sono dichiarati con il tipo seguito da parentesi quadre (es. `integer[]`). Il costruttore dell'array è l'operatore `ARRAY`, che cerca di comprendere il tipo dagli elementi dell'array stesso (se non effettuato un cast).

```
> SELECT ARRAY[1, 2, 3, 4]::integer[];
   array
-----------
 {1,2,3,4}
```

Array multidimensionali

Sono costruiti combinando assieme piu' array.

```
> SELECT ARRAY[ ARRAY[ 2, 4 ], ARRAY[ 1, 3 ] ];
     array
---------------
 {{2,4},{1,3}}
```

Array in tabella

Le tabelle possono includere degli array.

```
> CREATE TABLE software( name text, versions text[] );
> INSERT INTO software
  VALUES( 'Java', '{1.5, 1.6, 1.7, 1.8 }' );
> INSERT INTO software
  VALUES( 'Perl', '{5.10, 5.20, 6.c}' );
```

Array in tabella: indici

Gli indici degli array funzionano come nel linguaggio C, ma gli indici partono da 1!:

```
> SELECT name, versions[1] FROM software;
 name | versions
------+----------
 Perl | 5.10
 Java | 1.5
```

Array in tabella: estremi

Le funzioni `array_lower()` e `array_upper()` forniscono il minimo e massimo indice usabile in una determinata dimensione dell'array:

```
> SELECT name,
  versions[ array_lower( versions, 1) ] AS required,
  versions[ array_upper( versions, 1 ) ] AS optimal
  FROM software;
 name | required | optimal
------+----------+---------
 Perl | 5.10     | 6.c
 Java | 1.5      | 1.8
```

Array in tabella: slicing

L'operatore `[:]` permette di ottenere lo slicing:

```
> SELECT name, versions[1:2] FROM software;
 name |  versions
------+------------
 Perl | {5.10,5.20}
 Java | {1.5,1.6}
```

Array in tabella: aggiunta di elementi

Si possono seguire diverse strade:

- ricostruire l'intero array con i nuovi valori tramite `ARRAY`;
- appendere l'array originale con uno singolo (costruito con `ARRAY` o `'{}'`);
- usare `array_prepend()` per inserire un elemento in testa, `array_append()` per aggiungerlo in coda e/o `array_cat()` per concatenare due array.

Array in tabella: aggiunta di elementi (esempio 1)

```
> UPDATE software
  SET versions = versions || ARRAY['2017.09']
  WHERE name = 'Perl';

> UPDATE software
  SET versions = versions || '{1.8_168}'
  WHERE name = 'Java';
```

Array in tabella: aggiunta di elementi (esempio 2)

```
> UPDATE software
  SET versions = array_append( versions, '1.8_169' )
  WHERE name = 'Java';

> UPDATE software
  SET versions = array_prepend( '1.4', versions )
  WHERE name = 'Java';
```

Array in tabella: concatenazione

```
> UPDATE software
  SET versions = array_cat( ARRAY[ '5.6', '5.8' ], versions )
  WHERE name = 'Perl';
```

Array in tabella: ricerca

Ci sono due operatori principali:

- `ANY` ricerca un valore in uno qualunque degli elementi dell'array;
- `ALL` ricerca un valore in tutti gli elementi dell'array.

Altri operatori utili nel confronto di due array:

- `&&` ricerca le sovrapposizioni;
- `@>` *contains*;
- `<@` *is contained by*.

Array in tabella: ricerca (esempio 1)

```
> SELECT name FROM software WHERE '6.c' = ANY( versions );
 name
------
 Perl

> SELECT name FROM software WHERE '6.c' = ALL( versions );
 name
------
(0 rows)
```

Array in tabella: ricerca (esempio 2)

```
> SELECT name FROM software WHERE ARRAY[ '5.10', '5.20' ] && versions;
 name
------
 Perl
```

Array in tabella: ricerca (esempio 3)

```
-- contains
> SELECT name FROM software
  WHERE ARRAY[ '5.10', '5.20' ] @> versions;

(0 rows)

-- is contained by
> SELECT name FROM software
  WHERE ARRAY[ '5.10', '5.20' ] <@ versions;
 name
------
 Perl
```

Array in tabella: eliminare un elemento

La funzione `array_remove()` toglie un valore da un array, mentre `array_replace()` sostituisce un elemento con un altro. Restituiscono l'array modificato!

```
> UPDATE software
  SET versions = array_remove( versions, '5.6' );

> UPDATE software
  SET versions = array_replace( versions, '5.8', '5.8.2' );
```

Array in tabella: trovare un elemento

Le funzioni `array_position()` e `array_positions()` ritornano la posizione di uno elemento (eventualmente ripetuto) nell'array:

```
> SELECT name, array_positions( versions, '5.20' )
  FROM software;
 name | array_positions
------+-----------------
 Perl | {3}
 Java | {}
```

Array in tabella: trasformare un array in tabella

```
> SELECT name, unnest( versions ) FROM software;
 name | unnest
------+---------
 Perl | 5.8.2
 Perl | 5.10
 Perl | 5.20
 Perl | 6.c
...
```

string_to_array

La funzione string_to_array spezza una stringa sulla base di un separatore e restituisce le sottostringhe come array:

```
> SELECT
   string_to_array( 'Luca ferrari', ' ' ) AS array,
   v[1] AS nome
  FROM
   string_to_array( 'Luca Ferrari', ' ' ) v;

array | {Luca,ferrari}
nome  | Luca
```

Range di dato

I tipi di dato *range* sono valori *non esattamente definiti*. L'idea è quella di identificare un tipo ammesso di valori (denominato subtype), sul quale si imposta un valore di inizio e di fine e tutti i valori fra questi inclusi. Esempio: *dalle ore 8:00 alle ore 9:00*. Con un solo valore range si indica l'inizio (*8:00*) e la fine (*9:00*) del sottotipo (es. `time`).

Range predefiniti

PostgreSQL supporta i seguenti tipi range:

- `int4range` range di `integer`;
- `int8range` range di `biginteger`;
- `numrange` range di `numeric`;
- `tsrange` rande di timestamp;
- `daterange` range di `date`.

Esempio di utilizzo di range

```
> CREATE TABLE ticket(
    pk SERIAL PRIMARY KEY,
    tipo text,
    periodo daterange,
    altezza int4range );
```

Costruzione di un range: sintassi stringa

Un tipo range viene sempre specificato come stringa (fra apici) e può valere:

- `[ begin, end ]` oppure `( begin, end )`
- `empty` per non specificare nessun valore (simile a `NULL`),

Come la forma matematica, le parentesi quadre indicano l'inclusione dell'estremo mentre quelle tonde l'esclusione dell'estreno.

Costruzione di range: costruttori

Alternativamente alla forma stringa, ogni tipo di range predefinito include un costruttore con lo stesso nome del tipo di range e che accetta due o tre parametri:

- `typerange( a, b )` costruisce un range di `type` come `'[a, b)'`;
- `typerange( a, b, '()' )` costruisce un range di `type` con gli estremi specificati dalla combinazione delle parentesi.

Inserimento di valori di range

```
> INSERT INTO ticket( tipo, periodo, altezza )
  VALUES( 'GRATUITO-BIMBO',
  '[2017-06-01, 2017-09-30)',
  '(60, 100)' );

> INSERT INTO ticket( tipo, periodo, altezza )
  VALUES( 'GRATUITO-ANZIANO',
  '[2017-06-01, 2017-10-31]',
  'empty' );
```

Operatori di range

Per ricercare fra un range si usano operatori simili a quelli di un array:

- `@>` il range (a sinistra) contiene il valore scalare a destra;
- `isempty` indica se il range è vuoto;
- `upper` e `lower` estraggono gli estremi del range;
- `&&` sovrapposizione fra due range.

Query sui range

```
> SELECT tipo FROM ticket
  WHERE periodo
     && daterange( '2017-07-01',
                   '2017-07-31',
                   '[]' );
```

Query sui range (2)

```
> SELECT tipo FROM ticket
  WHERE periodo @> '2017-10-31'::date;
```

Query sui range (3)

Testare se un range è vuoto:

```
> SELECT
    isempty(
     daterange( CURRENT_DATE, CURRENT_DATE, '()' ) );

isempty | t
```

Query sui range (4)

I range permettono anche di riscivere in modo *migliore* alcune query di tipo *compreso fra*:

```
> SELECT *
  FROM persona
  WHERE ts >= '2018-02-17 12:22:39.858051'
    AND ts <= '2018-02-17 12:22:39.858053';
```

diventa

```
> SELECT *
  FROM persona
  WHERE tsrange( '2018-02-17 12:22:39.858051',
                 '2018-02-17 12:22:39.858053',
                 '[]' ) @> ts;
```

Attenzione a come si presentano i range

```
> SELECT
   daterange( current_date, current_date, '[]' ) as SAME_INCLUSIVE,
   daterange( CURRENT_DATE, CURRENT_DATE , '()' ) as SAME_EXCLUSIVE,
   daterange( CURRENT_DATE, CURRENT_DATE + 1, '(]' ) as TOMORROW_INCLUSIVE;

same_inclusive     | [2018-02-17,2018-02-18) -- oggi fino a domani escluso = oggi!
same_exclusive     | empty                   -- non oggi = nulla!
tomorrow_inclusive | [2018-02-18,2018-02-19) -- non oggi fino a domani incluso = do
```

Tipi personalizzati

PostgreSQL consente di creare dei *tipi* personalizzati:

- compositi (una sorta di struttura);
- enum le classiche enumerazioni, sostanzialmente una serie di *etichette*;
- range un tipo che identifica un range di valori;
- scalare un tipo fortemente integrato nel server e che richiede la scrittura di opportune funzioni in codice C;

Creare un tipo enumerazione

Si vogliono *standardizzare* gli stati di un software:

```
> CREATE TYPE sw_version
  AS ENUM ( 'stable',
            'unstable',
            'EOL',
            'development' );
```

Creare un tipo composito

SI supponga di voler creare un semplice tipo strutturato per un repository software:

```
> CREATE TYPE sw_repository
  AS ( url text,
       author text );
```

Il tipo composito si usa con parentesi tonde!

Usare i tipi in una tabella

```
> CREATE TABLE software(
      pk SERIAL PRIMARY KEY,
      name text,
      version sw_version,
      repository sw_repository );
```

Inserire i tipi compositi

```
> INSERT INTO software( name, version, repository )
  VALUES( 'PostgreSQL-9.6',
      'stable',
      ( 'https://www.postgresql.org', 'PGDG' ) );

> INSERT INTO software( name, version, repository )
  VALUES( 'Perl-6',
     'stable',
     ( 'https://www.perl6.org', 'Perl Developers' ) );
```

Estrarre i tipi composti

```
> SELECT name, (repository).url FROM software;
      name       |             url
-----------------+-----------------------------
 PostgreSQL-9.6 | https://www.postgresql.org
 Perl-6         | https://www.perl6.org
```

Creare un tipo personalizzato range

```
> CREATE OR REPLACE FUNCTION f_versions_diff( older float, newer float )
RETURNS float AS $BODY$
DECLARE
BEGIN
    RETURN (newer - older)::integer;
END;
$BODY$
LANGUAGE plpgsql IMMUTABLE;
```

Creare un tipo personalizzato range (2)

```
> CREATE TYPE sw_major_version AS RANGE (
subtype = float,
subtype_diff = f_versions_diff );
```

Creare un tipo personallizato range (3)

```
> SELECT '[9.6, 10.0)'::sw_major_version;
 sw_major_version
------------------
 [9.6,10)
```

Tipi di dato JSON

PostgreSQL supporta due tipi di dato /JSON (JavaScript Object Notation):

- `json` tipo di dato testuale che richiede un nuovo *parsing* ogni volta che si opera sul dato stesso;
- `jsonb` una forma che viene destrutturata e memorizzata in formato binario per successive elaborazioni piu' rapide (richiede maggior tempo in inserimento per la conversione).

Il formato `json` mantiene l'input intatto, quindi spazi bianchi, chiavi duplicate (solo l'ultima viene trattata dagli operatori). IL tipo `jsonb` rimuove spazi bianchi ridondanti nonché mantiene un solo valore per ogni chiave (l'ultimo nel caso di chiavi duplicate).

Sintassi JSON

La sintassi per creare un tipo JSON prevede:

- *scalari* (di tipo intero o stringa);

```
> SELECT '"hello"'::json;
> SELECT '10'::json;
```

- *array* (anche di tipi differenti), stabiliti da parentesi quadre;

```
> SELECT '[ "Luca", "Ferrari", 39 ]'::json;
```

- *oggetti* identificati da parentesi graffe;

```
> SELECT '{ "name" : "Luca", "surname" : "Ferrari", "age" : 39 }'::json;
```

Un esempio pratico di differenza fra JSON e JSONB

Si ricordi che `json` mantiene intalterati i dati di input, mentre `jsonb` li riorganizza e li "ottimizza" tenendo solo l'ultima chiave nel caso di duplicazione:

```
> SELECT '{ "name" : "Luca",
  "surname" : "Ferrari",
  "age" : 29,
  "age" : 39 }'::json;
          json
-----------------------
 { "name" : "Luca",    +
 "surname" : "Ferrari",+
 "age" : 29,           +
 "age" : 39 }
> SELECT '{ "name" : "Luca",
    "surname" : "Ferrari",
    "age" : 29,
    "age" : 39 }'::jsonb;
                      jsonb
-----------------------------------------------
 {"age": 39, "name": "Luca", "surname": "Ferrari"}
```

* Operatori JSON Gli operatori hanno tutti due varianti:
  1. se usati con indice numerico forniscono accesso ad un array;

Operatori JSON: esempi

```
> SELECT '[ "foo", 0, "bar", 1 ]'::json->1; -- 0

> SELECT '{ "foo" : 0, "bar" : 1 }'::json->'foo'; -- 0

> SELECT '{ "foo" : { "bar" : 1 } }'::json#>'{foo, bar}'; --1
```

Operatori JSONB: esempi

Il tipo `jsonb` dispone di altri operatori comodi:

- `?` ricerca una chiave nell'oggetto;

```
> SELECT '{ "foo" : { "bar" : 1 } }'::jsonb ? 'bar'; -- true
```

- `@>` e `<@` contenimento da einistra a destra e viceversa (solo al livello principale);

```
> SELECT '{ "foo" : { "bar" : 1 } }'::jsonb
    @> '{"bar" : 1}'::jsonb;  -- false

> SELECT '{ "foo" : { "bar" : 1 } }'::jsonb
   @> '{"foo" : {"bar" : 1}}'::jsonb;  -- true
```

- `||` concatenazione di due oggetti.

```
> SELECT '{ "name" : "Luca" }'::jsonb
  || '{ "surname" : "Ferrari" }'::jsonb;
```

Esempio di uso di JSONB in tabella

```
> CREATE TABLE persona (
    pk SERIAL PRIMARY KEY,
    name text,
    surname text,
    stuff jsonb );

> INSERT INTO persona( name, surname, stuff )
  VALUES ( 'Luca', 'Ferrari',
    '{ "email" : "luca@mail.me", "web" : "http://fluca1978" }' );

> INSERT INTO persona( name, surname, stuff )
  VALUES ( 'Emanuela', 'Santunione',
    '{ "email" : "luca@mail.me" }' );
```

Esempio di JSONB in tabella (2)

Quali persone hanno un sito web?

```
SELECT name, surname, stuff->'web'
FROM persona
WHERE stuff ? 'web';

 name | surname |      ?column?
------+---------+--------------------
 Luca | Ferrari | "http://fluca1978"
```

Funzioni di utilitaì JSON

Ci sono molte funzioni di utilità per convertire dati da e per JSON:

- `row_to_json` converte una tupla in un oggetto JSON (esportabile);
- `jsopn_build_object` costruisce un oggetto JSON da una lista di valori;
- `array_to_json` converte un array PostgreSQL in uno JSON.

```
> SELECT row_to_json( row( name, surname, stuff->'email' ) )
  FROM persona
  WHERE stuff ? 'email';
                        row_to_json
-----------------------------------------------------------
 {"f1":"Luca","f2":"Ferrari","f3":"luca@mail.me"}
 {"f1":"Emanuela","f2":"Santunione","f3":"luca@mail.me"}

> SELECT json_build_object( 'name', 'Luca', 'surname', 'Ferrari' );
             json_build_object
---------------------------------------
 {"name" : "Luca", "surname" : "Ferrari"}

> SELECT array_to_json( ARRAY[ 1, 2, 3, 4 ] );
 array_to_json
---------------
 [1,2,3,4]
```

Esempio di JSONB in linea

```
> SELECT '{ "nome": "Luca", "cognome" :
      "Ferrari", "cognome" : "FERRARI",
"db": [ "postgres", "sqlite3" ] }'::jsonb->'db'->>1;
-[ RECORD 1 ]--------------------
?column? | postgres
```

hstore

L'estensione `hstore` consente di memorizzare dati nella forma *chiave, valore* ove il valore può essere anche `NULL`.
La chiave e il valore sono salvati come *testo* in un'unica colonna, e diversi operatori esistono per dereferenziare ogni singola parte.
Piu' valori possono essere memorizzati assieme, separati da una virgola.
Supporta indici btree (uguaglianza), gin e gist (sottoinsiemi ed inclusione).

```
# CREATE EXTENSION hstore;
```

hstore: operatori

- -> dereferenzia una chiave o un array di chiavi;
- ? indica se una determinata chiave esiste;
- ?& indica se un array di chiavi esistono;
- || concatenazione di due hstore;
- @> e <@ contenimento sinistra verso destra e viceversa;
- - rimuove una chiave o un insieme di chiavi
- %% e %# converte in un array monodimensionale o multidimensionale.

hstore: esempi

```
> SELECT
   'name => Luca'::hstore,
   'name => Luca, surname => Ferrari'::hstore;

hstore | "name"=>"Luca"
hstore | "name"=>"Luca", "surname"=>"Ferrari"

> SELECT 'name => Luca, surname => Ferrari'::hstore
 ? 'surname';

surname_exists | t

> SELECT
   'name => Luca, surname => Ferrari'::hstore
        ->'surname' AS surname;

surname | Ferrari
```

hstore: esempi (2)

```
> SELECT 'name => Luca, surname => Ferrari'::hstore
       ->ARRAY[ 'name', 'surname' ] AS name_surname;

name_surname | {Luca,Ferrari}

> SELECT 'name => Luca, surname => Ferrari'::hstore
  - ARRAY[ 'name', 'age' ] AS reduced;
-[ RECORD 1 ]----------------
reduced | "surname"=>"Ferrari"
```

hstore: esempi (3)

```
> SELECT
      %% 'name => Luca, surname => Ferrari'::hstore
      AS mono_array;

mono_array | {name,Luca,surname,Ferrari}

> SELECT
    %# 'name => Luca, surname => Ferrari'::hstore
    AS multi_array;

multi_array | {{name,Luca},{surname,Ferrari}}
```

Manipolazione di hstore

```
> SELECT hstore( rec )
  FROM software rec;

 "name"=>"Perl", "versions"=>"{5.8.2,5.10,5.20,6.c,2017.09}"
 "name"=>"Java", "versions"=>"{5.8.2,1.4,1.5,1.6,1.7,1.8,1.8_168,1.8_169}"
 "name"=>"Perl", "versions"=>"{5.8.2,5.10,5.20,6.c,2017.09}"

> SELECT hstore_to_jsonb(
        'name => Luca, surname => Ferrari'::hstore );

 {"name": "Luca", "surname": "Ferrari"}

> SELECT each(
         'name => Luca, surname => Ferrari'::hstore );

 (name,Luca)
 (surname,Ferrari)
```

Introduzione

I *tablespaces* sono un meccanismo per "agganciare" al server altri percorsi fisici di memorizzazione. Gli oggetti del database possono poi essere creati/spostati su uno specifico tablespace.
I tablespaces possono sovrascrivere i parametri di costo dell'ottimizzatore (ad esempio perché su dispositivi piu' veloci).

Creazione di un tablespace

Anzitutto occorre creare la directory fisica (mount-point o altro) e concedere i privilegi all'utente che esegue il cluster.

```
% sudo mkdir /mnt/data2/my_tablespace
% sudo chown postgres:postgres /mnt/data2/my_tablespace
% sudo chmod 700 /mnt/data2/my_tablespace
```

A questo punto sul server è possibile creare il table space (occorre essere superutenti):

```
# CREATE TABLESPACE my_tablespace
     OWNER luca  -- opzionale
     LOCATION '/mnt/data2/my_tablespace'
     WITH ( seq_page_cost = 2 ); --opzionale
```

Usare un tablespace

E' possibile ora creare gli oggetti sul nuovo tablespace:

```
> CREATE TABLE baz( value int )
  TABLESPACE my_tablespace;

> \d baz
      Table "public.baz"
 Column |  Type   | Modifiers
--------+---------+-----------
 value  | integer |
Tablespace: "my_tablespace"
```

Spostare oggetti verso un tablespace

E' possibile spostare gli oggetti da un tablespace ad un altro:

```
> ALTER TABLE persona
  SET TABLESPACE my_tablespace;
```

Una volta che gli oggetti sono raggruppati in un tablespace diventa facile spostarli in blocco:

```
> ALTER TABLE ALL IN TABLESPACE my_tablespace
  SET TABLESPACE other_tablespace;
```

Le stesse query valgono per `ALTER INDEX`.

Informazioni sui tablespace

La vista speciale pg_tablespace fornisce informazioni sui tablespace attivi:

```
> SELECT * FROM pg_tablespace;
-[ RECORD 1 ]-----------------
spcname    | pg_default
spcowner   | 10
spcacl     |
spcoptions |
-[ RECORD 2 ]-----------------
spcname    | pg_global
spcowner   | 10
spcacl     |
spcoptions |
-[ RECORD 3 ]-----------------
spcname    | my_tablespace
spcowner   | 16387
spcacl     |
spcoptions | {seq_page_cost=2}
```

Tablespace: cosa avviene fisicamente

I tablespaces sono un meccanismo per *scappare* dalla directory `$PGDATA` e vengono infatti linkati all'interno di essa tramite `pg_tblspc`:

```
% sudo ls -l /mnt/data1/pgdata/pg_tblspc
lrwx------  1 postgres  postgres  24 Nov 13 17:43 19749 -> /mnt/data2/my_tablespace
lrwx------  1 postgres  postgres  26 Nov 13 17:50 19756 -> /mnt/data2/my_tablespace
```

Occorre tenere presente questo aspetto quando si decide di spostare in blocco `$PGDATA` o qualora un mount point sia malfunzionante.

Importare dati con COPY

Il comando COPY permette di importare dati da un file in una tabella.

```
# COPY persona( nome, cognome, codice_fiscale )
   FROM '/home/luca/anagrafica.csv'
   WITH
   ( DELIMITER ';',
     FORMAT 'csv',
     HEADER 'on') ;
```

Se si copia da un file occorre essere superutenti, per copiare da standard input non serve.

Il file deve essere locale al server! Il percorso del file è relativo a PGDATA o va specificato come assoluto.

Esportare dati con COPY

Il comando COPY consente anche di esportare i dati verso un file:

```
# COPY evento TO '/tmp/evento.csv'
   WITH ( FORMAT 'csv',
          HEADER true,
          QUOTE '"',
          DELIMITER ';' );
```

COPY da e verso un programma

Il comando COPY permette di specificare anche un programma come sorgente dati o output:

```
# COPY persona( nome, cognome, codice_fiscale )
   FROM PROGRAM '/bin/cat /home/luca/anagrafica.csv'
  WITH
   ( DELIMITER ';',
     FORMAT 'csv',
     HEADER 'on') ;
```

Il comando \copy

Il comando \copy del client psql effettua un COPY ma dal client verso il server, quindi trasferendo i dati tramite la connessione al server. La sintassi è identica a quella del comando COPY, per esempio:

```
> \copy persona(nome, cognome, codice_fiscale)
   from 'anagrafica.csv'
   with ( header true,
          format 'csv',
          delimiter ';' )
```

Informazioni sulla transazione corrente

In generale tutti i valori `current_*` forniscono informazioni sullo stato attuale, a questo si possono aggiungere alcune funzioni di utilità quali:

- `txid_current()` restituisce il transaction identifier della transazione corrente;
- `pg_backend_pid()` fornisce il process ID (del sistema operativo) della connessione corrente (o meglio del processo backend che sta servendo la connessione corrente);
- `current_role` e `current_user` cono equivalenti!

```
> SELECT current_user, current_date,
         current_timestamp,
         txid_current(),
         pg_backend_pid();
-[ RECORD 1 ]--+------------------------------
current_user   | luca
date           | 2017-11-20
now            | 2017-11-20 13:29:56.139872+01
txid_current   | 3063
pg_backend_pid | 775
```

Informazioni di connessione

Ci sono dei valori speciali:

- `current_schema()` e `current_database()` indicano lo schema corrente e il database corrente;
- `inte_client_addr()`, `inet_client_port()` e le duali `inet_server_addr()` e `inet_server_port()` forniscono indicazioni TCP/IP.

```
# SELECT current_user
    || '@' || inet_server_addr()
    || ':' || inet_server_port() AS myself,
    inet_client_addr() || ':'
    || inet_client_port() AS connected_from,
    current_database(),
    current_schema();

myself           | postgres@127.0.0.1/32:5432
connected_from   | 127.0.0.1/32:12951
current_database | testdb
current_schema   | public
```

Uptime del servizio

La funzione `pg_postmaster_start_time()` indica quando è stato avviato il cluster, e quindi può essere usata per ottenere una sorta di *uptime* del servizio:

```
# SELECT date_trunc( 'second',
   now() - pg_postmaster_start_time() );

date_trunc | 01:43:48
```

Informazioni di dimensione di una tabella

Le dimensioni di un oggetto sono ottenute tramite una serie di funzioni `pg_*_size()`:

- `pg_column_size()` fornisce la dimensione in bytes di una colonna;
- `pg_relation_size()` fornisce la dimensione di una relazione rispetto al suo *fork* (`vm`, `fsm`, `main`);
- `pg_table_size()` dimensione della tabella e dello spazio TOAST (se usato);
- `pg_total_relation_size()` fornisce la dimensione della tabella, del TOAST e degli indici.

Le funzioni `pg_size_bytes()` e `pg_size_pretty()` forniscono una versione "umanamente leggibile" delle dimensioni, e spesso sono usate come wrapper per funzioni di dimensione.

Informazioni di dimensione di una tabella: esempio

```
> SELECT pg_relation_size( 'evento', 'main' ) AS main,
        pg_relation_size( 'evento', 'fsm' ) AS free_space_map,
        pg_relation_size( 'evento', 'vm' ) AS visibility_map,
        pg_table_size( 'evento' ),
        pg_total_relation_size( 'evento' ),
        pg_size_pretty( pg_total_relation_size( 'evento' ) );

main                  | 110059520
free_space_map        | 0
visibility_map        | 0
pg_table_size         | 110067712
pg_total_relation_size | 224051200
pg_size_pretty        | 214 MB
```

Informazioni di dimensione di una tabella: esempio 2

```
> SELECT pg_column_size( 'evento.description' ) AS description_size;
-[ RECORD 1 ]----+---
description_size | 19
```

Informazioni sulla dimensione del tablespace e del database

Se è presente un tablespace, la funzione (sovraccaricata) `pg_tablespace_size()` fornisce la sua dimensione su disco. La funzione `pg_database_size()` indica invece la dimensione di un database.

```
> SELECT pg_size_pretty( pg_tablespace_size( 'my_tablespace' ) ),
         pg_size_pretty( pg_database_size( 'testdb' ) );

pg_size_pretty | 512 bytes
pg_size_pretty | 371 MB
```

Directory e file

PostgreSQL mette a disposizione dell'utente amministratore una serie di funzioni per ottenere informazioni (in formato tabellare) su file e directory:

- `pg_ls_dir()` fornisce l'elenco dei file, il percorso deve essere relativo a `$PGDATA`;
- `pg_stat_file()` fornisce lo `stat(1)` del file;
- `pg_read_file()` e `pg_read_binary_file()` ritornano rispettivamente il contenuto testuale (`text`) o binario (`bytea`) del file.

Esempio di listato di directory

```
# WITH files AS ( SELECT pg_ls_dir( 'pg_log' ) )
  SELECT * FROM files
  WHERE pg_ls_dir like '%2018-01-__%.log';
             pg_ls_dir
---------------------------------
 postgresql-2018-01-11_104609.log
```

Esempio di lettura di un file

```
# SELECT regexp_split_to_table(
            pg_read_file( 'pg_log/postgresql-2018-01-11_104609.log' ),
                          E'\\n' );

LOG:  database system was shut down at 2018-01-11 10:46:09 CET
LOG:  database system is ready to accept connections
ERROR:  could not stat file "pg_log/postgresql-2017-11-13_093647.log": No such file
STATEMENT:  FETCH FORWARD 20 FROM _psql_cursor
ERROR:  current transaction is aborted, commands ignored until end of transaction bl
STATEMENT:  CLOSE _psql_cursor
```

Il comando ALTER TABLE

Il comando ALTER TABLE è uno dei piu' potenti e complessi comandi per la gestione di una tabella esistente. Consente di spostare la tabella da uno schema ad un altro, di aggiungere o rimuovere colonne, rinominarle, aggiungere vincoli, cambiare tipi di dato, ecc.
Il comando può essere eseguito in una transazione, in realtà molti comandi *DDL* possono essere eseguiti in una transazione (a parte in generale tutti i CREATE).
ALTER TABLE consente di eseguire piu' operazioni in un colpo solo (ad esempio aggiunta di una colonna e rimozione di un'altra).
Molti dei sottocomandi (azioni) di ALTER TABLE supportano IF NOT EXISTS e IF EXISTS.

Aggiungere una colonna

```
> BEGIN;
> ALTER TABLE persona
    ADD COLUMN data_nascita date
        DEFAULT '1950-01-01'::date;
> COMMIT;
```

ATTENZIONE: un comando come quello sopra richiede una riscrittura dell'intera tabella!

Rimuovere una colonna, rinominare una colonna

```
> ALTER TABLE persona DROP COLUMN data_nascita;

> ALTER TABLE persona RENAME eta TO eta_years;
```

ALTER TABLE con azioni multiple

Il comando supporta azioni multiple, se correlate (vedere la documentazione):

```
> ALTER TABLE persona
   ADD COLUMN IF NOT EXISTS
          description text NOT NULL DEFAULT 'N/A',
   ADD COLUMN IF NOT EXISTS
          data_nascita date NOT NULL DEFAULT '1950-01-01'::date,
   ALTER COLUMN eta_years SET NOT NULL;


> ALTER TABLE persona
    DROP COLUMN IF EXISTS data_nascita,
    DROP COLUMN IF EXISTS description,
    ADD COLUMN  IF NOT EXISTS titolo varchar(6);
```

ALTER TABLE USING

Il comando ALTER TABLE ammette una clausola USING che consente di specificare come applicare le modifiche, ad esempio il valore di una colonna. Questo comando funziona solo se si cambia un tipo di dato, al limite anche lasciandolo uguale al precedente:

```
> BEGIN;
> ALTER TABLE persona
   ADD COLUMN data_nascita date
    NOT NULL DEFAULT '1950-01-01'::date;

> ALTER TABLE persona
   ALTER COLUMN data_nascita
   SET DATA TYPE date
   USING date_trunc( 'year', current_date )::date - ( eta_years * 365 );

> COMMIT;
```

In questo caso prima si aggiunge la colonna con un valore di default, poi si istruisce la colonna a ricalcolarsi con un valore approssimato della data di nascita. Lo stesso risultato si sarebbe potuto ottenere con un UPDATE del valore della colonna.

ALTER TABLE USING (2)

La potenza di `ALTER TABLE USING` ovviamente si ottiene quando si cambia il tipo di dato:

```
-- supponendo di avere description come char
> ALTER TABLE persona ALTER COLUMN description
    SET DATA TYPE text
    USING upper(nome) || ' ' || upper(cognome);
```

Uso di filter su funzioni di aggregazione

```
> SELECT
  COUNT(*) FILTER ( WHERE nome LIKE 'F%' ) AS begin_with_f
 ,COUNT(*) FILTER ( WHERE nome LIKE 'L%' ) AS begin_with_l
 FROM persona;
```

Uso di filter in group by

```
> SELECT nome, count(*)
  FROM persona
  GROUP BY nome
  HAVING COUNT(*) FILTER( WHERE nome LIKE 'L%' ) > 1;

 nome | count
------+-------
 Luca |  5000
```

Reset della sessione

Il comando `DISCARD` consente di resettare una sessione, eliminando tabelle temporanee, advisory locks, piani di esecuzione in cache, variabili, ecc.
Supporta alcuni tipi di discard:

- `ALL` resetta completamente la sessione;
- `TEMPORARY` cancella ogni tabella temporanea;
- `PLANS` cancella i piani di esecuzione degli statement preparati.

Concetti

Un *prepared statement* è uno statement SQL privo di valori, che saranno forniti in seguito.
I prepared statement hanno diversi vantaggi:

- permettono di eseguire le query piu' velocemente, a patto che siano ripetitive;
- permettono di fare un buon escaping dei valori.

L'idea è che lo statement viene preparato lato server, poi solo i valori/parametri sono inviati e l'executor può subito eseguire lo statement senza dover ripercorrere le fasi di parsing.

Esempio: PREPARE e EXECUTE

```
> PREPARE find_persona_stmt( char ) AS
   SELECT cognome, nome
   FROM persona
   WHERE substr( codice_fiscale, 1, 1 ) = $1;

> EXECUTE find_persona_stmt( 'c' );
```

Cancellare un prepared statement

Il comando DEALLOCATE annulla un prepared statement. Occorre specificare il nome dello statement, oppure opzinalmente tutti gli statement con ALL:

```
> DEALLOCATE find_persona_stmt;

-- se per errore si tenta di usare lo statement...
> EXECUTE find_persona_stmt( 'c' );
ERROR:  prepared statement "find_persona_stmt" does not exist
```

Introduzione

Ci sono due tipi di backup:

- *logico* effettua un "dump" degli oggetti del database essendo collegato al database stesso e quindi eseguendo una transazione sottoposta a tutte le regole di visibilità dei dati. Il dump è sempre coerente, ma a seconda della dimensione dei dati, potrebbe non includere tutti gli ultimi dati;
- *fisico* effettua una copia del filesystem del cluster (`$PGDATA`). Il problema è che questo tipo di backup non è coerente con le transazioni in corso nel cluster e quindi i dati sono inconsistenti (sostanzialmente è un *crash-recovery* pre-annunciato).

PostgreSQL mette a disposizione appositi programmi per il backup logico (coerente) che possono essere schedulati e automatizzati. Il backup fisico viene solitamente affidato a strumenti del sistema operativo (*snapshosts*, `rsync`, ecc.), ma per essere usabile richiede un setup del cluster particolare (es. *Point In Time Recovery*).

Esistono anche applicazioni esterne che aumentano la consistenza/automazione/sicurezza dei backup, ad esempio *pgBackRest*.

Backup logico da parte del sistema operativo

Diversi sistemi operativi installano anche degli script pre-configurati per il backup schedulato (*backup logico*).
Un ottimo punto di partenza è lo script `502.pgsql` usato con `periodic(8)` di FreeBSD.

pg_dump

Il comando shell `pg_dump` consente di effettuare un *backup logico* di un determinato database.
Il backup è consistente (transazione in modalità *read-committed*).
Supporta output in *plain-text* (istruzioni SQL), archivio compresso, nonché dump parallelo.

pg_dump: dump SQL

```
% pg_dump -h localhost -U postgres \
  -C \
  -f testdb.sql \
  testdb
```

Verrà creato un file `testdb.sql` che contiene i comandi SQL per la creazione (`-C`) del database e il relativo popolamento.

```
% less testdb.sql

CREATE DATABASE testdb
       WITH TEMPLATE = template0
       ENCODING = 'UTF8' LC_COLLATE = 'C' LC_CTYPE = 'C';
ALTER DATABASE testdb OWNER TO luca;
\connect testdb
...
```

pg_dump: opzioni comuni e utili

- `-C` indica di inserire i comandi di creazione del database;
- `-f` indica il file di output (o la directory di output);
  - `-F` può valere `p` (plain SQL), `t` (tar), `d` (directory), `c` (custom) per `pg_restore`;
- `-s` solo definizione dei dati (DDL), non i dati stessi;
- `-a` effettua il dump dei soli dati, senza i comandi DDL;
- `--inserts` effettua il popolamento con una serie di `INSERT` invece che con il comando `COPY` (aumenta la dimensione del file ma lo rende piu' portabile);
- `--column-inserts` effettua il popolamento con una serie di `INSERT` con anche i nomi delle colonne, massima portabilità SQL;
- `-j` numero di thread paralleli per il dump;
- `-Z` indica il livello di compressio (0..9).

pg_dumpall

Il comando `pg_dumpall` effettua il backup di un intero cluster:

- richiama `pg_dump` per ogni database utente definito;
- effettua il dump anche degli oggetti "intra-database" (ruoli, tablespaces, ecc).

Le opzioni sono simili a quelle di `pg_dump`.

Restore via psql

Se il dump fornito è SQL si deve procedere con psql per il restore:

```
% psql -h localhost -U luca template1 < testdb.sql
```

o in modalità interattiva:

```
% psql -h localhost -U luca template1
>\i testdb.sql
```

Restore via pg_restore

Il comando pg_restore consente di ripristinare un archivio non SQL creato con pg_dump e formato *custom* (-Fc).
In default il comando esporta i comandi SQL necessari al ripristino del database.

```
% pg_restore testdb.dump > testdb.sql
```

Se si vuole operare un restore *in place* occorre specificare a quale database collegarsi per avviare una sessione, e lasciare che pg_restore ricarichi tutto dentro al database specificato nell'archivio:

```
% pg_restore -h localhost -U postgres \
    -C \
    -d template1 \
    testdb.dump
```

pg_restore parallelo

Anche il comando pg_restore supporta la modalità parallela.
Si tenga presente che la modalità parallela dipende anche dal numero di oggetti (es. tabelle) da esportare/importare: ad esempio una tabella sola non verrà mai esportata/importata in parallelo.

Esempio di backup con *custom format*

Si supponga di aver creato un backup *custom* (`-Fc`):

```
% pg_dump -Fc -h localhost
          -U postgres
          -f testdb.dump testdb
```

Vedere il contenuto del backup

Il comando pg_restore consente di visionare il contenuto del backup (binario):

```
% pg_restore --list testdb.dump

;
; Archive created at 2018-02-20 19:32:13 CET
;     dbname: testdb
;     TOC Entries: 118
...
254; 1255 16405 FUNCTION public somma_sottrazione(integer, integer) luca
250; 1255 16406 FUNCTION public tr_check_eta() luca
1831; 1417 16688 SERVER - another_pg postgres
2735; 0 0 USER MAPPING - USER MAPPING postgres SERVER another_pg postgres
...
```

Riordinare il restore

Loutput di `--list` può a sua volta essere usato, manipolato, come input di `-L` per:

- escludere determinati oggetti dal ripristino (elimando le righe o commentadole con un ;);
- riordinare l'esecuzione di ripristino.

```
% pg_restore --list testdb.dump > testdb.restore.list
% emacs testdb.restore.list
% pg_restore -L testdb.restore.list testdb.dump
```

PostgreSQL supporta due meccanismi di memorizzazione di oggetti binari "grossi":

- *large object* che fornisce un'API a stream (quindi utile per memorizzare e recuperare parti dell'oggetto);
- *TOAST* (The Oversize Attribute Storage Technique) che si usa quando l'oggetto è piu' grande di una singola pagina dati.

Si tenga presente che i large object memorizzano dati fino a 4TB mentre TOAST permette di memorizzare non piu' di !GB su singolo campo.
Tutti i *large objects* dispongono di una API a stream e sono memorizzati sempre nella tabella `pg_largeobject` e la sua duale `pg_largeobject_metadata`.
I large objects sono divisi in *chunk*, ovvero pezzi di stream binario da memorizzare come tuple. I large objects supportano dati sparsi, ovvero è possibile scrivere parti di un large object senza partire dall'inizio (le letture vedranno degli zeri per le parti non ancora allocate).
Le funzioni messe a disposizione da PostgreSQL per manipolare i large objects sono un clone delle funzioni di sistema Unix per la manipolazione dei file (es. `open(2)`).
I file descriptor di un large object (denominati `Oid`) persistono solo

all'interno di un blocco di transazione, quindi i large objects vanno gestiti all'interno di una transazione!

I large objects sono manipolati con funzioni *client-side* (`libpq`) o server-side. L'unica differenza è che le ultime richiedono accesso come superutente, mentre quelle client-side no.

Le principali funzioni server-side sono:

- `lo_creat()` crea l'oggetto con l'oid specificato o ne seleziona uno nuovo se `-1`;
- `lo_unlink()` distrugge il large object con l'oid specificato;
- `lo_import()` importa il file specificato (eventualmente nell'oid specificato);
- `lo_export()` esporta il large object specificato come oid indicandone il file fisico;
- `lo_put()`, `lo_get()` scrivono e leggono dati collegati a un large object;
- `loread()`, `lowrite()` leggono dei byte da un large object;
- `lo_lseek()` effettua un *lseek* sul large specificato come oid.

Tabella di esempio

Si deve creare una tabella con un dato di tipo `oid` per poterla collegare al `pg_largeobject`:

```
> CREATE TABLE picture(
   pk SERIAL PRIMARY KEY,
   data oid NOT NULL,
   description text );
```

Importare un large object in un colpo solo

```
> INSERT INTO picture( description, data )
  VALUES ( 'Bud Spencer & Terence Hill',
           lo_import( '/home/luca/Downloads/bud.jpg' ) );
```

Dove è finita l'immagine?

All'immagine è stato assegnato oid 16691:

```
> SELECT * FROM picture;
 pk | data  |          description
----+-------+-------------------------------
  1 | 16691 | Bud Spencer & Terence Hill
```

delle informazioni fisiche sono state iserite nella tabella `pg_largeobject`, in particolare l'immagine è stata *divisa* nei chunk:

```
> SELECT loid, count( pageno )
   FROM pg_largeobject
   WHERE loid = 16691
   GROUP BY loid;
 loid  | count
-------+-------
 16691 |   104
```

ossia ci sono 184 chunk, la cui somma di lunghezza:

```
> SELECT SUM( length( data ) )
  FROM pg_largeobject
  WHERE loid = 16691;
 sum
--------
 212164
```

Esportare il file

```
> SELECT lo_export( 16691, '/tmp/export.jpg' );
```

ATTENZIONE: si devono avere i permessi di scrittura sull'oggetto!

e da un controllo sul file system i due oggetti sono gli stessi:

```
% md5sum ~/Downloads/bud.jpg /tmp/export.jpg
e39cbea9c8f95be157dd6c5abc82c50d  /home/luca/Downloads/bud.jpg
e39cbea9c8f95be157dd6c5abc82c50d  /tmp/export.jpg
```

Sovrascrivere un large object

Si può importare un large object sopra ad uno esistente usando lo stesso `oid`:

```
> SELECT lo_unlink( 16691 );
> SELECT lo_import( '/home/luca/Downloads/sofia.png',
                    16691 );
```

Si noti che non è stata manipolata la tabella `picture`!

Leggere dati da un large object

Dal lato server-side è meglio usare le funzioni `lo_get` e `lo_put`:

```
> SELECT lo_get( 16691, 0, 10 );
          lo_get
------------------------
 \x89504e470d0a1a0a0000
```

che legge i primi 10 byte e restituisce un `bytea` come risultato.

Superare la dimensione delle pagine dati

Le pagine dati in PostgreSQL sono a dimensione fissa, solitamente 8kB; per questo nessun attributo di tabella può superare la dimensione fisica di una pagina dati.
Per evitare il problema quando un attributo diventa troppo grande per una pagina dati PostgreSQL collega automaticamente e trasparentemente una tabella *TOAST* (collegata mediante `pg_class.reltoastrelid` che contiene l'attributo *spezzato* su piu' tuple.
Perché si attivi TOAST il tipo di attributo deve essere TOAST-abile (lunghezza variabile). TOAST usa due bit dei 32 a disposizione per funzionamento interno e quindi ogni attributo diventa al massimo di 2^30 = 1GB grosso.
Il sistema TOAST si attiva quando un attributo supera solitamente i 2kB, ovvero quando non riescono a essere contenuti 4 tuple con quell'attributo in una singola pagina dati.

Vantaggi di TOAST

L'idea dietro a TOAST è semplice: solitamente le query prevedono clausole costruite su valori *piccoli* e quindi l'ottimizzatore può continuare a lavorare trasparentemente. Quando è ora di estrarre i dati, questi vengono de-toastati opportunamente.
Se il dato si trova ancora sulla tabella principale si dice che è *on-line*, se è stato spostato nella tabella TOAST si dice che è *offline*.

Tipi di memorizzazione TOAST

Ci sono quattro strategie di memorizzazione TOAST:

- `PLAIN` il dato non può essere toastato, ossia non viene compresso e non viene messo offline nella tabella toast;
- `MAIN` il dato viene compresso ma non può essere spostato offline nella tabella di appoggio TOAST;
- `EXTENDED` è il default, il dato viene compresso e spostato offline nella tabella TOAST;
- `EXTERNAL` il dato viene messo offline ma non può essere compresso.

Introduzione

Il *Multi Version Concurrency Control (MVCC)* è un meccanismo per garantire alta concorrenza in accesso ai dati (tuple).
Non è prerogativa di PostgreSQL, viene usato anche in altri DBMS!
L'idea è concettualmente semplice: per evitare che processi di scrittura blocchino (lock) i processi di lettura e viceversa, ogni transazione vede uno *snapshot* dei dati, ossia una "istantanea" dei dati. Ogni tupla contiene delle *informazioni di visiilità*che indicano quali transazioni possono vedere la tupla stessa e quali no.

MVCC Bloating

Al fine di poter mantenere lo snapshot dei dati, ogni volta che una tupla viene "toccata" in scrittura (es. `UPDATE`) essa viene *invalidata* e ne viene posizionata in "append" una nuova con i relativi valori aggiornati.
In altre parole: *ci possono essere contemporaneamente piu' versioni della stessa tupla!*
Queste produce un "rigonfiamento" dei dati (*bloating*) che viene risolto da un apposito meccanismo detto *vacuuming*.

Informazioni di visibilità

Ogni tabella ha quattro colonne nascoste, che quindi sono disponibili su ogni tupla:

- `xmin` indica il *transaction id* della transazione che ha generato la tupla;
- `xmax` indica il *transaction id* della transazione che ha invalidato la tupla;
- `cmin` indica il *command id* che ha generato la tupla all'interno di una transazione;
- `cmax` indica il *command id* che ha invalidato la tupla all'interno di una transazione.

Se `xmax` vale zero allora la tupla è valida (non è stata invalidata). Se `cmin` e/o `cmax` sono zero allora il comando non faceva parte di una transazione esplicita (`BEGIN ... END`).

Informazioni di visibilità: estrarle

Nominando le colonne "nascoste" si possono vedere le informazioni di visibilità:

```
> SELECT xmin, xmax, cmin, cmax, pk
  FROM persona LIMIT 2;
 xmin | xmax | cmin | cmax | pk
------+------+------+------+----
 1202 |    0 |    0 |    0 | 10
 1202 |    0 |    0 |    0 | 11
```

Quando è visibile una tupla?

In generale la tupla è visibile se: a) la tupla è stata creata da questa transazione e non è stata invalidata se non dalla transazione stessa

```
xmin == current_xid && cmin >= current_command_id
   &&
xmax == 0 || ( xmax == current_xid && cmax >= current_command_id )
```

b) la tupla è stata creata prima di questa transazione (o comunque committata) e non è stata invalidata se non dalla transazione corrente o da una transazione che non ha ancora fatto il commit

```
xmin committed
&&
( xmax == 0 || xmax == current_xid || xmax ! commited )
```

Il test è complesso! In generale la regola è `xmin == current_xid || xmin committed` per candidarla come possibile tupla visibile: e `xmax == 0 || xmax == current_xid || xmax ! committed` per stabilire se non è stata invalidata.

Savepoint

I `SAVEPOINT` sono delle sotto-transazioni che consentono di rendere persistente una parte di transazione. Questo ha delle implicazioni sull'uso dei campi `xmin` e `xmax` poiché all'interno di uno stesso blocco `BEGIN...END` possono esserci piu' transazioni in esecuzione.
Il parametro di configurazione `ON_ERROR_ROLLBACK` di `psql` utilizza dei savepoint appunto per simulare questo comportamento ed evitare di abortire tutta una transazione.

Un esempio di utilizzo dei campi

```
> CREATE TABLE foo( i int );
> BEGIN;
> INSERT INTO foo(i) VALUES( 1 );
> INSERT INTO foo(i) VALUES( 2 );
> SELECT xmin, cmin, xmax, cmax, i FROM foo;
 xmin | cmin | xmax | cmax | i
------+------+------+------+---
 2488 |    0 |    0 |    0 | 1
 2488 |    1 |    0 |    1 | 2
-- xmax = 0 e quindi le tuple sono valide!

> SELECT txid_current();
 txid_current
--------------
         2488
```

Un esempio di utilizzo dei campi (2)

```
-- nella stessa transazione
> DECLARE my_cursor CURSOR FOR
  SELECT xmin, cmin, xmax, cmax, i FROM foo;

> DELETE FROM foo;

> FETCH ALL FROM my_cursor;
 xmin | cmin | xmax | cmax | i
------+------+------+------+---
 2488 |    0 | 2488 |    0 | 1
 2488 |    1 | 2488 |    1 | 2
```

Un esempio di utilizzo dei campi: cosa è successo?

Il cursorse è stato definito ad un certo punto, e quindi deve poter vedere uno *snapshot* dei dati. I dati cancellati dalla seguente `DELETE` vengono rimossi (`xmax` viene valorizzato) ma sono ancora visibili al cursore che è stato definito prima!

Snapshot

Come fa il sistema a sapere quali transazioni sono ancora attive? La funzione `txid_current_snapshot()` fornisce gli `xmin` e `xmax` delle transazioni ancora attive quanto quella corrente è partita. In una sessione:

```
> BEGIN;
> SELECT txid_current();
 txid_current
--------------
         2491
```

e contemporaneamente in un'altra sessione:

```
> BEGIN;
> INSERT INTO foo(i) VALUES( 1 );
> SELECT xmin, xmax, cmin, cmax, i FROM foo;
 xmin | xmax | cmin | cmax | i
------+------+------+------+---
 2492 |    0 |    0 |    0 | 1

> SELECT * FROM txid_current_snapshot();
 txid_current_snapshot
-----------------------
 2491:2491:
```

quindi la transazione 2492 sa che la 2491 è ancora attiva e non ancora conclusa, ma tutte quelle prima della 2491 sono terminate

HOT

Il meccanismo *Heap Only Tuples* (HOT) permette di ridurre l'I/O necessario in `UPDATE` di colonne non in indice.
L'idea è semplice: se la pagina dati contiene ancora spazio per l'inserimento della nuova versione della tupla, questa viene inserita nella pagina dati e la precedente versione è marcata come espirata. Questo significa che non occorre aggiornare l'indice, riducendo quindi il livello di I/O complessivo.
Se la pagina dati non contiene spazio, o le colonne soggette ad aggiornamento fanno parte di un indice, allora HOT non è possibile.

Introduzione

Il comand `VACUUM` permette di "ricompattare" lo spaizo disco andando ad eliminare quelle tuple che non sono piu' *naturalmente* visibili (ossia sono state invalidate per ogni transazione).
Il comando può agire su un intero database o su una singola tabella, ed è I/O intensive (deve riorganizzare le pagine dati).
Il comando `VACUUM` può essere combinato con `ANALYZE`: si ricompattano i dati e si aggiornano le statistiche usate dall'ottimizzatore.
`VACUUM` può spostare il file fisico di una relazione (ad esempio se la riscrive elimnando le tuple rimosse).

Il problema dello xid wraparound

PostgreSQL numera le transazioni in un intero a 32 bit denominato *xid*. Ogni transazione ottiene uno xid progressivo (e quindi superiore) rispetto alle transazioni precedenti. Questo permette di identificare una transazione nel futuro o nel passato comparando gli *xid*.
Tuttavia prima o poi lo *xid* subirà il *wraparound*, e precisamente dopo 2^{31} transazioni le nuove transazioni otterranno un numero identificativo minore (riavviato) e improvvisamente appariranno nel passato.

Quando ci si avvicina allo xid wraparound

Per sicurezza PostgreSQL interrompe le proprie attività quando ci si avvicina al rischio di wraparound. Nelle versioni molto vecchie questo imponeva un `VACUUM` manuale, nelle versioni attuali l'autovacuum previene il problema:

```
WARNING:  database "mydb" must be vacuumed within 177009986 transactions
HINT:  To avoid a database shutdown, execute a database-wide VACUUM in "mydb".
...
RROR:  database is not accepting commands to avoid wraparound data loss in database
HINT:  Stop the postmaster and vacuum that database in single-user mode.
```

xid wraparound: soluzioni

In passato (< 9.4) PostgreSQL numerava le transazioni partendo da 3 e usava il valore 2 come `FrozenTransactionId`. Era quindi sufficiente modificare lo `xmin` di ogni tupla con `FrozenTransactionId` per fare in modo che questa *comparisse sempre nel passato*. (il valore 1 era usato come `BootstrapTransactionId` per indicare che l'oggetto è stato creato durante l'inizializzazione del database)
Nelle versioni attuali di PostgreSQL viene mantenuto lo `xmin` originale (utile per debugging) e viene aggiunto un flag *frozen* per indicare che la tupla è *nel passato*.

xid wraparound: VACUUM FREEZE

Il comando VACUUM accetta un'opzione particolare, FREEZE che va a controllare ogni tupla e la marca come *frozen* (quindi sempre nel passato) se possibile.
Per ottimizzare l'I/O, VACUUM controlla la *visibility map* di ogni pagina e salta le pagine dati che non hanno tuple "espirate", quindi non applicando il FREEZE a tutte le tuple. Per effettuare l'analisi anche di queste pagine dati occorre dare l'opzione DISABLE_PAGE_SKIPPING.

Vacuum in azione

```
# VACUUM FREEZE VERBOSE software;
INFO:  vacuuming "public.software"
INFO:  index "software_pkey" now contains 8 row versions in 2 pages
DETAIL:  0 index row versions were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
INFO:  index "software_name_version_key" now contains 8 row versions in 2 pages
DETAIL:  0 index row versions were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
INFO:  "software": found 4 removable, 8 nonremovable row versions in 1 out of 1 pag
DETAIL:  0 dead row versions cannot be removed yet.
There were 2 unused item pointers.
Skipped 0 pages due to buffer pins.
0 pages are entirely empty.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
INFO:  vacuuming "pg_toast.pg_toast_16559"
INFO:  index "pg_toast_16559_index" now contains 0 row versions in 1 pages
DETAIL:  0 index row versions were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
INFO:  "pg_toast_16559": found 0 removable, 0 nonremovable row versions in 0 out of
DETAIL:  0 dead row versions cannot be removed yet.
There were 0 unused item pointers.
```

Tabella di eventi

Si immagini di popolare una tabella di eventi:

```
> CREATE TABLE evento( pk serial NOT NULL,
                       description text,
                       ts timestamp default current_timestamp,
                        PRIMARY KEY(pk) );
```

e di popolarla con alcuni dati...

```
> INSERT INTO evento( description )
 SELECT 'Evento ' ||
    CASE WHEN ( random() * 10 )::integer % 2 = 0
         THEN 'pari'
         ELSE 'dispari'
    END
 FROM generate_series( 1, 1000000 );

INSERT 0 1000000
Time: 8635.289 ms
```

Tabella di eventi: quanto occupa?

```
> SELECT relname, relpages, reltuples
FROM pg_class
WHERE relkind = 'r' AND relname = 'evento';
 relname | relpages | reltuples
---------+----------+-----------
 evento  |     6879 |     1e+06
```

indicativamente 6879 * 8kB = 53,74 MB di spazio disco. Verifichiamo...

```
% oid2name -H localhost -U luca -d testdb -t evento
From database "testdb":
  Filenode  Table Name
----------------------
     16465       evento

% oid2name -H localhost -U luca
  16393         testdb

% sudo ls -lh /mnt/data1/pgdata/base/16393/16465
-rw-------  1 postgres  postgres    54M Oct 18 10:40 /mnt/data1/pgdata/base/16393/1
```

Tabella di eventi: quanto occupa? (2)

Metodo meno "scomodo" di cercare il file fisico su disco...

```
> SELECT pg_size_pretty(
             pg_relation_size( 'evento'::regclass )
         );
 pg_size_pretty
----------------
 54 MB
(1 row)
```

Tabella di eventi: statistiche

Le statistiche di sistema sembrano funzionare correttamente:

```
> SELECT pg_relation_size( 'evento'::regclass ) AS dimensione_fisica, relpages, rel
  FROM pg_class  WHERE relkind = 'r' AND relname = 'evento';
 dimensione_fisica | relpages | reltuples
-------------------+----------+-----------
          56352768 |     6879 |     1e+06

> TRUNCATE evento;

> SELECT pg_relation_size( 'evento'::regclass ) AS dimensione_fisica, relpages, rel
  FROM pg_class  WHERE relkind = 'r' AND relname = 'evento';
 dimensione_fisica | relpages | reltuples
-------------------+----------+-----------
                 0 |        0 |         0
```

Tabella di eventi: statistiche (2)

Ma cosa succede se fermiamo autovacuum?

```
 > INSERT INTO evento( description ) -- 1000000 tuple
...
INSERT 0 1000000

> SELECT pg_relation_size( 'evento'::regclass ) AS dimensione_fisica, relpages, rel
  FROM pg_class  WHERE relkind = 'r' AND relname = 'evento';
 dimensione_fisica | relpages | reltuples
-------------------+----------+-----------
          56344576 |        0 |         0
```

Il numero di tuple e pagine non è stato aggiornato. Che impatto ha questo?

Il planner non sarà in grado di capire che "mole di dati" ha davanti!

Tabella di eventi: statistiche (3)

Niente panico! E' possibile lanciare vacuum manualmente.

```
> VACUUM FULL VERBOSE evento;
INFO:  vacuuming "public.evento"
INFO:  "evento": found 0 removable, 1000000 nonremovable row versions in 6878 pages
DETAIL:  0 dead row versions cannot be removed yet.
CPU 0.14s/0.30u sec elapsed 0.82 sec.

> SELECT pg_relation_size( 'evento'::regclass ) AS dimensione_fisica, relpages, rel
  FROM pg_class  WHERE relkind = 'r' AND relname = 'evento';
 dimensione_fisica | relpages | reltuples
-------------------+----------+-----------
          56344576 |     6878 |     1e+06
```

Free Space Map

PostgreSQL memorizza su file system, per ogni relazione, una *Free Space Map*, identificata da un file con nome pari all'oid della relazione e suffisso `.fsm`.

La mappa è organizzata come un albero binario:

- ogni foglia memorizza la quantità di spazio disponibile nella relativa pagina dati della relazione;
- i livelli intermedi memorizzano il valore massimo dato dai figli.

In questo modo è possibile sapere sempre lo spazio ancora disponibile nella relazione.

Visibility Map

Ogni pagina dati di una relazione ha una *Visibility Map*, fisicamente memorizzata su disco con nome pari a quello della relazione e suffisso `.vm`. La mappa contiene due bit per pagina ad indicare:

- se tutte le tuple sono visibili (nessuna necessità di `VACUUM`) e quindi si può effettuare un *Index Only Scan*;
- se tutte le tuple sono *frozen* e quindi non necessitano di nessun `VACUUM` per evitare perfino il wraparound dello `xid`.

La mappa è conservativa: se i bit sono attivi allora la condizione relativa è sicuramente vera, se non lo sono potrebbe essere vera o falsa.

NULL Map

Per ogni tupla la pagina datai contiene una mappa con un bit che indica se la relativa tupla contiene valori `NULL` nella colonna specificata.

Introduzione

Lo scopo di *MVCC* è quello di ridurre al massimo i lock (e in particolare il lock contention), permetetndo quindi la massima concorrenza.
Nonostante questo, alcuni comandi richiedono dei *lock* sui dati per garantire un accesso corretto.
PostgreSQL riconosce diversi tipi di lock, ciascuno con alcune incompatibilità con gli altri: le incompatibilità possono creare *contention*, ovvero rallentamenti nella concorrenza, e in alcuni casi perfino dei *deadlock*.

Tipi di lock: tabella

Tutti i lock sono a livello di tabella, anche se il nome include la dicitura `row` (per retrocompatibilità).

- `ACCESS_SHARE` fornisce sola lettura, es. `SELECT`;
- `ROW_SHARE` fornisce lock in scrittura, es. `SELECT FOR UPDATE`;
- `ROW_EXCLUSIVE` lock per modifica ai dati, es `UPDATE`, `DELETE`, `INSERT`;
- `SHARE_UPDATE_EXCLUSIVE` modifica di dati da pate di comandi di utilità, ad es. `VACUUM`, `CREATE INDEX CONCURRENTLY`, `ALTER TABLE`, ...;
- `SHARE` per la creazione di indici non concorrenti, `CREATE INDEX`;
- `SHARE_ROW_EXCLUSIVE` per la creazione di trigger e alcune versioni di `ALTER TABLE`;
- `EXCLUSIVE` usato per `REFRESH MATERIALIZED VIEW CONCURRENTLY`;
- `ACCESS_EXCLUSIVE` usato per comandi aggressivi, es. `DROP`, `TRUNCATE`, `VACUUM FULL`, ...

Si noti che una `SELECT` viene bloccata solo da un `ACCESS_EXCLUSIVE`.

Tipi di lock: tupla

I lock a livello di tupla sono usati *automaticamente* e non bloccano mai l'accesso ai dati in lettura, ma solo in scrittura o per un lock esplicito di tabella.

- `FOR UPDATE` spesso usato con `SELECT FOR UPDATE` blocca le tuple selezionate da modifiche in altre transazioni;
  - `FOR NO KEY UPDATE` come il precedente, ma lock piu' leggero e consente altre transazioni di bloccare le tuple con `FOR KEY SHARE`;
- `FOR SHARE` lock leggero, permette alle altre transazioni di acquisire un lock dello stesso tipo sulle stesse tuple;
  - `FOR KEY SHARE` lock leggero, impedisce alle altre transazioni di acquire lock che modifichino le chiavi ma consente modifiche agli altri dati;

Deadlock

La possibilità di richiedere lock diversi sugli stessi oggetti da parte di transazioni differenti produce il problema di *deadlock*.
Un deadlock è un evento naturale e PostgreSQL può individuare i deadlock facendo terminare le relative transazioni.

Deadlock: esempio

Transazione A
BEGIN;
UPDATE persona SET cognome = 'FERRARI' WHERE nome = 'Luca';
UPDATE persona SET cognome = 'FERRARI' WHERE nome = 'Diego';
– bloccata in attesa di lock per tupla

```
ERROR:  deadlock detected
DETAIL:  Process 952 waits for ShareLock on transaction 3019; blocked by process 93
Process 935 waits for ShareLock on transaction 3020; blocked by process 952.
HINT:  See server log for query details.
CONTEXT:  while updating tuple (0,1) in relation "persona"
```

Individuare i deadlock

Il parametro `deadlock_timeout` nel file `postgresql.conf` specifica un intervallo di tempo affinché il cluster controlli se sono presenti deadlocks e, conseguentemente, uccida le relative transazioni.

```
deadlock_timeout = 1s
```

La rilevazione dei deadlock è un'operazione costosa, quindi questo tempo indica al cluster con che intervallo preoccuparsi dei deadlocks.

Advisory locks

PostgreSQL supporta anche la definizione di lock a livello applicativo, chiamati *advisory lock*.
Questi lock sono interamente a carico dell'applicazione: per ogni operazione di *lock* ci deve essere una corrispondente *unlock*.
I lock possono essere creati a livello di sessione o di transazione, i primi sono pericolosi perché possono persistere una transazione che effettua un `ROLLBACK`.
PostgreSQL mette a disposizione le funzioni `pg_advisory_lock()` e `pg_advisory_unlock()` per gestire i lock.

Esempio di advisory lock

Transazione A	Tra
BEGIN;	BE
SELECT $pg_{advisorylock}$(pk) FROM persona WHERE nome = 'Luca';	
– work	SE
	– b
SELECT $pg_{advisoryunlock}$(pk) FROM persona WHERE nome = 'Luca';	– s

Vedere i lock

La vista speciale pg_locks contiene le informazioni circa le richieste di lock, e può quindi essere usata per ottenere informazioni su chi ad esempio è sospeso in attesa di lock.

```
> SELECT a.usename, a.application_name, a.datname, a.query,
         l.granted, l.mode
     FROM pg_locks l
     JOIN pg_stat_activity a ON a.pid = l.pid;

-[ RECORD 13 ]---+-----------------------------------------------------
usename          | luca
application_name | psql
datname          | testdb
query            | UPDATE persona SET cognome = 'FERRARI' WHERE nome = 'Diego';
granted          | t
mode             | RowExclusiveLock
```

Scovare lock contention

E' un processo in tre fasi:

1. vedere se esistono query bloccate in attesa di lock, e in particolare se sono attese da molto tempo (è normale che qualche transazione si blocchi in contesti di alta concorrenza);
2. verificare lo stato dei lock e capire quale transazione ha acquisito il lock senza ancora rilasciarlo;
3. stabilire se terminare o meno il backend con la query che non ha rilasciato il lock.

1: Scoprire se qualcuno blocca

Mediante pg_stat_activity si possono trovare le query che risultano bloccate in attesa di acquisire un lock:

```
# SELECT query, backend_start, xact_start, query_start,
         state_change, state,
         now()::time - state_change::time AS locked_since,
         pid, wait_event_type, wait_event
  FROM pg_stat_activity
  WHERE wait_event_type IS NOT NULL
ORDER BY locked_since DESC;

-[ RECORD 1 ]---+---------------------------------------
query           | UPDATE persona SET nome = upper( nome );
backend_start   | 2017-11-15 09:37:55.88444+01
xact_start      | 2017-11-15 09:37:58.186149+01
query_start     | 2017-11-15 09:38:16.362419+01
state_change    | 2017-11-15 09:38:16.362424+01
state           | active
locked_since    | 00:05:10.078716
pid             | 786
wait_event_type | Lock
wait_event      | transactionid
```

La query agganciata al processo 786 è in attesa di un Lock da parte di un'altra transazione (transactionid). *Il fatto che la query sia in attesa*

2: vedere se effettivamente non ha ottenuto il lock

```
# SELECT a.usename, a.application_name, a.datname, a.query,
         l.granted, l.mode, transactionid
    FROM pg_locks l
    JOIN pg_stat_activity a ON a.pid = l.pid
    WHERE granted = false AND a.pid = 786;
-[ RECORD 1 ]----+---------------------------------------
usename          | luca
application_name | psql
datname          | testdb
query            | UPDATE persona SET nome = upper( nome );
granted          | f
mode             | ShareLock
transactionid    | 3031
```

Quindi la query con pid 786 è in attesa di lock di tipo ShareLock (per aggiornare dati) dalla transazione 3031.

3: Vedere chi blocca l'acquisizione del lock

```
# SELECT a.usename, a.application_name, a.datname, a.query,
         l.granted, l.mode, transactionid,
         now()::time - a.state_change::time AS acquired_since,
         a.pid
    FROM pg_locks l
    JOIN pg_stat_activity a ON a.pid = l.pid
    WHERE granted = true AND transactionid = 3031;
-[ RECORD 1 ]----+-----------------------------------------
usename          | luca
application_name | psql
datname          | testdb
query            | UPDATE persona SET cognome = lower( cognome );
granted          | t
mode             | ExclusiveLock
transactionid    | 3031
acquired_since   | 00:27:20.264908
pid              | 780
```

La transazione 3031 ha effettivamente acquisito un lock da circa 27 minuti per eseguire l'UPDATE mostrato.

Vedere chi blocca i lock: prendere una decisione

Siccome la transazione 3031 sta bloccando un'altra transazione da 27 minuti, è bene decidere cosa fare. Qualora la mole di dati da processare sia ridotta rispetto al tempo impiegato, si può desumere che la transazione è stata *abbandonata*.
Si può quindi procedere alla sua eliminazione tramite il relativo pid:

```
# SELECT pg_terminate_backend( 780 );
```

Autocommit

Il terminale `psql` utilizza in default l'*autocommit*. La variabile che controlla questo comportamento è `AUTOCOMMIT`, che se impostata a 0 apre una transazione non committandola:

```
[luca @ testdb on localhost] 1  > \set AUTOCOMMIT 0

[luca @ testdb on localhost] 1  > TRUNCATE pictures;
-- l'asterisco indica che c'è una transazione non terminata!
[luca @ testdb on localhost] 1 * > COMMIT;
```

Livelli di isolamento

Le problematiche sono:

- dirty reads: è possibile vedere (leggere) tuple non ancora committate da altre transazioni;
- unrepeatable reads: una query ripetuta piu' volte riporta risultati con valori differenti (valori dei dati);
- phantom reads: la stessa eseguita con gli stessi filtri fornisce un result set differente (dimensione dei dati).

Livelli di isolamento in PostgreSQL

PostgreSQL supporta 3 livelli di isolamento sui 4 standard (*read uncommited*, *read committed*, *repeatable read*, *serializable*). Questo non rappresenta un problema poiché è sufficiente garantire il *livello di isolamento superiore*.
Una transazione può essere di tipo:

- read committed: *(default)* uno statement vede solo le tuple consolidate prima dell'inizio dello statement stesso;
- repeatable read: ogni statement della transazione può vedere solo le tuple consolidate prima dell'esecuzione del primo statement (ovvero, all'inizio della transazione);
- serializable: come repeatable read ma impone che le transazioni siano *idempotenti* rispetto al loro effettivo ordine di esecuzione (monitoring di consistenza seriale).

Impostare i livelli di isolamento in PostgreSQL

All'interno di una transazione è possibile impostare il livello di isolamento, che però non può essere piu' cambiato dopo l'esecuzione di uno statement di modifica (incluso `SELECT`). Quando si avvia una transazione è possibile specificare l'isolation level direttamente nel blocco `BEGIN`, mentre per gli altri casi c'è `SET TRANSACTION`:

```
> BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
ERROR:  SET TRANSACTION ISOLATION LEVEL must not be called in a subtransaction
```

Transazioni `READ ONLY` contro `READ WRITE`

PostgreSQL permette di specificare due tipi di transazioni:

- READ WRITE: (default) è possibile eseguire ogni statement SQL;
- READ ONLY: è una *promessa* di non alterare i dati. Tutti i comandi generici (`UPDATE`, `INSERT`, `DELETE`, `EXECUTE`, `ALTER TABLE`, `VACUUM`, ecc.) sono disabilitati a meno che non agiscano su tabelle temporanee.
  Non c'è nessuna garanzia che una transazione `READ ONLY` non effettui scritture su disco!

Transazioni DEFERRABLE

C'è un caso particolare di transazione: `DEFERRABLE`. Questo si applica solo a transazioni `READ ONLY` e `SERIALIZABLE`.
L'idea è che la transazione accetta di "attendere" prima della sua esecuzione, e in cambio l'overhead di monitoring per la serializzazione è ridotto. Ciò produce dei vantaggi per le transazioni di reportistica.

Il tempo nelle transazioni

Il tempo nelle transazioni è discreto: `now()` e `CURRENT_TIMESTAMP` restituiscono sempre lo stesso valore (sono definite come *stable*). Esiste la funzione speciale `clock_timestamp()` che restituisce il valore basato sul clock di sistema, quindi differente ogni volta:

```
> BEGIN;
> SELECT now(), clock_timestamp();
-[ RECORD 1 ]---+-----------------------------
now             | 2018-02-17 12:05:05.483757+01
clock_timestamp | 2018-02-17 12:05:14.071801+01
--                              ^^^^
```

Funzioni

PostgreSQL permette la creazione di funzioni tramite `CREATE FUNCTION`.
Le funzioni possono accettare dei parametri (eventualmente tipizzati e con valori di default), possono definire delle variabili (eventualmente tipizzate) e avere un valore di ritorno (tipizzato).
Le funzioni sono create in uno specifico linguaggio, solitamente `pgpsql` e sono identificate dalla loro *signature*.

Template di CREATE FUNCTION

```
CREATE OR REPLACE FUNCTION function_name( <arg>, <arg>, ... )
RETURNS <return type>
AS
-- definition
LANGUAGE language [<volatilità>];
```

Prototiti di funzione

- la lista di parametri può includere solo i tipi, nel qual caso i parametri sono accessibili con $1, $2, ecc.;
- i parametri possono includere un valore di default specificato con `DEFAULT`;
- i parametri possono essere in ingresso (`IN`, default), in uscita (`OUT`, non importa farne il `RETURN`) o in entrambi (`INOUT`).

Volatilità di una funzione

Le funzioni hanno tre livelli di *volatilità*:

- `VOLATILE`: *default*, modifica il database e può dare risultati diversi anche se inocata con gli stessi argomenti;
- `STABLE`: non modifica il database e ritorna lo stesso valore con gli stessi argomenti *per tutte le tuple nello stesso statement* (è indipendente dal valore delle singole tuple), utile per operatori di confronto (es. negli indici);
- `IMMUTABLE`: non modifica il database e ritorna lo stesso valore con gli stessi argomenti *sempre*.
  Esempi: `random()` è `VOLATILE`, `current_timestamp` è `STABLE`, `version()` è `IMMUTABLE`.

Security

Una stored procedure può essere definita in due contesti di sicurezza:

- `SECURITY INVOKER` la funzione esegue con i permessi dell'utente che la invoca (ovvero deve avere le `GRANT` sulle tabelle);
- `SECURITY DEFINER` la funzione esegue con i permessi dell'utente che l'ha definita, e quindi non sono necessarie le grant all'utente invocante.

Informazioni di sistema

E' possibile incrociare le viste pg_proc e pg_language per ottenere le informazioni relative ad una stored procedure, e nel caso sia interpretata anche il codice sorgente:

```
> SELECT p.proname, l.lanname, p.prosrc
   FROM pg_proc p
   JOIN pg_language l ON p.prolang = l.oid
   WHERE proname = 'do_count';

-[ RECORD 1 ]----------------------------------------------------------
proname | do_count
lanname | plpgsql
prosrc  |
        | DECLARE
        |         v_count int;
        | BEGIN
        |         EXECUTE 'SELECT count(*) FROM persona WHERE eta >= $1'
        |         INTO STRICT v_count
        |         USING eta_min;
        |
        |         RETURN v_count;
        | END;
```

E' il linguaggio *default* per la gestione di blocchi di codice:

- prevede la dichiarazione delle variabili di funzione con `DECLARE`;
- racchiude il body della funzione fra `BEGIN` e `END`
- consente cicli e istruzioni condizionali.

Un blocco di codice `plpgsql` è composto sempre da:

```
DECLARE
   ...
BEGIN
  ...
END;
```

Si noti che `BEGIN/END` non definiscono i confini di una transazione!

Esempio di semplice funzione: somma

```
CREATE OR REPLACE FUNCTION somma( int, int )
RETURNS INTEGER
AS
$BODY$
DECLARE --nessun variabile
BEGIN
        RETURN $1 + $2;
END;
$BODY$
LANGUAGE plpgsql;
```

```
> select somma( 10, 20 );
 somma
-------
    30
```

Esempio di semplice funzione: somma con alias parametri

Simile alle *sub* di Perl 5!

```
CREATE OR REPLACE FUNCTION somma( int, int )
RETURNS INTEGER
AS
$BODY$
DECLARE
        a ALIAS FOR $1;
        b ALIAS FOR $2;
BEGIN
        RETURN a + b;
END;
$BODY$
LANGUAGE plpgsql;
```

Esempio di semplice funzione: somma con parametri nominali

```
CREATE OR REPLACE FUNCTION somma( a int, b int )
RETURNS INTEGER
AS
$BODY$
DECLARE
BEGIN
        RETURN a + b;
END;
$BODY$
LANGUAGE plpgsql;
```

Esempio di semplice funzione: somma e sottrazione

```
CREATE OR REPLACE FUNCTION somma_sottrazione( INOUT a int ,
                                              INOUT b int )
AS $BODY$
DECLARE
BEGIN
        a = a + b;
        b = a - b;
END; $BODY$
LANGUAGE plpgsql;
```

Esempio di semplice funzione: somma e sottrazione (2)

```
> SELECT somma_sottrazione( 10, 20 );
 somma_sottrazione
-------------------
 (30,10)

> SELECT b, a FROM somma_sottrazione( 10, 20 );
 b  | a
----+----
 10 | 30
```

Esempio di semplice funzione: somma con valori di default

I valori di default possono essere omessi nella chiamata di funzione (NULL è considerato un valore):

```
CREATE OR REPLACE FUNCTION somma( a int DEFAULT 0,
                                  b int DEFAULT 0 )
RETURNS INTEGER
AS $BODY$
DECLARE
BEGIN
        RETURN a + b;
END;
$BODY$
LANGUAGE plpgsql;
```

```
 > SELECT somma();
 somma
-------
     0
```

Esempio di semplice funzione: costanti

E' possibile definire delle costanti in fase di dichiarazione di variabili con CONSTANT, ed è opportuno definirne il valore.

```
CREATE OR REPLACE FUNCTION somma( a int DEFAULT 0,
                                  b int DEFAULT 0 )
RETURNS INTEGER
AS
$BODY$
DECLARE
      c CONSTANT integer := 10;
BEGIN
        RETURN a + b + c;
END;
$BODY$
LANGUAGE plpgsql;
```

Variabili costanti

Se si cercasse di modificare una variabile costante verrebbe generato un errore di compilazione della funzione:

```
psql:functions.sql:43: ERROR:  "c" is declared CONSTANT
LINE 38:  c := a + b;
          ^
```

Variabili ROWTYPE

Alcune variabili possono essere dichiarate con l'attributo `%rowtype` specificando la tabella a cui fanno riferimento. Le variabili di fatto rappresentano una tupla (strutturata) della tabella dichiarata.

```
DECLARE
  my_record persona%rowtype;
BEGIN
 my_record.nome = 'Luca';
 my_record.eta  = 39;
...-- return next etc.
END;
```

Variaibili RECORD

Le variabili `record` sono la generalizzazione di quelle `rowtype`: possono contenere un record estratto da una qualsiasi tabella e assumeranno la struttura di quella tupla, ma prima del loro assegnamento non possono essere usate.
Le si possono assegnare piu' volte a tuple di struttura differente.

Funzioni che ritornano tuple

Ci sono due modi per dichiarare una funzione che ritorna una o piu' tuple:

- `RETURNS RECORD` restituisce una singola tupla contenuta in un `record` (questo ad esempio è il caso con parametri `OUT`);
- `RETURNS SETOF` restituisce piu' tuple di un tipo specificato (es. di una tabella).

RETURN NEXT e RETURN QUERY

Le funzioni che ritornano un `SETOF` possono usare `RETURN NEXT` per appendere un nuovo record al result set. Una volta conclusa la costruzione del result set si usa un normale `RETURN` per uscire dalla funzione. Alternativamente si può usare `RETURN QUERY` per inserire nel result set il risultato di una query.

RETURN NEXT: esempio

```
CREATE OR REPLACE FUNCTION find_maggiorenni()
RETURNS SETOF persona AS $BODY$
DECLARE
  current_maggiorenne persona%rowtype;
BEGIN
  FOR current_maggiorenne IN
        SELECT * FROM persona WHERE eta >= 18
  LOOP
          RETURN NEXT current_maggiorenne;
  END LOOP;

  RETURN;
END;
$BODY$ LANGUAGE plpgsql;
```

RETURN QUERY

Se si vuole ritornare il risultato di una query si può usare l'operatore `RETURN QUERY`:

```
CREATE OR REPLACE FUNCTION find_maggiorenni()
RETURNS SETOF persona
AS $BODY$
DECLARE
        current_maggiorenne persona%rowtype;
BEGIN
        RETURN QUERY SELECT * FROM persona WHERE eta >= 18;
        RETURN;
END;
$BODY$ LANGUAGE plpgsql;
```

IF...ELSE

```
IF eta >= 18 THEN
     RETURN 'maggiorenne';
ELSIF eta >= 10 THEN
     RETURN 'teen-ager';
ELSE
    RETURN 'bimbo';
END IF;
```

Cicli

I cicli hanno tutti una parola chiave `LOOP` e finiscono con `END LOOP`.
E' possibile specificare una etichetta per identificare il ciclo, l'etichetta deve:

- essere dichiarata prima del ciclo fra doppie parentesi angolari (es. «`CICLO_1`»);
- essere inclusa nella chiusura del ciclo (es. `END LOOP CICLO1`).

LOOP

Ciclo (con eventuale etichetta). Può essere annidato.
Due operatori sintatticamente simili controllano il flusso:

- `CONTINUE` ricomincia l'iterazione dall'etichetta specificata (o dal ciclo corrente) quando la condizione `WHEN` si verifica;
- `EXIT` esce (termina) il ciclo specificato dall'etichetta (o quello corrente) quando la condizione `WHEN` viene soddisfatta.

LOOP: esempio

```
<<LP_ETA>>
  LOOP
     eta := eta + 1;
     EXIT LP_ETA WHEN eta >= 18;
  END LOOP LP_ETA;
```

oppure semplificando:

```
LOOP
   eta := eta + 1;
   EXIT  WHEN eta >= 18;
END LOOP;
```

WHILE

Analogo al ciclo *while* di ogni linguaggio di programmazione, è un `LOOP` con `EXIT` integrata! Può includere una etichetta.

```
<<LP_ETA>>
  WHILE eta <= 18 LOOP
     eta := eta + 1;
  END LOOP LP_ETA;
```

FOR

Analogo al ciclo *for* del linguaggio C, senza la definizione di incremento diventa un *foreach*:

```
<<LP_ETA>>
  FOR current_eta IN 18..30 LOOP
     RAISE INFO 'vALORE %', current_eta;
  END LOOP LP_ETA;
```

o analogamento al ciclo for del linguaggio C:

```
<<LP_ETA>>
  FOR current_eta IN 18..30 BY 2 LOOP
     RAISE INFO 'vALORE %', current_eta;
  END LOOP LP_ETA;
```

FOREACH

Itera fra gli elementi di un array. Segue le stesse regole sintattiche del ciclo FOR (etichetta, ecc).

```
FOREACH current_version IN ARRAY v_array LOOP
   RAISE INFO 'Version %', current_version;
END LOOP;
```

FOR IN query

Quando si deve iterare sul result set di una query la sintassi è `FOR IN <query>`:

```
FOR my_record IN SELECT * FROM persona LOOP
END LOOP;
```

oppure la variante con `EXECUTE` se la query viene specificata da una stringa:

```
query_string := 'SELECT * FROM persona';
FOR my_record IN EXECUTE query_string LOOP
END LOOP;
```

Eccezioni

Il blocco `BEGIN...END` di una funzione permette l'opzionale `EXCEPTION` che funziona da *catch* per le eccezioni. Al suo interno l'eccezione viene cercata fra quelle specificate nelle clausole `WHEN`.
Se nessuna clausola `WHEN` di `EXCEPTION` fa match, o se il blocco `EXCEPTION` non è stato specificato, la funzione termina con errore!
Le eccezioni possibili sono *etichettate* e definite alla pagina `https://www.postgresql.org/docs/9.6/static/errcodes-appendix.html` come ad esempio `division_by_zero`.
Il sistema mantiene lo stato delle variabili al momento in cui si verifica l'errore, l'uso di un blocco `EXCEPTION` è *costoso* per il sistema, quindi non va usato ove non si possono verificare o non interessa catturare eccezioni.

Eccezioni: variabili diagnostiche

All'interno di un blocco `EXCEPTION` le variabili speciali `SQLSTATE` e `SQLERRM` contengono lo stato e il messaggio associato all'errore generato. E' inoltre possibile usare il comando `GET_STACKED_DIAGNOSTIC` per estrarre ulteriori informazioni circa l'errore di esecuzione (es. su quale tabella, colonna, constraint, ecc.).

Eccezioni: esempio

```
BEGIN
    RETURN divide_what / divide_by;
EXCEPTION
    WHEN division_by_zero THEN
      RAISE INFO 'Divisione per zero';
      RAISE INFO 'Errore % %', SQLSTATE, SQLERRM;
      RETURN 0;
END;
```

PERFORM: il problema

Se in una stored procedure si utilizza una query che ritorna dei risultati, ma si vuole scartare l'insieme dei risultati stessi, occorre usare `PERFORM`. In altre parole: nelle stored procedures è possibile usare ogni comando SQL che non ritornino risultati o dove i risultati siano *memorizzati in variabili* (anche di iterazione).

```
CREATE OR REPLACE FUNCTION do_select()
RETURNS VOID AS $BODY$
DECLARE
BEGIN
        SELECT * FROM persona;
END;
$BODY$ LANGUAGE plpgsql;
```

```
> SELECT do_select();
ERROR:  query has no destination for result data
HINT:  If you want to discard the results of a
SELECT, use PERFORM instead.
```

PERFORM: la soluzione

```
CREATE OR REPLACE FUNCTION do_select()
RETURNS VOID
AS $BODY$ DECLARE
BEGIN
        PERFORM  * FROM persona;
END;
$BODY$ LANGUAGE plpgsql;
```

PERFORM sostituisce SELECT in una query!

SELECT INTO

E' possibile assegnare a una (o piu') variabili il risultato di una SELECT specificando appunto la lista di variabile ove inserire (INTO) il risultato, considerando che:

- se la query non ritorna alcuna tupla e non è stato specificato STRICT allora le variabili sono impostate a NULL, se è stato specificato STRICT allora si ha un errore di esecuzione;
- se la query ritorna piu' di una tupla e non è specificato STRICT allora solo la prima tupla viene inserita nelle variabili e la query termina (ossia SELECT INTO esegue un implicito LIMIT 1);
- se la query ritorna un numero di tuple diverso da uno (anche nessuna tupla!) ed è specificato STRICT allora si ha un errore di esecuzione (ossia STRICT impone che la query fornisca una sola tupla).

SELECT INTO vs :=

Lo statement SELECT INTO permette l'assegnamento contemporaneo a piu' variabili (simile a Perl), mentre := tratta un assegnamento alla volta.

```
DECLARE
  y int; m int; d int;
BEGIN
  SELECT extract( year from current_date ),
         extract( month from current_date ),
         extract( day from current_date )
 INTO y, m, d;
 -- y := extract( year from current_date );
 -- m := extract( month from current_date );
 -- d := extract( day from current_date );
END
```

SELECT INTO: esempio senza STRICT

```
CREATE OR REPLACE FUNCTION do_select()
RETURNS VOID AS $BODY$
DECLARE
        v_nome text;
        v_cognome text;
BEGIN
        SELECT nome, cognome
        INTO v_nome, v_cognome
        FROM persona;

        RAISE INFO 'Trovati % %',
                    v_nome,
                    v_cognome;
END;
$BODY$ LANGUAGE plpgsql;
```

SELECT INTO: esempio senza STRICT (2)

```
> SELECT count(*) FROM persona;
 count
-------
     4 -- ci sono 4 persone!
> SELECT do_select();
INFO:  Trovati EMANUELA SANTUNIONE
```

SELECT INTO: errore di STRICT

Se la query viene eseguita con STRICT si ottiene un errore di esecuzione:

```
> SELECT do_select();
ERROR:  query returned more than one row
```

SELECT INTO: esempio con STRICT

```
CREATE OR REPLACE FUNCTION do_select()
RETURNS VOID AS $BODY$
DECLARE
        v_nome text;
        v_cognome text;
BEGIN
        SELECT nome, cognome
        INTO STRICT v_nome, v_cognome
        FROM persona
        WHERE codice_fiscale = 'FRRLCU78L19F257B';

        RAISE INFO 'Trovati % %',
                    v_nome,
                    v_cognome;
END;
$BODY$ LANGUAGE plpgsql;
```

Si filtra sulla chiave per ottenere esattamente un record!

EXECUTE

Il comando `EXECUTE` consente l'esecuzione di query costruite al volo (*no-caching*) in formato stringa. Il comando permette la sotituzione di variabili all'iterno della stringa di query:

- le variabili sono identificate con `$1`, `$2`, ecc.;
- il comando deve avere una forma con `USING` seguito dalla lista dei parametri da sostituire.

I parametri possono essere usati solo per dei dati, per i nomi di tabella/colonna si deve effettuare una concatenazione di stringa!
Inoltre `EXECUTE` permette di effettuare un `INTO [STRICT]` come una normale `SELECT`.

EXECUTE: esempio

```
-- SELECT do_count( 2, 'persona' );
CREATE OR REPLACE FUNCTION do_count( pk int, tabella text)
RETURNS integer
AS $BODY$ DECLARE
        v_count int;
BEGIN
        EXECUTE 'SELECT count(*) FROM '
                 || tabella
                 || ' WHERE pk >= $1'
        INTO STRICT v_count
        USING pk;

        RETURN v_count;
END;
$BODY$ LANGUAGE plpgsql;
```

EXECUTE USING: esempio

```
CREATE OR REPLACE FUNCTION f_tellers( tid int )
RETURNS SETOF pgbench_tellers
AS $BODY$
DECLARE
   r pgbench_tellers%rowtype;
BEGIN
   EXECUTE
    'SELECT * FROM pgbench_tellers '
    || 'WHERE tid = $1'
    USING (tid)
   INTO r;
   RETURN NEXT r;
END;
$BODY$ LANGUAGE plpgsql;
```

plperl (5)

Consente l'esecuzione di Perl 5 come backend per le stored procedures e, di conseguenza, per i trigger.
Non ha accesso diretto al database! Le query vengono eseguite tramite funzioni *SPI (Service Provider Interface)*, analogamente i `RAISE` sono sostituiti da altre funzioni.
Ci sono due versioni del linguaggio:

- `plperl` trusted definibile da ogni utente, esegue il codice in un interprete separato per ogni utente e non ammette l'uso di funzioni che alterino il sistema ospite (es. apertura di file, socket, ecc.);
- `plperlu` *untrusted* definibile solo da utenti amministratori, ammette l'uso di tutte le funzionalità di Perl (inclusa `use`).

Installazione

Occorre installare un pacchetto apposito, ad esempio su FreeBSD:

```
% sudo pkg install postgresql10-plperl-10.1
```

e successivamente nel database abilitare l'estensione:

```
> CREATE EXTENSION plperl;
```

Un semplice esempio

```
CREATE OR REPLACE function perl_count( contains text )
RETURNS int AS $BODY$
   my ( $pattern ) = @_;
   my $tot = 0;
   my $result_set = spi_exec_query( 'SELECT codice_fiscale FROM persona' );
   elog( INFO, "Trovate $result_set->{ processed } tuple" );
   for my $current_row ( @{ $result_set->{ rows } } ) {
       $tot++ if ( $current_row->{ codice_fiscale } =~ /$pattern/ );
   }
   return $tot;
$BODY$
LANGUAGE plperl;
```

Funzioni che ritornano tuple: array reference

```
-- invocazione SELECT * FROM perl_find( 'persona', 'codice_fiscale', 'b' );
CREATE OR REPLACE function perl_find( tab text, col text, contains text )
RETURNS SETOF text
AS $BODY$
   my ( $table, $column, $pattern ) = @_;
   my $result_set = spi_exec_query( "SELECT $column FROM $table" );
   my @found;
   for my $current_row ( @{ $result_set->{ rows } } ) {
       push @found, $current_row->{ $column }
              if ( $current_row->{ $column } =~ /$pattern/ );
   }
   return \@found;
$BODY$
LANGUAGE plperl;
```

Funzioni che ritornano tuple: return_next

```
-- invocazione SELECT * FROM perl_find( 'persona', 'codice_fiscale', 'b' );
CREATE OR REPLACE function perl_find( tab text, col text, contains text )
RETURNS SETOF text
AS $BODY$
   my ( $table, $column, $pattern ) = @_;
   my $result_set = spi_exec_query( "SELECT $column FROM $table" );
   for my $current_row ( @{ $result_set->{ rows } } ){
      return_next( $current_row->{ $column } )
             if ( $current_row->{ $column } =~ /$pattern/ );
   }
   return undef; # IMPORTANTE !!
$BODY$
LANGUAGE plperl;
```

DO

PostgreSQL supporta un costrutto `DO` che permette l'esecuzione di un blocco di codice. Sostanzialmente è una forma per eseguire immediatamente una funzione anonima definita nel comando stesso. Se non si specifica alcun linguaggio, il blocco è considerato `plpgsql`.

DO: esempio in plpgsql

```
> DO $CODE$
 DECLARE
 BEGIN
   FOR i IN 1..10 LOOP
    RAISE INFO 'Valore %', i;
   END LOOP;
 END
 $CODE$;

INFO:  Valore 1
INFO:  Valore 2
INFO:  Valore 3
INFO:  Valore 4
INFO:  Valore 5
INFO:  Valore 6
INFO:  Valore 7
INFO:  Valore 8
INFO:  Valore 9
INFO:  Valore 10
```

DO: esempio in plperl

```
> DO LANGUAGE plperl $CODE$
for ( 1..10 ){
  elog( INFO, $_ );
}
$CODE$;

INFO:  1
INFO:  2
INFO:  3
INFO:  4
INFO:  5
INFO:  6
INFO:  7
INFO:  8
INFO:  9
INFO:  10
```

DO: prototipizzazione rapida

Lo stamente DO può essere usato per testare rapidamente un blocco di codice complesso a piacere da inserire in una funzione:

```
> do $block$
declare cur cursor( ts1 timestamp,
                    ts2 timestamp )
       for select *
           from persona where tsrange( ts1, ts2, '[]' ) @> ts;
r persona%rowtype;
begin open cur( '2018-02-17 12:22:39.858051', '2018-02-17 12:22:39.858053' );
fetch next from cur into r;
while FOUND loop
  raise info 'Record %', r.pk; fetch next from cur into r;
end loop;
end $block$;
```

Cos'è un cursore?

Un cursore è un meccanismo che trasforma un record-set in una iterazione fra record: non si ottiene piu' l'insieme dei risultati in un unico blocco ma si *scorre* il risultato tupla per tupla.
I cursori sono di due tipologie:

- *bound* sono dichiarati con la relativa query a cui fanno riferimento;
- *unbound* sono dichiarati senza la query, che sarà fornita successivamente.

Tutti i cursori sono di tipo `refcursor`, ma solo quelli *unbound* vengono esplicitamente dichiarati come tali. Un cursorse si dice *scrollable* se può scorrere le tuple in avanti e indietro.

Workflow di un cursore

Solitamente il workflow è il seguente:

1. dichiarazione (bound/unbound);
2. `OPEN` indica che si vuole usare il cursore;
3. `FETCH` preleva una tupla dalla query (imposta la variabile `FOUND` di conseguenza);
   - `MOVE`
   - `INSERT` o `UPDATE`
4. `CLOSE` si è finito di usare il cursore, le risorse sono liberate.

Esempio di cursore

```
CREATE OR REPLACE FUNCTION do_cursor()
RETURNS int AS $BODY$
DECLARE
   counter int := 0;
   my_curs CURSOR FOR SELECT * FROM persona;
   my_record persona%rowtype;
BEGIN
    OPEN my_curs;
    FETCH my_curs INTO my_record;
    WHILE FOUND
    LOOP
      IF my_record.eta >= 18
      THEN
         counter := counter + 1;
      END IF;
      FETCH my_curs INTO my_record;
    END LOOP;

     CLOSE my_curs;
     RETURN counter;
END;
$BODY$ LANGUAGE plpgsql;
```

Esempio di cursore parametrico

```
CREATE OR REPLACE FUNCTION do_cursor()
RETURNS int AS $BODY$
DECLARE
        counter int := 0;
        my_curs CURSOR (s_eta int)
             FOR SELECT * FROM persona
                 WHERE eta >= s_eta;
        my_record persona%rowtype;
BEGIN
        OPEN my_curs( 18 );
        FETCH my_curs INTO my_record;
        WHILE FOUND
        LOOP
          counter := counter + 1;
          FETCH my_curs INTO my_record;
        END LOOP;

     CLOSE my_curs;
     RETURN counter;
END; $BODY$ LANGUAGE plpgsql;
```

Esempio di cursore unbound

```
CREATE OR REPLACE FUNCTION do_cursor()
RETURNS int AS $BODY$
DECLARE
        counter int := 0;
        my_curs refcursor;
        my_record persona%rowtype;
BEGIN
        OPEN my_curs FOR
            SELECT * FROM persona
            WHERE eta >= 18;
        FETCH my_curs INTO my_record;
        WHILE FOUND
        LOOP
          counter := counter + 1;
          FETCH my_curs INTO my_record;
        END LOOP;

     CLOSE my_curs;
     RETURN counter;
END;
$BODY$ LANGUAGE plpgsql;
```

Ciclo "automatico" su cursore

Invece che il workflow `OPEN`, `FETCH`, `LOOP`, `CLOSE` si può usare il ciclo `FOR` che riduce le operazioni necessaire al solo `LOOP`.
Con `FOR` non si usa `OPEN`, `CLOSE`, `FETCH`!

Esempio di ciclo "automatico" su cursore

```
CREATE OR REPLACE FUNCTION do_cursor()
RETURNS int AS $BODY$
DECLARE
        counter int := 0;
        my_curs CURSOR FOR
          SELECT * FROM persona
          WHERE eta >= 18;
        my_record persona%rowtype;
BEGIN
   FOR my_record IN my_curs
   LOOP
     counter := counter + 1;
   END LOOP;
  RETURN counter;
END; $BODY$ LANGUAGE plpgsql;
```

Capire FETCH

L'istruzione `FETCH` permette di specificare la direzione e il posizionamento assoluto o relativo. La direzione viene specificata con `FORWARD` (default) o `BACKWARD`. Il posizionamento viene specificato con `FIRST`, `LAST`, `ABSOLUTE n` e `RELATIVE n`.

Esempio di spostamento con FETCH: lettura all'indietro

```
OPEN my_curs;
FETCH LAST FROM my_curs INTO my_record;
WHILE FOUND
LOOP
   IF my_record.eta >= 18 THEN
     counter := counter + 1;
   END IF;
   FETCH BACKWARD FROM my_curs INTO my_record;
END LOOP;
CLOSE my_curs;
```

Esempio di spostamento con FETCH: lettura all'indietro ogni due record

```
OPEN my_curs;
FETCH LAST FROM my_curs INTO my_record;
WHILE FOUND
LOOP
   IF my_record.eta >= 18 THEN
     counter := counter + 1;
   END IF;
   FETCH RELATIVE -2 FROM my_curs INTO my_record;
END LOOP;
CLOSE my_curs;
```

UPDATE/DELETE FOR CURRENT

E' possibile eseguire un aggiornamento/cancellazione di una tupla su cui è posizionato il cursore. La sintassi è `UPDATE <table> SET <cols> WHERE CURRENT OF <cursor>` o nel caso di una cancellazione `DELETE FROM <table> WHERE CURRENT OF <cursor>`.

```
OPEN my_curs;
 FETCH LAST FROM my_curs INTO my_record;
 WHILE FOUND
 LOOP
    IF my_record.eta >= 18 THEN
      counter := counter + 1;
    ELSE
      UPDATE persona set valid = false
           WHERE CURRENT OF my_curs;
    END IF;
    FETCH RELATIVE -2 FROM my_curs INTO my_record;
 END LOOP;
 CLOSE my_curs;
```

Multicursori

E' possibile definire una funzione che ritorni piu' di un cursorse alla volta. In questo caso ad ogni cursore viene associato un nome, e all'interno della stessa transazione è possibile recuperare i dati di un cursorse.

Multicursori: esempio di funzione

```
> CREATE OR REPLACE FUNCTION f_query_more_than_one()
  RETURNS SETOF refcursor
  AS $BODY$
      DECLARE
          persona_curs refcursor;
          evento_curs  refcursor;
      BEGIN
         OPEN persona_curs FOR SELECT * FROM persona;
         RETURN NEXT persona_curs;

         OPEN evento_curs FOR SELECT * FROM evento;
         RETURN NEXT evento_curs;

      RETURN;
  END;
  $BODY$
  LANGUAGE plpgsql;
```

Multicursori: esempio di utilizzo

```
> BEGIN;
> SELECT f_query_more_than_one();
 f_query_more_than_one
-----------------------
 <unnamed portal 5>
 <unnamed portal 6>

> FETCH NEXT FROM "<unnamed portal 5>";
  pk   | nome | cognome |  codice_fiscale  | sesso | eta
-------+------+---------+------------------+-------+-----
 12007 | Luca | Ferrari | FRRLCU78L19F257B | M     |   1

> FETCH NEXT FROM "<unnamed portal 6>";
   pk    |  description
---------+---------------
 1017695 |  Evento 15695
```

Comunicazione fra sessioni (IPC)

PostgreSQL prevede un meccanismo di comunicazione a *canali* definito tramite due primitive:

- `LISTEN` accetta messaggi su un determinato canale;
- `NOTIFY` invia un messaggio su un determinato canale.

Ogni messaggio dispone di un eventuale *payload*, ovvero di un corpo. L'interazione è asincrona, è compito del client applicativo interrogare il sistema di notifica. Questo significa, ad esempio, che il terminale `psql` non visualizzerà immediatamente le notifiche ma solo all'esecuzione di un nuovo comando.

Primo esempio di LISTEN e NOTIFY

Il nome del canale è arbitrario, si deve scegliere opportunamente per evitare sovrapposizioni. Si ricevono gli eventi notificati solo dopo una listen avvenuta. In una sessione:

```
-- due
> NOTIFY my_channel, 'First event';
> NOTIFY my_channel, 'Second event';
```

e in un'altra sessione, quando viene eseguito un qualche statement:

```
-- uno
> LISTEN my_channel;
...
-- tre
> SELECT 1 + 1;
...
Asynchronous notification "my_channel"
    with payload "First event" received
    from server process with PID 969.
Asynchronous notification "my_channel"
    with payload "Second event" received
    from server process with PID 969.
```

Funzioni di supporto

Il sistema di notifica usa in realtà una coda, e la funzione `pg_notify()` può essere usata per notificare un determinato canale con un determinato payload.
Non esiste una funzione di ricezione, essendo quella a carico dell'interfaccia client (ad esempio `DBI` di Perl usa un metodo `pg_notifies`).

Esempio di NOTIFY tramite rule

```
> CREATE OR REPLACE RULE d_evento
AS ON DELETE TO evento
DO ALSO
SELECT pg_notify( 'delete_channel',
                  'Eliminato evento ' || OLD.pk );
```

e quando si effettua una cancellazione si ottiene, in un'altra sessione:

```
Asynchronous notification "delete_channel"
    with payload "Eliminato evento 1014713" received
    from server process with PID 969
```

Introduzione ai triggers

PostgreSQL supporta (da sempre) i trigger a livello di *statement* o *tupla* sia per esecuzione prima (before) o dopo (after). Sulle viste agiscono i trigger *INSTEAD OF*.
In PostgreSQL i trigger vengono definiti con un workflow ben definito:

1. definire una funzione (tipicamente in `plpgsql`) che non accetta alcun argomento e ritorna un valore di tipo `trigger`;
2. si *aggancia* la funzione ad un evento scatenato su una tabella (tramite `CREATE TRIGGER`).

Tipologia di trigger

I trigger possono essere eseguiti per `statement` o per `riga`. Nel caso di esecuzione a livello di statement il valore di ritorno del trigger è ignorato (`NULL`), nel caso di esecuzione per riga il valore di ritorno può essere `NULL` per indicare che l'oeprazione deve abortire o ritornare una tupla (`NEW`, `OLD`) per inserimento/modifica.

Tiplogia di trigger (riassunto)

Statement SQL	Quando	A livello di
`INSERT, UPDATE, DELETE`	*BEFORE*	tupla su tabelle e FK
	AFTER	tupla su tabelle e FK
`INSERT, UPDATE, DELETE`	*BEFORE*	statement su tabelle (anche este
	AFTER	statement su tabelle (anche este
`TRUNCATE`	*BEFORE*	statement su tabelle
	AFTER	statement su tabelle
`INSERT, UPDATE, DELETE`	*INSTEAD OF*	tupla su vista

Esecuzione dei trigger

I trigger sono eseguiti in ordine alfabetico (per tipologia di evento).
I trigger `BEFORE` vengono eseguiti prima del controllo delle constraint, analogamente i trigger `AFTER` dopo l'esecuzione e quindi dopo il controllo delle constraint.
I trigger per operazioni che *consolidano* una tupla (`INSERT`, `UPDATE`) vedono sempre la tupla di uscita `NEW` (quella che verrà *memorizzata*) e se sono per update anche quella di ingresso `OLD` (quella che si sta aggiornando). Se il trigger elimina una tupla (non c'è tupla di uscita) si vede solo la tupla di ingresso `OLD` (caso `DELETE`).
Il tipo di operazione che ha fatto scattare il trigger è disponibile tramite la stringa `TR_OP`, il numero di argomenti del trigger tramite `TG_NARGS` e i singoli argomenti nell'array di stringhe `TG_ARGV`.

Esempio di trigger: descrizione

Si vuole creare un semplice trigger che, al momento di inserimento o aggiornamento di una tupla della tabella `persona`, controlli che l'età inserita sia coerente con quella del relativo codice fiscale e la "auto-corregga" calcolandola dall'anno attuale.

Esempio di trigger: funzione

```
CREATE OR REPLACE FUNCTION tr_check_eta()
RETURNS TRIGGER
AS $BODY$
DECLARE
        current_eta integer;
        current_year integer;
BEGIN
  IF NEW.codice_fiscale IS NOT NULL THEN
       current_eta  = to_number(
                       substring( NEW.codice_fiscale FROM 7 FOR 2 ),
                      '99' );
       current_year = to_number( to_char( current_date, 'YY' ),
                      '99' );
       IF current_year < current_eta THEN
          current_eta = current_year + 100 - current_eta;
       END IF;
  END IF;
```

Esempio di trigger: funzione (2)

```
  IF TG_OP = 'INSERT' THEN
       NEW.eta = current_eta;
       RAISE INFO 'Calcolo eta'' INSERT = %', current_eta;
       RETURN NEW;
   END IF;

  IF TG_OP = 'UPDATE' THEN
     IF OLD.eta > NEW.eta OR NEW.eta <= 0 THEN
        RAISE INFO 'Aggiusto eta'' UPDATE da % a %', OLD.eta, current_eta;
        NEW.eta = current_eta;
     END IF;
  END IF;

END;
$BODY$
LANGUAGE plpgsql VOLATILE;
```

Esempio di trigger: aggancio alla tabella

```
> CREATE TRIGGER tr_persona_eta
  BEFORE INSERT OR UPDATE
  ON persona
  FOR EACH ROW
  EXECUTE PROCEDURE tr_check_eta();
```

Esempio di trigger: esecuzione

```
> INSERT INTO persona( nome, cognome, codice_fiscale )
VALUES( 'Luca', 'Ferrari', 'FRRLCU78L19F257B' );
INFO:  Calcolo eta' INSERT = 39

> UPDATE persona SET eta = 10 WHERE codice_fiscale = 'FRRLCU78L19F257B';
INFO:  Aggiusto eta' UPDATE da 39 a 39
```

Esempio di trigger: agganciarlo a una colonna

Il trigger di esempio non dovrebbe scattare per ogni `UPDATE` incondizionato, ma solo per gli `UPDATE` che coinvolgono la colonna `eta` o `codice_fiscale`. E' possibile specificare a quali colonne il trigger deve essere agganciato:

```
>  DROP TRIGGER tr_persona_eta ON persona;
> CREATE TRIGGER tr_persona_eta
  BEFORE INSERT OR UPDATE OF eta, codice_fiscale
  ON persona
  FOR EACH ROW
  EXECUTE PROCEDURE tr_check_eta();
```

Esempio di trigger: aggancio a una colonna (2)

Il trigger scatta ora solo per gli aggiornamenti delle colonne specificate:

```
> UPDATE persona SET eta = 10 WHERE codice_fiscale = 'FRRLCU78L19F257B';
INFO:  Aggiusto eta' UPDATE da 39 a 39

> UPDATE persona SET nome = 'LUCA' WHERE codice_fiscale = 'FRRLCU78L19F257B';
-- non è scattato il trigger!
```

Trigger parametrici

E' possibile passare dei parametri alla funzione di un trigger, tali parametri saranno visibili tramite l'array `TG_ARGV`. I parametri sono letterali e convertiti sempre a stringa.

Esempio di trigger parametrico: funzione con parametro

La funzione dell'esempio precedente può essere modificata per accettare l'età da usare come default:

```
CREATE OR REPLACE FUNCTION tr_check_eta()
RETURNS TRIGGER
AS $BODY$
DECLARE
        current_eta integer;
        current_year integer;
BEGIN
        IF TG_NARGS > 0 THEN
           current_eta := to_number( TG_ARGV[ 0 ], '99' );
        ELSE
           current_eta := -1;
        END IF;
...
```

Esempio di trigger parametrico: aggancio

```
> CREATE TRIGGER tr_persona_eta
  BEFORE INSERT OR UPDATE OF eta, codice_fiscale
  ON persona
  FOR EACH ROW EXECUTE PROCEDURE tr_check_eta( 10 );

> UPDATE persona SET eta = 10, codice_fiscale = NULL
  WHERE codice_fiscale = 'FRRLCU78L19F257B';
INFO:  Aggiusto eta' UPDATE da 39 a 10
```

Vedere i trigger

I trigger attivi su una tabella possono essere visti con il comando \d in psqlì, oppure interrogando il catalogo di sistema ~pg_trigger e la funzione speciale pg_get_trigger_def():

```
> SELECT tgname, pg_get_triggerdef(oid, false)
    FROM pg_trigger
    WHERE tgname = 't_in_up';

tgname             | t_in_up
pg_get_triggerdef | CREATE TRIGGER t_in_up
                           BEFORE INSERT OR UPDATE
                           ON public.events
                           FOR EACH ROW
                           EXECUTE PROCEDURE f_tr_ins_up()
```

Introduzione ai trigger DDL

PostgreSQL permette di definire dei trigger per *eventi* di DDL, ovvero `CREATE`, `DROP`, `ALTER`, `COMMENT`, `GRANT`, `REVOKE`. Gli istanti in cui si possono intercettare gli eventi sono:

- `ddl_command_start`: appena il comando viene avviato, senza alcun controllo sull'esistenza dell'oggetto a cui si applica;
- `ddl_command_end`: appena il comando è finito (quindi i cataloghi di sistema sono stati aggiornati ma non ancora consolidati);
- `table_rewrite`: si effettua un `ALTER [TABLE | TYPE]` su una tabella;
- `sql_drop`: il comando `DROP` ha rimosso dal catalogo di sistema degli oggetti.

Bisogna essere superutenti!

Creazione di un event trigger

Le regole sono simili a quelle di un trigger normale:

1. si crea una funzione che deve ritornare un tipo `event_trigger`. Il valore di ritorno è solo un marcaposto, questi trigger non ritornano alcun valore!
2. si aggancia il trigger all'evento con `CREATE EVENT TRIGGER`;
3. si possono usare alcune funzioni di utilità per scoprire *cosa* sta accadendo e cosa ha *scatenato* il trigger (non esistono al momento variabili speciali da interrogare ma solo funzioni di catalogo).

Dati interni di un event trigger

Ci sono delle funzioni speciali che ritornano delle tuple contenenti le informazioni di come un trigger è stato invocato:

- `pg_event_trigger_ddl_commands()` per `ddl_command_start` e `ddl_command_stop`;
- `pg_event_trigger_dropped_objects()` per `sql_drop`;
- `pg_event_trigger_table_rewrite_oid()` per `table_rewrite`.

Esempio di event trigger

```
CREATE OR REPLACE FUNCTION ddl_start_end()
RETURNS EVENT_TRIGGER AS
$BODY$
DECLARE
        command record;
BEGIN
   RAISE INFO 'event trigger fired!';
   FOR command IN SELECT * FROM pg_event_trigger_ddl_commands()
   LOOP
           RAISE INFO 'Comando % su %',
                 command.command_tag,
                 command.object_identity;
   END LOOP;
END; $BODY$
LANGUAGE plpgsql;
```

Esempio di esecuzione con event trigger

```
# CREATE EVENT TRIGGER tr_start_end
  ON ddl_command_start
  EXECUTE PROCEDURE ddl_start_end();

# ALTER TABLE testddl ADD COLUMN foo int;
INFO:  event trigger fired!
INFO:  Comando ALTER TABLE su public.testddl
ALTER TABLE
```

Esempio di even trigger (drop)

```
CREATE OR REPLACE FUNCTION ddl_drop_fn()
RETURNS EVENT_TRIGGER AS $BODY$
DECLARE
        dropping record;
BEGIN
        RAISE INFO 'event trigger fired!';
        FOR dropping IN SELECT * FROM pg_event_trigger_dropped_objects()
        LOOP
                RAISE INFO 'drop % di %',
                      tg_tag, -- cosa si droppa?
                      dropping.object_identity;
        END LOOP;
END; $BODY$ LANGUAGE plpgsql;
```

Esecuzione di event trigger (drop)

```
# CREATE EVENT TRIGGER tr_drop
  ON sql_drop
  EXECUTE PROCEDURE ddl_drop_fn();
```

Esecuzione di event trigger (drop)

```
# DROP TABLE testddl;
INFO:  event trigger fired! -- ddl_command_start
INFO:  drop DROP TABLE di public.testddl
INFO:  drop DROP TABLE di testddl_pkey on public.testddl
INFO:  drop DROP TABLE di public.testddl_pkey
INFO:  drop DROP TABLE di public.testddl_pk_seq
INFO:  drop DROP TABLE di public.testddl_pk_seq
INFO:  drop DROP TABLE di for public.testddl.pk
INFO:  drop DROP TABLE di public.testddl
INFO:  drop DROP TABLE di public.testddl[]
INFO:  drop DROP TABLE di pg_toast.pg_toast_16508
INFO:  drop DROP TABLE di pg_toast.pg_toast_16508_index
INFO:  drop DROP TABLE di pg_toast.pg_toast_16508
INFO:  event trigger fired! -- ddl_command_end
```

PostgreSQL distingue due tipi principali di `GRANT` e conseguentemente di `REVOKE`:

- su un oggetto di database (es. tabella), nel qual caso anche a livello di colonna;
- su un ruolo (ovvero gestione dei gruppi).

La parola `PUBLIC` indica tutti gli utenti, mentre la parola chiave `ALL` indica tutti i tipi di permessi. Per agire a livello globale (*pericoloso!*):

```
> REVOKE ALL ON <oggetto> FROM PUBLIC;
> GRANT ALL ON <oggetto> TO PUBLIC;
```

Il proprietario di un oggetto ha per definizione tutti i permessi sull'oggetto che ha creato.
Chi possiede il permesso `GRANT` può concedere lo stesso permesso ad altri ruoli: si può quindi non solo abitare agli oggetti, ma anche alle abilitazioni stesse verso altri utenti. PostgreSQL supporta un tipo particolare di `GRANT`: fra ruoli. Una `GRANT` fra ruoli compone un gruppo fra i ruoli, così facendo i permessi dell'utente destinazione vengono ereditati da quelli dell'utente di partenza.

```
# GRANT luca TO dev1;
-- dev1 ora appartiene al gruppo luca
```

```
# REVOKE ALL ON software FROM PUBLIC;
# GRANT SELECT(name) ON software TO luca;
```

e come utente `luca`:

```
> SELECT name FROM software; --OK!
> SELECT name, versions FROM software;
ERROR:  permission denied for relation software
```

Esistono comandi appositi per *passare* un oggetto da un ruolo ad un altro e per eliminare tutti gli oggetti posseduti da un determinato ruolo. La parola chiave `CURRENT_USER` identifica l'utente corrente.

```
> REASSIGN OWNED BY CURRENT_USER TO dba;
-- drop altri oggetti
> DROP OWNED BY dismissed;
```

Tipicamente si usano questi comandi prima della cancellazione di un ruolo!

E' possibile specificare, tabella per tabella, una sicurezza a livello di tupla, denominata *Row Level Security*.
La Row Level Security si basa su delle *policy* che devono discriminare quali dati mostrare/nascondere. Se nessuna policy viene creata si usa un default di *deny all*.
ATTENZIONE: è bene concedere i permessi di riga solo dopo aver assegnato i permessi `GRANT` normali!

```
> CREATE POLICY view_maggiorenni
  ON persona
  FOR SELECT  -- quale statement?
  TO PUBLIC   -- quale ruolo ?
  USING  (eta >= 18); -- condizione di SELECT

> ALTER TABLE persona ENABLE ROW LEVEL SECURITY;
```

e come altro utente si vedranno solo le tuple che soddisfano USING.
Anzitutto l'utente che effettua lo statement deve avere le opportune GRANT. Il proprietario dell'oggetto non è soggetto alle policy.
Nel caso di statement SELECT la condizione è data da USING, nel caso di INSERT o UPDATE da CHECK, e si può combinare tutto quanto assieme:

```
> CREATE POLICY handle_maggiorenni
  ON persona
  FOR ALL                   -- SELECT, UPDATE, DELETE, INSERT
  TO PUBLIC                 -- quale ruolo ?
  USING  (eta >= 18)        -- condizione di SELECT
  WITH CHECK (eta >= 18);   -- condizione DML
```

Tipicamente questo meccanismo viene usato per nascondere le tuple di altri utenti:

```
> CREATE POLICY handle_my_tuples
  ON usernames
```

```
FOR ALL                                    -- SELECT, UPDATE, DELETE, INSERT
TO PUBLIC                                  -- quale ruolo ?
USING  (usernam = CURRENT_USER)            -- condizione di SELECT
WITH CHECK (usernam = CURRENT_USER);       -- condizione DML
```

Access Control List

PostgreSQL memorizza i permessi dati con `GRANT` sono memorizzati come *Access Contol List (ACL)*.
Ogni ACL (`aclitem`) è formata da tre parti:

- utente a cui i privilegi si riferiscono;
- stringa dei permessi (simile a quella dei file Unix);
- utente che ha concesso il permesso.

Le ACL sono memorizzate come array `aclitem[]` in `pg_class`.

Il comando dp

Il comando speciale \dp di psql fornisce la visualizzazione dei permessi come stringa *ACL*. Ogni lettera nella stringa ACL si riferisce ad un permesso specifico:

```
r -- SELECT ("read")          t -- TRIGGER
w -- UPDATE ("write")         X -- EXECUTE
a -- INSERT ("append")        U -- USAGE
d -- DELETE                   C -- CREATE
D -- TRUNCATE                 c -- CONNECT
x -- REFERENCES               T -- TEMPORARY
          arwdDxt -- ALL PRIVILEGES
```

Vedere tutti i permessi (\dp): esempio

```
> \dp soci

Access privileges | conoscerelinux=arwdDxt/conoscerelinux
Column privileges | nome:                                          +
                  |   enrico=r/conoscerelinux                      +
                  | data_associazione:                             +
                  |   enrico=w/conoscerelinux
Policies          | mypol:                                         +
                  |   (u): (data_associazione IS NOT NULL)         +
                  |   (c): (lower(nome) = (CURRENT_USER)::text)+
                  |   to: enrico
```

Decodificare a mano i permessi (con CTE e string manipulation)

```
> WITH acl AS (
    -- 1) si estraggono le ACL come tuple di stringhe
    SELECT unnest( relacl::text[] ) AS acl
    FROM pg_class
    WHERE relname = 'soci'
    AND relkind = 'r'
)
```

Decodificare a mano i permessi (con CTE e string manipulation) (2)

```
,split_acl AS (
  -- 2) si estraggono i singoli pezzi della ACL
  -- separando dove vi sono i simboli '=' e '/'
   SELECT acl,
    position( '=' in acl ) AS equal_at,
    substring( acl from 1
              for position( '=' in acl ) - 1 )
              AS granted_to,
    substring( acl from
              position( '=' in acl ) + 1
              for position( '/' in acl )
                  - position( '=' in acl ) - 1 )
              AS granted_what,
    substring( acl from
                 position( '/' in acl ) + 1 )
              AS grantee
   FROM acl
)
```

Decodificare a mano i permessi (con CTE e string manipulation) (3)

```
 , decode_acl AS (
   SELECT CASE
          WHEN position( 'r' in granted_what ) > 0
          THEN 'SELECT' END
         ,CASE
          WHEN position( 'd' in granted_what ) > 0
          THEN 'DELETE' END
         ,CASE
          WHEN position( 'D' in granted_what ) > 0
          THEN 'TRUNCATE' END
          -- e così via
   , * FROM split_acl
)

-- query finale
SELECT * FROM decode_acl;
```

Tipologie di viste

Esistono due tipi principali di viste:

- *viste dinamiche*: effettuano la query al momento propagandola alle tabelle di definizione della vista;
- *materializzate*: contengono una versione memorizzata dei dati prelevati dalle tabelle sottostanti e necessitano di essere aggiornata (refresh).

Viste dinamiche

Si definisce la vista attraverso una query (che può includere join, order-by, ecc):

```
> CREATE VIEW vw_persona
  AS
  SELECT upper( cognome ), nome, eta
  FROM persona
  ORDER BY cognome, nome;

> SELECT * FROM vw_persona;
```

Viste dinamiche aggiornabili

Una vista viene creata come automaticamente aggiornabile se:

- effettua una SELECT da una sola tabella;
- non include clausole DISTINCT, funzioni di aggregazione o window function, LIMIT e altre clausole che rendano il result set "non lineare".

```
> CREATE VIEW vw_hello AS SELECT 'Hello';
> INSERT INTO vw_hello VALUES( 'World!' );
ERROR:  cannot insert into view "vw_hello"
DETAIL:  Views that do not select from a single
         table or view are not automatically updatable.
HINT:  To enable inserting into the view,
       provide an INSTEAD OF INSERT trigger
       or an unconditional ON INSERT DO INSTEAD rule.
```

Viste aggiornabili: WITH CHECK OPTION

Le viste aggiornabili possono specificare la clausola `WITH CHECK OPTION` che limita l'aggiornamento alle sole tuple presenti nel result set della vista stessa, e non della tabella sottostante. In particolare `WITH CHECK OPTION` può essere:

- `LOCAL` si considera la definizione solo della vista;
- `CASCADED` si considera la definizione di eventuali sottoviste.

Viste aggiornabili: esempio WITH CHECK OPTION

```
> CREATE TABLE numbers( value int DEFAULT 0 );

> CREATE VIEW vw_positives AS
   SELECT value FROM numbers
   WHERE value > 0;

> CREATE VIEW vw_positives AS
   SELECT value FROM vw_real
   WITH CASCADED CHECK OPTION;
```

Viste aggiornabili: esempio WITH CHECK OPTION (2)

```
-- KO: local check option
> INSERT INTO vw_real( value ) VALUES( -1 );
ERROR:  new row violates check option for view "vw_real"
DETAIL:  Failing row contains (-1).

-- KO: cascaded check option
> INSERT INTO vw_positives( value ) VALUES( -1 );
ERROR:  new row violates check option for view "vw_real"
DETAIL:  Failing row contains (-1).

-- OK: cascaded check option
> INSERT INTO vw_positives( value ) VALUES( 0 );
```

Viste materializzate

Si utilizza il comando `CREATE MATERIALIZED VIEW` specificando se prelevare subito i dati (`WITH DATA`) o no (`WITH NO DATA`). In quest'ultimo caso la vista deve prima essere sottoposta a `REFRESH` per essere utilizzata.

```
> CREATE MATERIALIZED VIEW vw_m_persona
  AS SELECT upper( cognome ), nome
  FROM persona
  ORDER BY cognome, eta
  WITH NO DATA;

 > SELECT * FROM vw_m_persona;
ERROR:  materialized view "vw_m_persona"
        has not been populated
```

Popolare e fare refresh di una vista materializzata

Il comando `REFRESH MATERIALIZED VIEW` viene usato per popolare la vista materializzata. Sostanzialmente il comando effettua:

- un `TRUNCATE` della vista (quindi i dati precedenti sono persi);
- un *lock* della vista (che quindi non può essere usata fino a che il refresh non è terminato;
- una esecuzione della query di definizione della vista per popolarla con i dati.

Per evitare il *lock* si può usare `CONCURRENTLY`, così che la vista possa essere letta mentre viene popolata. Se si specifica `WITH NO DATA` la query non viene eseguita e la vista ritorna ad uno stato non usabile fino al prossimo refresh.

Popolare (e ripopolare) la vista materializzata

```
> REFRESH MATERIALIZED VIEW vw_m_persona;
-- equivalente a
> REFRESH MATERIALIZED VIEW vw_m_persona WITH DATA;
```

Popolare una vista materializzata senza lock

Per usare `CONCURRENTLY` nel refresh si deve avere un indice *unique* su almeno una colonna della vista, e non lo si può usare per il primo popolamento della vista.

```
> CREATE UNIQUE INDEX idx_vw_m_persona_nome
  ON vw_m_persona( nome );

> REFRESH MATERIALIZED
  VIEW CONCURRENTLY vw_m_persona
  WITH DATA;
```

Confondere l'ottimizzatore

Le viste possono *confondere* l'ottimizzatore se usate in modo improprio:

```
> CREATE VIEW vw_persona
  AS
  SELECT * FROM persona
  ORDER BY cognome DESC; -- ATTENZIONE!

> EXPLAIN SELECT * FROM vw_persona
          ORDER BY cognome ASC; -- ATTENZIONE !
                                QUERY PLAN
-------------------------------------------------------------------------
 Sort  (cost=25947.64..26197.64 rows=100000 width=44)
   Sort Key: persona.cognome
   ->  Sort  (cost=13316.32..13566.32 rows=100000 width=44)
         Sort Key: persona.cognome DESC
         ->  Seq Scan on persona  (cost=0.00..1935.00 rows=100000 width=44)
```

Viene eseguito un doppio *sorting* dei dati. E' bene non includere il sort nelle viste per evitare simili problemi!

Introduzione alle rules

Le rules sono definite anche query rewrite system. Le rules consentono la riscrittura al volo delle query (anticipando anche i trigger) e quindi di rimbalzare una query da un oggetto ad un altro o espandere una query in piu' di una.
A differenza di un trigger, una rule può intercettare una `SELECT`!

Parametri di una rule

Una rule deve specificare:

- un *evento* ovvero `SELECT`, `INSERT`, `UPDATE`, `DELETE`;
- la tabella a cui l'evento si applica (ossia su quale tabella il comando è stato eseguito);
- una eventuale *condizione* che può referenziare `OLD` e `NEW` come alias delle tabelle originali e verso cui rimbalzare la query;
- il *comando* da eseguire come `INSTEAD` o `ALSO`.

Le viste sono realizzate come una rule `INSTEAD`!

Esempio di una rule: DELETE

```
> CREATE OR REPLACE RULE r_delete_persona
  AS ON DELETE TO persona
  DO INSTEAD
    UPDATE persona
    SET valid = false
    WHERE OLD.pk = pk;
```

Esempio di rule: INSERT

```
> CREATE OR REPLACE RULE r_insert_persona
  AS ON INSERT
  TO persona
  DO ALSO
   INSERT INTO log( message )
   VALUES ( 'Inserimento/aggiornamento tupla '
            || NEW.cognome || ' '
            || NEW.nome );
```

Rules "RETURN"

La rule per una select è speciale e viene denominata "_RETURN" (ce ne può essere solo una) e deve essere eseguita su una tabella vuota. E' il modello con il quale PostgreSQL realizza le viste dinamiche.
Il workflow è:

1. creare una tabella vuota con le stesse colonne della tabella finale;
2. definire la rule "_RETURN" con DO INSTEAD e SELECT sulla tabella corretta.

Nel momento in cui "_RETURN" viene definita la tabella viene catalogata come vista!*

Esempio di rule: "RETURN"

```
-- CREATE TABLE maggiorenni() INHERITS( persona );
> CREATE TABLE maggiorenni( LIKE persona );
> CREATE OR REPLACE RULE "_RETURN"
  AS ON SELECT TO maggiorenni
  DO INSTEAD
    SELECT * FROM persona
    WHERE eta >= 18;
```

Esempio di rule: UPDATE

E' possibile *bloccare* una tabella con una rule `DO NOTHING`.

```
> CREATE OR REPLACE RULE r_update_persona
  AS ON update
  TO persona
  DO INSTEAD NOTHING;
```

Cosa è una window function?

Le *Window Functions* sono funzioni che effettuano un calcolo su tuple correlate a quella corrente, qualcosa idealmente simile ad una funzione di aggregazione ma senza la necessità di aggregare le tuple. Inoltre una window function può essere eseguita su una partizione delle tuple, non su tutte le tuple di una query.

OVER & PARTITION

Una window function è solitamente seguita da una clausola `OVER` che identifica su quale sottoinsieme di dati effettuare il calcolo. All'interno della clausola `OVER` si può specificare:

- `PARTITION BY` per indicare quali colonne usare per correlare le tuple (concettualmente simile a *group by*);
- `ORDER BY` per specificare l'ordinamento per processare le tuple nella funzione di correlazione.

Comprendere la *window*

La window è calcolata dinamicamente sul result set con riferimento alla tuple sulla quale ci si trova:

```
> SELECT v,
         first_value(v) OVER (ORDER BY v DESC),
         last_value(v) OVER (ORDER BY v DESC),
         array_agg(v) OVER (ORDER BY v DESC)
  FROM generate_series( 1, 4 ) v;

 v | first_value | last_value | array_agg
---+-------------+------------+-----------
 4 |           4 |          4 | {4}
 3 |           4 |          3 | {4,3}
 2 |           4 |          2 | {4,3,2}
 1 |           4 |          1 | {4,3,2,1}
```

Esempio di base per le Window Functions

```
> CREATE TABLE software( pk SERIAL PRIMARY KEY,
   name text,
   version numeric,
   UNIQUE( name, version ),
   CHECK( version > 0 ) );

-- inserimento di un po' di valori...
```

Esempio di Window Function: rank

La funzione `rank()` fornisce una classifica dei dati. Se non si specifica un `PARTITION BY` la funzione non sa come correlare la tupla corrente, quindi è come correlarla a sé stessa:

```
> SELECT name, version,
         rank() OVER ()
  FROM software
  ORDER BY name, version;
 name | version | rank
------+---------+------
 Java |     1.4 |    1
 Java |     1.5 |    1
 Java |     1.6 |    1
 Java |     1.7 |    1
 Java |     1.8 |    1
 Perl |     5.1 |    1
 Perl |    5.20 |    1
 Perl |     5.4 |    1
```

Esempio di Window Function: rank (2)

```
> SELECT name, version,
  rank() OVER ( PARTITION BY name ORDER BY version)
  FROM software
  ORDER BY name, version;
 name | version | rank
------+---------+------
 Java |     1.4 |    1
 Java |     1.5 |    2
 Java |     1.6 |    3
 Java |     1.7 |    4
 Java |     1.8 |    5
 Perl |     5.1 |    1
 Perl |    5.20 |    2
 Perl |     5.4 |    3
```

Alcune Window Functions comuni

- `row_number()` il numero di riga nella *window* (partizione);
- `cume_dist()` numero di righe precedenti divise per il numero totali di righe della partizione;
- `ntile()` accetta un intero e cerca di dividere la *window* in un numero di *bucket* bilanciati, specificando ogni riga a quele bucket appartiene;
- `lag()` accetta una colonna, un offset e un valore di default. Calcola la colonna *offset* righe prima o se non esistono, al valore specificato come default;
- `lead()` opposto di `lag()` (calcola sulle righe in avanti);
- `first_value()`, `last_value()`, `nth_value()` accettano una colonna e restituiscono il valore della tupla all'offset specificato rispettivamente come prima, ultima ed ennesima.

Simulare un contatore di riga

Per simulare un contatore di riga (es. `ROW_NUMBER`) si può usare una query come la seguente:

```
> SELECT
   row_number() OVER () AS ROW_ID ,
   name
  FROM family_tree
```

Tabella di esempio per capire le window functions

```
> CREATE TABLE score(
    pk int GENERATED ALWAYS AS IDENTITY,
    name text,
    score int,
    UNIQUE( name ),
    PRIMARY KEY( pk ) );

> INSERT INTO score( name, score) VALUES
  ( 'Luca', 100 ), ( 'Diego', 200 ),
  ('Emanuela', 50), ('Paolo', 25 );
```

Numero di righe e classifica

```
> SELECT name, score,
   row_number() OVER (),
   rank() OVER (ORDER BY score DESC)
  FROM score;
   name    | score | row_number | rank
-----------+-------+------------+------
 Diego     |   200 |          2 |    1
 Luca      |   100 |          1 |    2
 Emanuela  |    50 |          3 |    3
 Paolo     |    25 |          4 |    4
```

Distanza fra i classificati

```
> SELECT name, score,
   rank() OVER (ORDER BY score DESC),
   cume_dist() OVER ( ORDER BY score DESC )
  FROM score;
   name    | score | rank | cume_dist
----------+-------+------+-----------
 Diego    |   200 |    1 |      0.25
 Luca     |   100 |    2 |       0.5
 Emanuela |    50 |    3 |      0.75
 Paolo    |    25 |    4 |         1
```

L'ultimo classificato si trova a distanza 1, siccome ci sono 4 righe ogni riga si allontana dalla precedente di `1 / 4 = 0.25`.

Complicare un po' le cose...

Inseriamo dei punteggi doppi e riconsideriamo:

```
> SELECT name, score,
    rank() OVER (ORDER BY score DESC),
    cume_dist() OVER ( ORDER BY score DESC )
  FROM score;
   name    | score | rank |     cume_dist
-----------+-------+------+-------------------
 Diego     |   200 |    1 | 0.333333333333333
 Sara      |   200 |    1 | 0.333333333333333
 Luca      |   100 |    3 | 0.666666666666667
 Simone    |   100 |    3 | 0.666666666666667
 Emanuela |    50 |    5 | 0.833333333333333
 Paolo     |    25 |    6 |                 1
```

Questa volta le righe sono 6, quindi ci si sposta di 1 / 6 = 0,167 alla volta. Ma si noti che i punteggi doppi hanno la stessa distanza!

lag() e lead() in azione

```
 > SELECT name, score,
    lag( score, 1, 0 ) OVER ( ORDER BY score DESC ),
    lead( score, 1, 0 ) OVER ( ORDER BY score DESC )
  FROM score;
   name    | score | lag | lead
-----------+-------+-----+------
 Diego     |   200 |   0 |  200
 Sara      |   200 | 200 |  100
 Luca      |   100 | 200 |  100
 Simone    |   100 | 100 |   50
 Emanuela  |    50 | 100 |   25
 Paolo     |    25 |  50 |    0
```

In sostanza la prima riga nel *set* assume il valore di default (0), la riga successiva o precedente assume il valore score dell'altra riga e via così. Ci si muove a *zig-zag*.

Spezzare la finestra

```
 > SELECT name, score,
     ntile(4) OVER (ORDER BY score DESC )
   FROM score;
   name    | score | ntile
-----------+-------+-------
 Diego     |   200 |     1
 Sara      |   200 |     1
 Luca      |   100 |     2
 Simone    |   100 |     2
 Emanuela  |    50 |     3
 Paolo     |    25 |     4
```

`ntile()` cerca di dividere la finestra in *bucket* bilanciati, in questo caso 4, infilando un numero di valori pressoché uguali in ogni bucket.

CTE

Una *Common Table Expression (CTE)* è una forma speciale di statement SQL (`SELECT`, `INSERT`, `UPDATE`, `DELETE`) che viene *attaccato* mediante una clausola `WITH`.
Concettualmente è come definire una tabella temporanea che può essere usata in uno statement piu' complesso.

CTE SELECT

```
> WITH max_sw AS (
    SELECT max( version ) AS v, name
    FROM software GROUP BY name
  )
 SELECT name, v
  FROM max_sw
  ORDER BY name;
 name |  v
------+-----
 Java | 1.8
 Perl | 5.4
```

CTE UPDATE

```
> WITH delete_obsolete AS (
    UPDATE software SET valid = false
    WHERE version IN (
        SELECT MIN( version )
        FROM software GROUP BY name )
   RETURNING * )
  SELECT * FROM delete_obsolete;
 pk | name | version | valid
----+------+---------+-------
  1 | Perl |     5.1 | f
  5 | Java |     1.4 | f
```

CTE UPDATE (con piu' CTE)

```
> WITH min_version AS (
   SELECT name, min( version ) AS v
   FROM software GROUP BY name
  )
  , delete_obsolete AS (
    UPDATE software  SET valid = false
    WHERE version IN ( SELECT v FROM min_version )
    RETURNING *
  )
  SELECT * FROM delete_obsolete;
```

CTE ricorsive

Con la clausola `RECURSIVE` è possibile crea una query ricorsiva.
La query ricorsiva può referenziare se stessa, ovvero usare i risultati costruiti fino a quel momento per costruirne di nuovi.
Le query ricorsive possono produrre dei loop infiniti!
Le CTE ricorsive sono composte di due parti:

- una query non ricorsiva (che verrà evaluata per prima);
- una query che referenzia la CTE stessa in `UNION ALL`.

CTE ricorsive: generare una sequenza

```
> WITH RECURSIVE foo AS (
  SELECT 1 AS f
 UNION ALL
  SELECT 1 + f FROM foo )

SELECT * FROM foo LIMIT 10;
```

CTE ricorive: alberi

Il vantaggio delle CTE ricorsive si ha quando occorre generare dei *join* in numero variabile. Si supponga di avere una tabella che auto-referenzia se stessa:

```
> CREATE TABLE family_tree(
   pk int GENERATED ALWAYS AS IDENTITY,
   name text,
   parent_of int,
   FOREIGN KEY (parent_of) REFERENCES family_tree(pk),
   PRIMARY KEY(pk),
   CHECK( pk <> parent_of )  );
```

La tabella memorizza delle persone e la loro *catena di discendenza*. Si supponga di averla popolata con i seguenti dati:

```
> SELECT * FROM family_tree;
 pk |   name    | parent_of
----+-----------+-----------
  2 | Diego     |
  1 | Luca      |         2
  3 | Anselmo   |         1
  4 | Emanuela  |
  5 | Paolo     |         4
```

Ovvero la catena di discendenza è: *Anselmo -> Luca -> Diego* e *Paolo ->*

CTE ricorsive: attraversamento dell'albero

```
WITH RECURSIVE dump_family AS (
  -- parte non ricorsiva: figlio
  SELECT *,
         'FIGLIO' AS description
  FROM family_tree
  WHERE parent_of IS NULL
UNION
  -- parte ricorsiva: join dei genitori
  SELECT f.*,
         'GENITORE di ' || df.name AS description
  FROM dump_family df
       JOIN family_tree f ON f.parent_of = df.pk )

-- query finale
SELECT name, description FROM dump_family;
```

CTE ricorsive: attraversamento dell'albero (risultato)

```
   name    |      description
-----------+-----------------------
 Diego     | FIGLIO
 Emanuela  | FIGLIO
 Luca      | GENITORE di Diego
 Paolo     | GENITORE di Emanuela
 Anselmo   | GENITORE di Luca
```

CTE ricorsive: esplosione dell'albero

Basta modificare la colonna `description` in append per avere una *esplosione* dell'albero:

```
> WITH RECURSIVE dump_family AS (
   SELECT *, name || ' (FIGLIO)' AS description
   FROM family_tree WHERE parent_of IS NULL
UNION
   SELECT f.*, f.name || ' -> ' || df.description AS description
   FROM dump_family df
        JOIN family_tree f ON f.parent_of = df.pk )

SELECT name, description FROM dump_family;

   name    |             description
-----------+-------------------------------------
 Diego     | Diego (FIGLIO)
 Emanuela  | Emanuela (FIGLIO)
 Luca      | Luca -> Diego (FIGLIO)
 Paolo     | Paolo -> Emanuela (FIGLIO)
 Anselmo   | Anselmo -> Luca -> Diego (FIGLIO)
```

CTE ricorsive: profondità dell'albero

Basta introdurre anche una colonna di conteggio e si ottiene la profondità dell'albero:

```
> WITH RECURSIVE dump_family AS (
    SELECT *,
      name || ' (FIGLIO)' AS description,
      1 AS deep
    FROM family_tree WHERE parent_of IS NULL
UNION
    SELECT f.*,
      f.name || ' -> ' || df.description AS description,
      df.deep + 1 AS deep
    FROM dump_family df
      JOIN family_tree f ON f.parent_of = df.pk )

SELECT name, description, deep FROM dump_family;

   name    |             description              | deep
-----------+--------------------------------------+------
 Diego     | Diego (FIGLIO)                       |    1
 Emanuela  | Emanuela (FIGLIO)                    |    1
 Luca      | Luca -> Diego (FIGLIO)               |    2
 Paolo     | Paolo -> Emanuela (FIGLIO)           |    2
 Anselmo   | Anselmo -> Luca -> Diego (FIGLIO)    |    3
```

CTE ricorsive: esplorazione di un file system

Si supponga di avere la seguente tabella:

```
> select * from fs;
 pk |   name    | child_of | dir
----+-----------+----------+-----
  1 | /         |          | t
  2 | bin       |        1 | t
  3 | tmp       |        1 | t
  4 | home      |        1 | t
  5 | luca      |        4 | t
  6 | Desktop   |        5 | t
  7 | emacs     |        2 | f
  8 | cat.png   |        6 | f
```

CTE ricorsive: estrarre il path di un file e quanti directory

```
> WITH RECURSIVE percorso AS (
   -- termine non ricorsivo, ROOT!
   SELECT name, pk, dir, child_of,
          0 AS cd
    FROM fs
    WHERE child_of IS NULL
  UNION
   -- termine ricorsivo
   SELECT r.name ||
          CASE WHEN r.child_of IS NOT NULL THEN '/'
          ELSE '' END || f.name,
          f.pk, f.dir, f.child_of,
          CASE WHEN f.dir = false THEN r.cd
          ELSE r.cd + 1 END
   FROM fs f JOIN percorso r ON f.child_of = r.pk )

  SELECT name, cd FROM percorso WHERE dir = false;
            name              | cd
------------------------------+----
 /bin/emacs                   |  1
 /home/luca/Desktop/cat.png   |  3
```

CTE ricorsive: viste

E' possibile creare una vista ricorsiva:

```
> CREATE RECURSIVE VIEW
  percorso( name, pk, dir, child_of, cd )
  AS (
   SELECT name, pk, dir, child_of,
          0 AS cd
    FROM fs
    WHERE child_of IS NULL
  UNION
   -- termine ricorsivo
   SELECT r.name ||
          CASE WHEN r.child_of IS NOT NULL THEN '/'
          ELSE '' END || f.name,
          f.pk, f.dir, f.child_of,
          CASE WHEN f.dir = false THEN r.cd
          ELSE r.cd + 1 END
   FROM fs f JOIN percorso r ON f.child_of = r.pk );
```

Partitioning

Per *partitioning* si intende la possibilità di "dividere" i dati orizzontalmente, ovvero per quantità (non per tipologia) in modo da consentire un miglior accesso agli stessi.
Quando i dati tendono ad acculumarsi, è infatti possibile avere un rallentamento di performance, che può essere mitigato spezzando i dati in *gruppi (partizioni)* da interrogare "intelligentemente".

Quando si può effettuare il partitioning?

In linea di principio è sempre possibile effettuare il partitioning, ma farlo con dei dati "live" necessita di una ristrutturazione del database e del conseguente spostamento delle tuple. Non è sempre possibile "anticipare" le necessità di paritioning (e occorre ricordarsi che l'ottimizzazione preventiva è spesso causa di mal di testa!).
In generale:

- se si effettua il partitioning prima dell'inserimento dei dati si è nel caso *facile*: si strutturano i dati opportunamente da subito;
- se si effettua il partitioning con dati live si è nel caso *difficile*, ma non impossibile!

Come si effettua il partitioning?

PostgreSQL 10 implementa una sintassi apposita per il *declarative partitioning*, ovvero per indicare come spezzare in gruppi (partizioni) una tabella secondo due possibilità:

- list partitioning ossia si indicano esattamente quali valori di chiave appartengono ad ogni partizione;
- range partitioning si indica la chiave (ma non i valori che assume) sulla quale effettuare il partitioning.

Il *declarative partitioning* può essere effettuato solo se i dati non sono gia' presenti: PostgreSQL non ammette di mutare una tabella ordinaria in una partizionata con la sintassi per il partitioning.
E' quindi necessario usare il "vecchio" sistema di partizionamento basato su table inheritance + triggers/rules + constraint exclusion.

Su cosa si partiziona?

Trovare il vincolo di discriminazione fra le partizioni dei dati è il quesito principale.
Lo scopo dovrebbe essere quello di ottenere partizioni omogenee (in termini di dimensioni) e "stabili" (ossia con poche variazioni)/: piu' una partizione è "fissa" e piu' facile è ottimizzarla.
La scelta della colonna/colonne di partizionamento e del relativo valore dipende dal dominio dei dati.

Workflow per il *declarative partitioning*

1. definire la tabella *master* con la relativa struttura che sarà poi "clonata" su ogni partizione. La tabella non ammette chiavi primarie e/o vincoli di univocità. E' necessario specificare il tipo di partizionamento da operare:
   - `BY LIST` implica che ogni partizione indichi esplicitamente i propri valori accettati;
   - `BY RANGE` implica che ogni partizione indichi gli estremi del range dei valori ammessi;
2. creare le tabelle partizioni senza indicare alcuna struttura dati, ma solo `PARTITION OF` seguito dalla lista dei valori (o range) ammessi.
3. creare i constraint (`UNIQUE` e `PRIMARY KEY`) per ogni partizione figlia (se necessario);

Workflow per il partizionamento via `INHERITS`

1. definire le partizioni e dichiarare le tabelle;
2. abilitare l'instradamento delle query dalla tabella principale alle partizioni, in modo da impedire che la tabella principale sia "sporcata" con nuovi dati;
3. eventualmente spostare i dati dalla tabella principale a quella con il vincolo corretto;
4. verificare `constraint_exclusion` per aiutare l'ottimizzatore a instradare subito le query sulla partizione giusta.

Instradamento automatico: constraint_exclusion

Il partizionamento deve essere trasparente: le query devono poter avvenire "contro" la tabella principale ed essere *rimbalzate* alla partizione corretta. Il parametro `constraint_exclusion` permette all'ottimizzatore di analizzare i vincoli (*constraint*) di tabella al fine di "comprendere" meglio come instradare una query. I valori possibili per questo parametro sono:

- `on`: abilitato per tutte le query (aumenta lo *sforzo* del planner);
- `off`: disabilitato per tutte le query;
- `partition` (default): abilita il meccanismo solo in caso di tabelle con `INHERITS` e `PARTITION BY`.

```
constraint_exclusion = partition
```

Esempio by list

Supponiamo di avere una tabella `persona` che memorizza il `sesso`, e di voler partizionare appunto sul sesso. Si supponga di essere nel caso "facile": dati non presenti.
Questo esempio è volutamente semplice per mostrare i concetti del partitioning!

Esepio by list: preparazione

```
> CREATE TABLE persona (
    pk SERIAL NOT NULL,
    nome text,
    cognome text,
    codice_fiscale text NOT NULL,
    sesso char(1) DEFAULT 'M',
    CHECK ( sesso IN ( 'M', 'F' ) ) )
    PARTITION BY LIST(sesso);
```

Nessun vincolo `UNIQUE` o `PRIMARY KEY` è ammesso su una tabella sottoposta a partizionamento!

Esempio by list: creazione delle partizioni

```
> CREATE TABLE persona_maschio
  PARTITION OF persona
  FOR VALUES IN ('M');

> CREATE TABLE persona_femmina
  PARTITION OF persona
  FOR VALUES IN ('F');
```

Esempio by list: popolamento

```
> WITH random_seq AS (
    SELECT generate_series( 1, 10000 )
           AS random_val
  )
 INSERT INTO persona( nome, cognome, codice_fiscale, sesso )
 SELECT 'Nome' || random_val,
        'Cognome' || random_val,
        substring( md5( random_val::text ) from 1 for 16 ),
        CASE  WHEN random_val % 2 = 0 THEN 'M'
              ELSE 'F'
        END
  FROM random_seq;
```

Esempio by list: risultato

```
> SELECT count(*), sesso FROM persona GROUP  BY sesso;
 count | sesso
-------+-------
  5000 | F
  5000 | M

> SELECT count(*), sesso
  FROM persona_maschio
  GROUP BY sesso
     UNION

  SELECT count(*), sesso
  FROM persona_femmina
  GROUP  BY sesso;
 count | sesso
-------+-------
  5000 | F
  5000 | M
```

Esempio by range: data di nascita

Si supponga di voler partizionare per data di nascita:

```
> CREATE TABLE persona (
    pk SERIAL NOT NULL,
    nome text,
    cognome text,
    data_nascita date NOT NULL,
    codice_fiscale text NOT NULL,
    sesso char(1) DEFAULT 'M',
    CHECK ( sesso IN ( 'M', 'F' ) ) )
    PARTITION BY RANGE(data_nascita);
```

Esempio by range: creazione delle partizioni

Nelle partizioni occorre inserire il range dei valori ammessi:

```
> CREATE TABLE persona_last_century
  PARTITION OF persona
  FOR VALUES FROM ('1900-01-01')
              TO ('1999-12-31');

> CREATE TABLE persona_millenial
  PARTITION OF persona
  FOR VALUES FROM ('2000-01-01')
              TO ('2100-12-31');
```

Esempio by range: popolamento

```
> WITH random_seq AS (
    SELECT generate_series( 1, 10000 )
           AS random_val
  )
 , random_date AS (
   SELECT '1990-01-01'::date
          + random() * interval '123 years'
          - random() * interval '6 months'
          + random() * interval '852 days'
          AS random_ts
  )
 INSERT INTO persona( nome, cognome, codice_fiscale, sesso, data_nascita )
 SELECT 'Nome' || random_val,
        'Cognome' || random_val,
        substring( md5( random_val::text ) from 1 for 16 ),
        CASE  WHEN random_val % 2 = 0 THEN 'M'
              ELSE 'F'
        END
        , random_ts::date
  FROM random_seq, random_date;
```

Esempio by range: risultato

```
> SELECT count(*) FROM ONLY persona
  UNION ALL
  SELECT count(*) FROM persona_last_century
  UNION ALL
  SELECT count(*) FROM persona_millenial;
 persona
---------
       0
   10000
   10000
```

Cosa succede se si cerca di spostare una tupla da una partizione ad un'altra?

Il sistema non permette di spostare tramite UPDATE una tupla da una partizione ad un'altra:

```
> UPDATE persona_millenial
  SET data_nascita = '1990-01-07'
  WHERE pk = 1;

ERROR:  new row for relation "persona_millenial" violates partition constraint
DETAIL:  Failing row contains (1, Nome1, Cognome1, 1990-01-07, c4ca4238a0b92382, F)
```

Cosa succede se manca una partizione?

Il sistema non accetta l'inserimento di tuple che non possono ricadere su una partizione:

```
> INSERT INTO persona(codice_fiscale,  data_nascita)
  VALUES( 'FRRLCU77H88F257B', '1800-01-09');
ERROR:  no partition of relation "persona" found for row
DETAIL:  Partition key of the failing row contains (data_nascita) = (1800-01-09).
```

Attaccare una nuova partizione

```
> CREATE TABLE persona_1800
  PARTITION OF persona
  FOR VALUES FROM ('1800-01-01')
             TO ('1899-12-31');

> INSERT INTO persona(codice_fiscale, data_nascita)
  VALUES( 'FRRLCU77H88F257B', '1800-01-09');

> SELECT count(*) FROM persona_1800;
 count
-------
     1
```

Esempio con INHERITS

Si supponga di essere nel caso in cui la tabella persona è già stata popolata e la si vuole partizionare a posteriori sulla base dei valori della colonna sesso:

```
> CREATE TABLE persona (
    pk SERIAL NOT NULL,
    nome text,
    cognome text,
    codice_fiscale text NOT NULL,
    sesso char(1) DEFAULT 'M',
    CHECK ( sesso IN ( 'M', 'F' ) ) );

> WITH random_seq AS (
    SELECT generate_series( 1, 10000 )
           AS random_val
  )
 INSERT INTO persona( nome, cognome, codice_fiscale, sesso )
 SELECT 'Nome' || random_val,
        'Cognome' || random_val,
        substring( md5( random_val::text ) from 1 for 16 ),
        CASE  WHEN random_val % 2 = 0 THEN 'M'
              ELSE 'F'
        END
  FROM random_seq;
```

Setup delle tabelle

```
> CREATE TABLE persona_maschio(
    -- nessun campo !
    CHECK ( sesso = 'M' )
  ) INHERITS( persona );

> CREATE TABLE persona_femmina(
    -- nessun campo !
    CHECK ( sesso = 'F' )
  ) INHERITS( persona );
```

Partitioning by rule

Si vuole *instradare* ogni inserimento di una tupla "maschio" nella relativa tabella:

```
> CREATE OR REPLACE RULE r_partition_insert_persona_maschio AS
   ON INSERT
   TO persona
   WHERE sesso = 'M'
   DO INSTEAD
 INSERT INTO persona_maschio SELECT NEW.*;
```

Vanno create le realtive rule per ogni casistica (`INSERT`, `UPDATE`, `DELETE`) e instradamento (*maschio*, *femmina*).

Partitioning by trigger

Si vogliono instradare gli inserimenti nella tabella corretta:

```
> CREATE OR REPLACE FUNCTION f_tr_persona_insert()
  RETURNS TRIGGER
  AS $BODY$
 DECLARE
   BEGIN
   IF NEW.sesso = 'M' THEN
     INSERT INTO persona_maschio SELECT NEW.*;
   ELSIF NEW.sesso = 'F' THEN
     INSERT INTO persona_femmina SELECT NEW.*;
   END IF;
 RETURN NULL;
 END;
 $BODY$
 LANGUAGE plpgsql;
```

Partitioning by trigger (2)

Si aggancia il trigger:

```
> CREATE TRIGGER tr_persona_insert
    BEFORE INSERT
    ON persona
    FOR EACH ROW
    EXECUTE PROCEDURE f_tr_persona_insert();
```

Spostamento dei dati: maschi

```
> WITH move_persona AS (
        DELETE FROM ONLY persona
        WHERE sesso = 'M'
        RETURNING * )
  INSERT INTO persona_maschio
  SELECT * FROM move_persona;

> SELECT count(*),sesso
  FROM ONLY persona
  GROUP BY sesso;
 count | sesso
-------+-------
  5000 | F
```

Si noti l'uso di `FROM ONLY` nelle query! La clausola `FROM ONLY` indica a PostgreSQL di *spezzare* la catena di ereditarietà senza percorrerla, fermandosi solo alla tabella indicata senza andare a ricercare anche nelle tabelle figlie.

Spostamento dei dati: femmine

```
> WITH move_persona AS (
      DELETE FROM ONLY persona
      WHERE sesso = 'F'
      RETURNING * )
  INSERT INTO persona_femmina
  SELECT * FROM move_persona;

> SELECT count(*),sesso
  FROM ONLY persona
  GROUP BY sesso;
 count | sesso
-------+-------
(0 rows)
```

Test di inserimento

```
> INSERT INTO persona( nome, cognome, codice_fiscale, sesso )
  VALUES( 'Luca', 'Ferrari', 'FRRLCU78L19F257B', 'M' );
```

Dove è finita la tupla?

```
-- la trovo con una query generica senza
-- indicare le tabelle figlie !
> SELECT * FROM persona
  WHERE codice_fiscale = 'FRRLCU78L19F257B';
  pk   | nome | cognome |  codice_fiscale  | sesso
-------+------+---------+------------------+-------
 12001 | Luca | Ferrari | FRRLCU78L19F257B | M

-- NON e' nella tabella generale!
> SELECT * FROM ONLY persona
  WHERE codice_fiscale = 'FRRLCU78L19F257B';
 pk | nome | cognome | codice_fiscale | sesso
----+------+---------+----------------+-------
(0 rows)

-- è nella tabella figlia giusta !
> SELECT * FROM ONLY persona_maschio
  WHERE codice_fiscale = 'FRRLCU78L19F257B';
  pk   | nome | cognome |  codice_fiscale  | sesso
-------+------+---------+------------------+-------
```

Instradamento automatico

Le query di SELECT e di DELETE vengono instradate su tutte le tabelle figlie, quindi non importa impostare delle regole o dei trigger per gestirle. Ovviamente dovendo percorrere tutte le tabelle della partizione si incorre in un rischio di calo di performance, quindi l'unica ragione per scendere a livello di dettaglio su questi statement è un aumento di prestazioni.

```
-- cancello senza sapere dove si trova la tupla
> DELETE FROM persona
  WHERE codice_fiscale = 'FRRLCU78L19F257B';

> SELECT * FROM persona
  WHERE codice_fiscale = 'FRRLCU78L19F257B';
 pk | nome | cognome | codice_fiscale | sesso
----+------+---------+----------------+-------
(0 rows)
```

Instradamento automatico: svantaggio

L'instradamento automatico e il calo di performance è visibile nel piano di esecuzione:

```
 > EXPLAIN SELECT * FROM persona WHERE codice_fiscale = 'FRRLCU78L19F257B';
                                QUERY PLAN
-------------------------------------------------------------------------
 Append  (cost=0.00..219.00 rows=3 width=42)
   ->  Seq Scan on persona  (cost=0.00..0.00 rows=1 width=42)
         Filter: (codice_fiscale = 'FRRLCU78L19F257B'::text)
   ->  Seq Scan on persona_maschio  (cost=0.00..109.50 rows=1 width=42)
         Filter: (codice_fiscale = 'FRRLCU78L19F257B'::text)
   ->  Seq Scan on persona_femmina  (cost=0.00..109.50 rows=1 width=42)
         Filter: (codice_fiscale = 'FRRLCU78L19F257B'::text)
```

Vengono percorse tutte le tabelle nella partizione!

Instradamento automatico con constraint_exclusion

```
> EXPLAIN SELECT * FROM persona WHERE sesso = 'M' AND codice_fiscale like 'FRR%';
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Append  (cost=0.00..122.00 rows=2 width=42)
   ->  Seq Scan on persona  (cost=0.00..0.00 rows=1 width=42)
         Filter: ((codice_fiscale ~~ 'FRR%'::text) AND (sesso = 'M'::bpchar))
   ->  Seq Scan on persona_maschio  (cost=0.00..122.00 rows=1 width=42)
         Filter: ((codice_fiscale ~~ 'FRR%'::text) AND (sesso = 'M'::bpchar))
```

La tabella persona_femmina non viene presa in considerazione poiché il planner è in grado di stabilire a priori che il vincolo della query esclude tale tabella.

Instradamento automatico senza constraint_exclusion

```
> EXPLAIN SELECT * FROM persona WHERE sesso = 'M' AND codice_fiscale like 'FRR%';
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Append  (cost=0.00..244.00 rows=3 width=42)
   ->  Seq Scan on persona  (cost=0.00..0.00 rows=1 width=42)
         Filter: ((codice_fiscale ~~ 'FRR%'::text) AND (sesso = 'M'::bpchar))
   ->  Seq Scan on persona_maschio  (cost=0.00..122.00 rows=1 width=42)
         Filter: ((codice_fiscale ~~ 'FRR%'::text) AND (sesso = 'M'::bpchar))
   ->  Seq Scan on persona_femmina  (cost=0.00..122.00 rows=1 width=42)
         Filter: ((codice_fiscale ~~ 'FRR%'::text) AND (sesso = 'M'::bpchar))
```

Il planner non sa che ottimizzazioni fare sulla partizione, quindi percorre tutte le tabelle!

Introduzione alle estensioni

PostgreSQL indica con il termine *extension* un *pacchetto* di oggetti correlati fra di loro e che devono essere gestiti coerentemente in blocco. Ad esempio una estensione può includere la definizione di procedure, trigger, tipi di dato personalizzato, ecc., che vengono *racchiusi* in un pacchetto (l'estensione stessa) che viene gestita in blocco.
Gestire una estensione in blocco significa essere in grado di installare, spostare, aggiornare ed eliminare l'intero insieme di oggetti con comandi singoli. Inoltre PostgreSQL riconoscerà ogni singolo oggetto come appartenente all'estensione e ne impedirà la movimentazione/cancellazione scorrelata dagli altri oggetti.

Componenti di una estensione

Una *extension* è formata almeno da:

- *un file script* che contiene gli statement SQL necessari per definire e caricare l'estensione stessa (es. `CREATE FUNCTION`, `CREATE TYPE`, ...);
- *un file di controllo* che identifica degli attributi dell'estensione (versione, ecc.);
- *eventuali librerie (shared objects)* da caricare nel server assieme all'estensione (qualora questa dipenda da codice compilato).

Entrambi i file hanno un nome che deve iniziare con il nome dell'estensione stessa.

Ogni estensione ha un numero di versione: PostgreSQL utilizza i numeri di versione (che devono comparire nel nome del file di script e, opzionalmente, in quello di controllo) per effettuare gli aggiornamenti.

Il control file

Il *control file* è un file con dei metadati relativi all'estensione. La sintassi è nel formato `chiave = valore` e il nome del file deve quello dell'estensione stessa con il suffisso `.control`.
E' anche possibile specificare un file di controllo dipendente dalla versione specifica di una estensione, nel qual caso il nome del file deve essere con il suffisso `--<version>.control`.

Il file di script

Lo *script file* è un file con nome pari a quello dell'estensione, seguito da due trattini e il numero di versione, infine il suffisso `.sql`, ovvero:
`<extension>--1.2.sql`

Movimentazione di una estensione

Una estensione si dice *relocatable* (spostabile) se può essere migrata da uno schema ad un altro.
Le estensioni hanno tre livelli di movimentazione:

- *fully relocatable* l'estensione non dipende dallo schema in cui gira;
- *partially relocatable* l'estensione può essere installata in qualunque schema ma poi non può essere spostata;
- *non-relocatable* l'estensione deve essere installata in uno schema specifico e da lì non può essere mossa.

Il control file governa la *relocation* di una estensione:

parametro	valore	relocatable
`relocatable`	`true`	*fully*
`schema`		
`relocatable`	`false`	*partial*
`schema`		
`relocatable`	`false`	*none*
`schema`	`<schema_name>`	

Dump dei dati di una estensione

Una estensione può creare tabelle (o sequenze) da usare come spazio di configurazione. Tali dati non saranno sottoposti a *dump* del database: l'estensione tornerà a creare tali dati! Tuttavia potrebbe capitare che l'utente modifichi tali dati di configurzione, nel qual caso è importante farne il dump.
L'estensione deve informare il catalogo di sistema che gli oggetti devono essere sottoposti a dump: la funzione `pg_catalog.pg_extension_config_dump()` accetta il nome dell'oggetto da inserire nel dump.

Upgrade di una estensione

Per effettuare un upgrade di una estensione si può usare il comando `ALTER EXTENSION`.
Lo script file deve includere nel nome del file la versione di partenza e quella di arrivo, ad esempio: `<extension_name>--1.0--1.1-sql`
Nel caso di upgrade ad una versione, sono eseguiti tutti i file script in ordine per garantire il salto di versione finale.
*

Fase 1: script file

Il file `software--1.0.sql`:

```
CREATE TYPE sw_version
  AS ENUM ( 'stable',
            'unstable',
            'EOL',
            'development' );

CREATE TYPE sw_repository
  AS ( url text,
       branch text,
       author text );

--tabella configurazione
CREATE TABLE repositories(
   pk SERIAL PRIMARY KEY,
   repo sw_repository,
   -- flag boolean per indicare se la tupla
   -- è di configurazione o inserita da un utente
   user_defined boolean DEFAULT true
);

INSERT INTO repositories( repo, user_defined )
VALUES( ( 'git@github.com:fluca1978/fluca1978-pg-utils.git', true ),
        false
```

Fase 2: definizione del file di controllo

Il file `software.control`:

```
comment = 'Una semplice estensione'
default_version = '1.0'
relocatable = true
```

Fase 3: (opzionale) Makefile

Il file `Makefile` utilizza `pg_config` per sapere dove installare i file (*sharedir*):

```
EXTENSION = software
DATA = software--1.0.sql

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

Fase 4: installare l'estensione

```
% sudo make install
/bin/mkdir -p '/usr/share/postgresql/9.5/extension'
/bin/mkdir -p '/usr/share/postgresql/9.5/extension'
/usr/bin/install -c -m 644 .//software.control '/usr/share/postgresql/9.5/extension
/usr/bin/install -c -m 644 .//software--1.0.sql  '/usr/share/postgresql/9.5/extensi
```

Fase 5: usare l'estensione

Nel database si può usare `CREATE EXTENSION` per creare l'estensione dentro ad uno specifico database:

```
# CREATE EXTENSION software;

# SELECT (repo).author, (repo).branch FROM repositories;
    author     |     branch
---------------+----------------
 Luca Ferrari | postgresql-9.6
```

Fase 6: creare un aggiornamento dell'esntesione

Il file `software--1.0--1.1.sql`:

```
INSERT INTO repositories( repo, user_defined )
VALUES( ( 'git@github.com:fluca1978/fluca1978-coding-bits.git',
          'master',
    'Luca Ferrari' ),
        false
     );

ALTER TABLE repositories ADD COLUMN active boolean DEFAULT true;
```

e il `Makefile`

```
EXTENSION = software
DATA = software--1.0--1.1.sql

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

Fase 7: installare l'aggiornamento

Installare i file dove il cluster li può trovare:

```
% sudo make install
```

e poi si aggiorna l'esntesione:

```
# ALTER EXTENSION software UPDATE TO '1.1';

# SELECT (repo).author, (repo).branch FROM repositories;
    author     |     branch
---------------+----------------
 Luca Ferrari | postgresql-9.6
 Luca Ferrari | master
```

Spostare l'esntesione

Supponiamo di voler spostare l'estensione in un altro schema:

```
# CREATE SCHEMA sw_ext;

# ALTER EXTENSION software SET SCHEMA sw_ext;

# SELECT (repo).author, (repo).branch
  FROM sw_ext.repositories; -- nuovo schema!
    author     |     branch
---------------+----------------
 Luca Ferrari | postgresql-9.6
 Luca Ferrari | master
```

Eliminare l'estesione

Con il comando `DROP EXTENSION` tutti gli oggetti sono eliminati in un colpo solo:

```
# DROP EXTENSION software;
```

Infrastruttura per le estensioni

la *PGXS* è una infrastruttura per la gestione delle estensioni, in particolare quelle che includono codice nativo.
Le estensioni che vogliano usare *PGXS* devono usare un `Makefile` che includa quello di *PGXS*.

```
EXTENSION = software
DATA = software--1.0.sql
PG_CONFIG = pg_config

# imposta PGXS al file Makefile appropriato
# e lo include!
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

PGXN

La *PostgreSQL eXtension Network (PGXN)* è un repository online di estensioni che possono essere scaricate, disponibile all'URL `https://pgxn.org/`.
Le estensioni possono essere scaricate e installate con la procedura `make` vista in precedenza.

pgxnclient

Esiste un client che funge da shell per la *PGXN*. Va installato come pacchetto dipendente dalla distribuzione, ad esempio:

```
% sudo pkg install pgxnclient-1.2.1_1
```

o anche

```
% sudo apt-get install pgxnclient
```

Workflow con pgxn

Il comando shell `pgxn` ricorda un po' `cpan` di Perl (e gli analoghi per altri ecosistemi).
Il workflow solitamente è:

- *cercare* un pacchetto e vederne le informazioni;
- *installarlo* azione in piu' fasi che richiede un *download*, *build* e *install*.

Esempio di workflow con pgxn: ricerca di un pacchetto

pgxn in default cerca nella documentazione! Occorre specificare `--ext` per cercare fra le estensioni.

Si supponga di voler cercare qualcosa relativo al table partitioning:

```
% pgxn search --ext partitioning
pg_pathman 1.4.7
    *Partitioning* tool

pg_part 0.1.0
    Table *partitioning* extention
...
```

Esempio di workflow con pgxn: info di un pacchetto

Stabilito il nome di un pacchetto da un `search`, si può usare il comando `info` per ottenerne delle informazioni:

```
% pgxn info pg_part
name: pg_part
abstract: Table partitioning utility for PostgreSQL
WARNING: data key 'description' not found
maintainer: Satoshi Nagayasu <snaga@uptime.jp>
license: gpl_2
release_status: stable
version: 0.1.0
date: 2013-01-04T03:16:26Z
sha1: c376651dcbe3fddf01f24824e68c2774f48ef434
provides: pg_part: 0.1.0
```

Esempio di workflow con pgxn: installazione di un pacchetto

A questo punto si può procedere all'installazione:

```
% sudo pgxn install pg_part
...
```

In default pgxn installa solo pacchetti marcati come stable (nella sezione release_status). E' possibile forzare l'installazione di un pacchetto con le opzioni auto-esplicative --testing e --unstable.
Su alcune piattaforme è bene specificare il comando make da usare con --make, ad esempio:

```
% sudo pgxn install --make `which gmake`  pg_part
...
```

Introduzione

Un esempio di estensione interessante è `psql-http` che fornisce una serie di funzioni per gestire richieste e risposte *HTTP*.
L'estensione utilizza `curl(1)` per il proprio funzionamento, e consente anche di impostare alcune informazioni prima di invocare il processo esterno.

Installazione

```
% git clone https://github.com/pramsey/pgsql-http.git
% cd psql-http
% gmake && sudo gmake install
```

e all'interno del database che deve usare le utilità HTTP occorre creare l'estensione:

```
# CREATE EXTENSION http;
```

pgsql-http informazioni generali

Sono messe a disposizione una serie di funzioni particolari di tipo `http*`, le principali:

- `http_get()` e `http_post` invocano l'URL specificato con i dati eventualmente passati come metodo *GET* o *POST*;
- `http_head()` manipola l'header http;
- `urlencode` per creare stringhe con caratteri speciali;
- `http_set_curlopt()` e `http_reset_curlopt()` imposta una opzione per `curl` o le resetta tutte.

pgsql-http richieste e response

I metodi di pgsql-http manipola un tipo particolare http_request e il corrispondente http_reponse, definiti come:

```
# \d http_request

    Column     |       Type
---------------+------------------
 method        | http_method
 uri           | character varying
 headers       | http_header[]
 content_type  | character varying
 content       | character varying

# \d http_response

    Column     |       Type
---------------+------------------
 status        | integer
 content_type  | character varying
 headers       | http_header[]
 content       | character varying
```

Esempio di utilizzo di pgsql-http

Quali sono i link da cui scaricare le versioni di PostgreSQL?

```
# SELECT unnest( xpath( '/rss/channel/item/link/text()'::text, content::xml ) )
    FROM http_get( 'https://www.postgresql.org/versions.rss' );

                                  unnest
-------------------------------------------------------------------------
 https://www.postgresql.org/docs/10/static/release-10-1.html
 https://www.postgresql.org/docs/9.6/static/release-9-6-6.html
 https://www.postgresql.org/docs/9.5/static/release-9-5-10.html
 https://www.postgresql.org/docs/9.4/static/release-9-4-15.html
 https://www.postgresql.org/docs/9.3/static/release-9-3-20.html
 https://www.postgresql.org/docs/9.2/static/release-9-2-24.html
 https://www.postgresql.org/docs/9.1/static/release-9-1-24.html
 https://www.postgresql.org/docs/9.0/static/release-9-0-23.html
 https://www.postgresql.org/docs/8.4/static/release-8-4-22.html
 https://www.postgresql.org/docs/8.3/static/release-8-3-23.html
 https://www.postgresql.org/docs/8.2/static/release-8-2-23.html
...
```

Introduzione

Il modulo `auto_explain` permette di specificare una soglia (ad esempio di tempo) oltre la quale una query viene automaticamente sottoposta ad `EXPLAIN` esplicito, e il risultato viene riportato nei log del cluster. Questo consente di ripercorrere i log a caccia di query *lente* avendo già il relativo piano di accesso.
ATTENZIONE: la configurazione di default /non effettua alcun `EXPLAIN`: occorre configurare le soglie per ottenere qualche risultato!

Soglie per auto_explain

Le principali soglie configurabili sono:

- `auto_explain.log_min_duration` indica il tempo di durata di una query (secondi, millisecondi, minuti, ecc.) affinché il piano di esecuzione sia salvato nei log testuali;
- `auto_explain.log_verbose` salva nei log testuali l'output come fosse dato un `EXPLAIN VERBOSE`;
- `auto_explain.log_analyze` se abilitato salva nei log l'output come fosse stato dato un `EXPLAIN ANALYZE` sulle query (rischioso in termini di performance);
- `auto_explain.log_timing` se abilitato salva nei log testuali anche il tempo di esecuzione, va in coppia con `.log_analyze`;

Utilizzo di auto_explain

Come utente amministratore:

```
# LOAD 'auto_explain';
# SET auto_explain.log_min_duration = '10ms';

# INSERT INTO trgm_test SELECT * FROM trgm_test;
```

e nei log testuali di PostgreSQL:

```
LOG:  duration: 941.351 ms  plan:
  Query Text: INSERT INTO trgm_test SELECT * FROM trgm_test;
  Insert on trgm_test  (cost=0.00..6291.00 rows=300000 width=33)
    ->  Seq Scan on trgm_test trgm_test_1  (cost=0.00..6291.00 rows=300000 width=33
```

Utilizzo senza LOAD

E' possibile indicare `auto_explain` nelle librerie da caricare all'avvio del cluster, così come i relativi parametri di configurazione:

```
session_preload_libraries = 'auto_explain'

auto_explain.log_min_duration = '10ms'
```

Introduzione a FTS

La *Full Text Search* (FTS) è un meccanismo di ricerca testuale, supera i normali operatori come `LIKE` e di regular expression.
La FTS si basa sulla capacità di suddividere il testo in *tokens*, riconoscendo parole, numeri, ecc., e normalizzare i token in *lexems*, ossia riportare le parole alla loro radice (es. plurali e singolari), infine la possibilità di ordinare le ricerche con indici ed effettuare valutazioni di similarità.
Infine la FTS *is basa sul supporto linguistico*, ovvero ci deve essere un meccanismo conosciuto per la lingua specificata di ottnere i lexems dal documento originale.

Dizionari

Nella terminologia FTS un *dizionario* è una collezione di termine che permettono di identificare le *stop words* (parole che non devono essere considerate nella ricerca), riconoscere i sinonimi e i plurali per normalizzare meglio una parola.
Solitamente il dizionario si basa su `ispell(1)`.
Mediante il dizionario il documento testuale viene ridotto ad un `tsvector` (insieme di lexems), analogamente una ricerca viene esegutia tramite un `tsquery` che spezza le parole anch'esse in lexems.

Esempio di lexem in atto: to_tsvector

La funzione speciale to_tsvector converte un documento di testo in un insieme di lexem specificandone la posizione:

```
> SELECT to_tsvector( 'italian',
     'PostgreSQL e'' uno dei database piu'' avanzati al mondo!' );
                      to_tsvector
-----------------------------------------------------
 'avanz':7 'databas':5 'mond':9 'piu':6 'postgresql':1
```

Come si può notare ogni termine è stato convertito in minuscolo, dopodiché le stop-words (es., avverbi) sono stati rimossi, e infine ogni parola è stata riportata alla sua radice con la posizione (indice) della parola nel documento. Ad esempio la parola database viene normalizzata a databas e si trova all'indice 5 nel documento di testo (l'indice parte da 1!).

Query FTS

Una query FTS si compone di un insieme di *termini* eventualmente uniti da uno dei seguenti operatori:

- & and logico, entrambi i termini devono essere presenti in qualunque ordine;
- | or logico, almeno uno dei due termini deve comparire;
- ! not logico, il termine non deve apparire;
- <-> *followed by*, il primo termine deve essere seguito dal secondo.

Query FTS: to_tsquery

Analogamente alla costruzione di un `tsvector`, la funzione speciale `to_tsquery` costruisce una query di tipo `tsquery` partendo dai termini e dagli operatori indicati:

```
> SELECT to_tsquery( 'italian', 'PostgreSQL & database & ! MySQL' );
             to_tsquery
------------------------------------
 'postgresql' & 'databas' & !'mysql'
```

Anche qui i termini sono *manipolati* in accordo alle regole per i lexem e l'uso dei dizionari.

Query FTS: plainto_tsquery e phraseto_tsquery

Queste due funzioni *non riconoscono gli operatori di tsquery* e quindi si limitano a inserire rispettivamente l'operatore & o <-> ai termini normalizzati:

```
> SELECT plainto_tsquery( 'italian',
        'PostgreSQL & database & ! MySQL' );

------------------------------------
 'postgresql' & 'databas' & 'mysql'
-- è sparito il NOT!!!

> SELECT phraseto_tsquery( 'italian',
         'PostgreSQL & database & ! MySQL' );

-----------------------------------------
 'postgresql' <-> 'databas' <-> 'mysql'
-- considera i termini come costruzione di una frase !!!
```

Tutti gli operatori di tsquery sono stati automaticamente rimossi!

Usare gli operatori fra tsvector e tsquery

Una volta che si sono costruiti i tsvector e tsquery è possibile applicare i seguenti operatori:

- @@ match verifica se un tsvector fa match con un tsquery;
- || concatena due tsvector o due tsquery;
- && effettua un and logico fra due tsquery;
- !! negazione di un tsquery;
- @> e <@ indicano se una tsquery ne contiene un'altra o è contenuta.

Esempio di operatori di FTS

```
> SELECT to_tsquery( 'italian', 'database & open <-> source' )
     && to_tsquery( 'italian', '! avanzato' );

 > SELECT to_tsquery( 'italian', 'database & open <-> source' )
      <@
    to_tsquery( 'italian',
                'database & free <-> libre <-> open <-> source' );

----------
 t

------------------------------------------
 'databas' & 'open' <-> 'sourc' & !'avanz'
```

Match

Una volta che si sono definiti i `tsvector` e `tsquery` è possibile effettuare le query di match con l'operatore `@@`:

```
> SELECT to_tsvector( 'italian',
         'PostgreSQL e'' uno dei database piu'' avanzati al mondo!' )
    @@
         to_tsquery( 'italian',
                     'database | piu'' <-> avanzato' );

----------
 t
```

Ambiente di esempio

```
> CREATE TABLE email( pk serial PRIMARY KEY, subject text, message text );
> INSERT INTO email( subject, message)
    VALUES( 'Corso su PostgreSQL',
            'Ciao, ho preparato le slide del corso e sono pronto a esportarle con b
          , ( 'Corso postgres',
              'Hai avuto modo di revisionare le slide del corso?' )
          , ( 'link slides',
              'Puoi scaricare le slide dal mio repository github, branch postgresql-
          , ( 'PR-175',
              'Can you please review my pull request #175 and give back some hint?'
```

Esempio di ricerca

Si vogliono trovare tutte le email che nel soggetto hanno informazioni sul corso per PostgreSQL:

```
> SELECT subject FROM email
    WHERE to_tsvector( 'italian', subject )
    @@ to_tsquery( 'italian', 'corso & postgres' );
    subject
---------------
 Corso postgres
```

SBAGLIATO! Il sistema non riconosce la parola PostgreSQL come italiana, quindi non sa che di fatto *PostgreSQL* e *Postgres* rappresentano la stessa cosa. Si può allora usare l'operatore speciale :* che indica che la parola rappresenta un *prefisso* per qualcosa di piu' complesso:

```
> SELECT subject FROM email
   WHERE to_tsvector( 'italian', subject )
   @@ to_tsquery( 'italian', 'corso & postgres:*' );
       subject
--------------------
 Corso su PostgreSQL
 Corso postgres
```

Ulteriore esempio

Si vogliono trovare tutte le email che nel soggetto o nel corpo del messaggio contengono dei riferimenti alle slide del corso:

```
> SELECT subject FROM email
   WHERE to_tsvector( 'italian', subject || ' ' || message )
    @@ to_tsquery( 'italian',
                   'slide & (corso | postgres:*)' );

       subject
---------------------
 Corso su PostgreSQL
 Corso postgres
 link slides
```

Ranking

La funzione `ts_rank()` permette di calcolare il *ranking* di una query FTS:

```
> SELECT subject,
    ts_rank( to_tsvector( 'italian', subject || ' ' || message ),
             to_tsquery( 'italian',
                         'slide & (corso | postgres:*)' ) )
  FROM email ORDER BY 2 DESC;

       subject        |  ts_rank
----------------------+-----------
 Corso su PostgreSQL  |  0.369819
 Corso postgres       |  0.334374
 link slides          | 0.0852973
 PR-175               |     1e-20
```

Pesi di un tsvector

La funzione setweight() consente di specificare un peso (in termini di lettere fra A e D) per una ricerca testuale. Solitamente si usa per assegnare ad una parte della ricerca un peso differente, concatenando i tsvector risultanti. Ad esempio:

```
> SELECT subject, ts_rank(
        setweight( to_tsvector( 'italian', subject ), 'A' )
        ||
        setweight( to_tsvector( 'italian', message ), 'C' ),
        to_tsquery( 'italian', 'corso | slide' ) )
  FROM email
  ORDER BY 2 DESC;
        subject        |  ts_rank
-----------------------+-----------
 Corso su PostgreSQL |  0.379954
 Corso postgres      |  0.379954
 link slides         | 0.0607927
 PR-175              |         0

> SELECT subject, ts_rank(
        setweight( to_tsvector( 'italian', subject ), 'C' )
        ||
        setweight( to_tsvector( 'italian', message ), 'A' ),
        to_tsquery( 'italian', 'corso | slide' ) )
```

Capire cosa è indicizzabile

Non tutta la parte di una `tsquery` è sempre indicizzabile, la funzione `querytree()` consente di capire cosa si può effettivamente indicizzare:

```
> SELECT querytree( to_tsquery( 'italian', 'corso | slide' ) );
    querytree
-----------------
 'cors' | 'slid'
```

Costruire un indice per query FTS

Si utilizza l'indice *GIN* specificando su quale `tsvector` si vuole indicizzare:

```
> CREATE INDEX fts_email_idx
   ON email
   USING GIN ( to_tsvector( 'italian', subject || ' ' || message ) );
```

e se ci sono parecchi valori da forzare l'uso dell'indice:

```
> EXPLAIN SELECT subject FROM email
   WHERE to_tsvector( 'italian', subject || ' ' || message )
       @@ to_tsquery( 'italian', 'corso' );
                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on email  (cost=2420.44..79786.25 rows=261218 width=13)
   Recheck Cond: (to_tsvector('italian'::regconfig, ((subject || ' '::text) || mess
   ->  Bitmap Index Scan on fts_email_idx  (cost=0.00..2355.13 rows=261218 width=0)
         Index Cond: (to_tsvector('italian'::regconfig, ((subject || ' '::text) || m
```

Introduzione

Se le query che si vogliono ottimizzare sono basate su `LIKE` non occorre una piena FTS, ma è sufficiente usare i *trigram*, gruppi di tre caratteri sui quali viene misurata la similarità (o la relativa distanza).
I trigrammi non includono caratteri NON-ALFANUMERICI!

Installazione

L'estensione `pg_trgm` fornisce tutto l'occorrente per abilitare l'uso dei trigrammi:

```
# CREATE EXTENSION pg_trgm;
```

Tabella di esempio

Si consideri una tabella piena di testo:

```
> CREATE TABLE trgm_test( t text );
> INSERT INTO trgm_test(t)
  SELECT md5( v::text )
  FROM generate_series(1,400000) v;

> CREATE INDEX no_trgm_idx ON trgm_test(t);
```

Esempio senza pg_trgm

L'indice funziona solo per comparazioni con *inizio stringa*:

```
> EXPLAIN ANALYZE SELECT * FROM trgm_test WHERE t like 'cac%';

 Bitmap Heap Scan on trgm_test
   Filter: (t ~~ 'cac%'::text)
   Heap Blocks: exact=25
   ->  Bitmap Index Scan on no_trgm_idx
         Index Cond: ((t >= 'cac'::text) AND (t < 'cad'::text))
```

ma non funziona per comparazioni *in mezzo alla stringa*:

```
> EXPLAIN ANALYZE SELECT * FROM trgm_test WHERE t like '%cac%';

 Seq Scan on trgm_test
   Filter: (t ~~ '%cac%'::text)

 Planning time: 0.293 ms
 Execution time: 115.258 ms
```

Esempio con pg_trgm

```
> CREATE INDEX trgm_idx ON trgm_test USING GIST (t gist_trgm_ops);

> EXPLAIN ANALYZE SELECT * FROM trgm_test WHERE t like '%cac%';

 Bitmap Heap Scan on trgm_test
   Recheck Cond: (t ~~ '%cac%'::text)
   Heap Blocks: exact=511
   ->  Bitmap Index Scan on trgm_idx
         Index Cond: (t ~~ '%cac%'::text)


 Planning time: 28.877 ms
 Execution time: 99.718 ms
```

Introduzione

SQLMED (SQL Management of External Data) è una specifica per la gestione di "dati esterni" aggiunta allo standard nel 2003.
PostgreSQL disponeva di `dbi-link`, un modulo Perl che permetteva la connessioni (tramite `DBI`) a origini dati esterne implementando così un *SQLMED del poveraccio!*.
Con il ramo di sviluppo 9 sono stati introdotti i *Foreign Data Wrapper (FDW)*, un meccanismo formale per permettere ad un cluster PostgreSQL di accedere dati esterni.

Tipologie di FDW

PostgreSQL supporta diversi tipi di Foreign Data Wrapper, da quelli *SQL* (verso altri database), a quelli di tipo *file* (es. accesso a CSV), a quelli Internet (web e repository di codice come `git`).
I data wrapper possono essere estesi implementando una serie di funzioni specifiche. La maggior parte dei FDW sono "nativi", altri sono scritti in Python, Rust e linguaggi di piu' alto livello.

Come si utilizzano i FDW?

Lo schema di utilizzo segue questo workflow:

- creare un *server remoto* che incapsula la *connessione* (qualunque cosa significhi) all'origine dati remota;
- eventualmente, se necessario, creare uno *user mapping*, ossia una traduzione da utente locale ad utente remoto;
- creare una mappatura di ogni tabella che deve essere accessibile sul server remoto.

FDW OPTIONS

Sia i server che le foreign table accettano una serie di opzioni specificate tramite OPTIONS e una sintassi come la seguente:

```
OPTIONS( name 'value', name 'value', ... )
```

con valori stringa.

Le opzioni sono definite dal server e dalla foreign table stessa.

CREATE SERVER

Il comando SQL `CREATE SERVER` consente di specificare una connessione remota da usarsi con un FDW. Ogni server deve avere un nome univoco e una serie di opzioni, tipo e versione che vengono interpretati dal FDW di riferimento.
E' obbligatorio specificare per quale FDW si sta creando il server.

CREATE USER MAPPING

Il comando SQL `CREATE USER MAPPING` consente di mappare un utente locale su un utente remoto. Va usato per quei server collegati a FDW che richiedono un qualche processo di autorizzazione (ad esempio un database esterno).
Lo user mapping viene specificato per un server remoto, ed è quindi collegato al server stesso.

CREATE FOREIGN TABLE

Il comando SQL `CREATE FOREIGN TABLE` crea una mappatura fra una tabella locale (fittizia) e una tabella remota.
Il comando deve essere agganciato ad un server remoto, affinché sia chiaro che la tabella locale punta ad una tabella remota.
Una foreign table segue le stesse regole di creazione di una tabella normale (schema, nome, ecc.).

Multicorn

Il progetto *Multicorn* fornisce una serie di implementazioni di FDW scritte in Python. Una volta installato il modulo e creata la relativa estensione, tutti i FDW sono disponibili.
La documentazione non è esaustiva!

Introduzione

Il modulo `file_fdw` consente di leggere dei dati da un file testuale (ad esempio CSV).
Si suppnga di avere un file di testo con il seguente contenuto:

```
% cat anagrafica.csv
Nome;Cognome;CodiceFiscale
Luca;Ferrari;FRRLCU78L19F257B
Diego;Ferrari;FRRDGI78L19F257B
```

Fase 1: caricare il FDW

```
# CREATE EXTENSION file_fdw;
```

Fase 2: creare il server

```
# CREATE SERVER csv_server
  FOREIGN DATA WRAPPER file_fdw;
```

Fase 3: mappare il file su una tabella

```
# CREATE FOREIGN TABLE remote_persona (
   nome text,
   cognome text,
   codice_fiscale text )
 SERVER csv_server
 OPTIONS ( filename '/home/luca/anagrafica.csv',
           format 'csv',
           delimiter ';',
           header 'true' );
```

Fase 4: interrogazione della tabella

```
# SELECT * FROM remote_persona;
 nome  | cognome |  codice_fiscale
-------+---------+------------------
 Luca  | Ferrari | FRRLCU78L19F257B
 Diego | Ferrari | FRRDGI78L19F257B
```

Introduzione

E' un modulo *Multicorn* che permette di esplorare il filesystem locale come fosse una tabella.
Per installare il pacchetto Multicorn occorre usare `gmake`.

Fase 1: creazione del server

```
# CREATE EXTENSION multicorn;

# CREATE SERVER filesystem_server
  FOREIGN DATA WRAPPER multicorn
  OPTIONS ( wrapper 'multicorn.fsfdw.FilesystemFdw' );
```

Fase 2: creare la tabella esterna

```
# CREATE FOREIGN TABLE posts(
   posted_on text,
   short_title text,
   content text, --bytea per file binari
   full_file_name text )
  SERVER filesystem_server
  OPTIONS( root_dir '/home/luca/fluca1978.github.io/_posts/',
           pattern '{posted_on}-{short_title}.md',
           content_column 'content',
           filename_column 'full_file_name' );
```

NOTA: non posso usare una data concreta per `posted_on` perché alcuni post hanno dei nomi errati che confondono il `pattern`!

Fase 3: vedere la tabella

```
# SELECT posted_on, full_file_name FROM posts LIMIT 3;
    posted_on     |      full_file_name
------------------+--------------------------
 2017-09-20-Perl6 | 2017-09-20-Perl6-IO.md
 2017-09-21       | 2017-09-21-KDE3.5.10.md
 2017-07-13       | 2017-07-13-HelloWorld.md
```

Introduzione

Il data wrapper `postgres_fdw` consente di leggere un0istanza PostgreSQL remota (è stato uno degli ultimi FDW sviluppati...).
I passi per collegare un altro database PostgreSQL sono i seguenti (come amministratore del database):

1. installare l'estensione `postgres_fdw` (nativa);
2. creare il server remoto con la relativa stringa di connessione;
3. creare uno *user mapping* per identificare con quale ruolo si andrà ad operare sul server remoto;
4. creare le tabelle remote (ognuna deve essere *mappata*).

Istanze di prova

Si supponga di avere l'istanza principale che contiene il database `testdb`, mentre l'istanza remota (in esecuzione sulla porta 5433 contiene il database `anagraficadb` con la tabella `persona` che si vuole importare nel cluster principale.

Fase 1: caricare FDW

```
# CREATE EXTENSION postgres_fdw;
```

Fase 2: creare il server remoto

```
# CREATE SERVER another_pg
  FOREIGN DATA WRAPPER postgres_fdw
  OPTIONS ( host 'localhost',
            port '5433',
            dbname 'anagraficadb' );
```

Fase 3: creazione di uno user mapping

```
# CREATE USER MAPPING FOR CURRENT_USER
  SERVER another_pg
  OPTIONS( user 'luca',
           password 'xxxx' );
```

Questo *mappa* l'utente corrente sulla prima istanza come utente `luca` sull'istanza remota.

Fase 4: mappare le tabelle

```
# CREATE FOREIGN TABLE remote_persona (
   pk SERIAL,
   nome text,
   cognome text,
   codice_fiscale text NOT NULL )
   SERVER another_pg
   OPTIONS ( schema_name 'public',
             table_name 'persona' );
```

La dichiarazione della tabella deve rispecchiare esattamente quella della tabella remota!

Tuttavia i vincoli di `PRIMARY KEY` e `UNIQUE` non vanno replicati!

Fase 5: vedere il risultato

```
# \d remote_persona
                                  Foreign table "public.remote_persona"
     Column      |  Type   |                              Modifiers
-----------------+---------+-----------------------------------------------------
 pk              | integer | not null default nextval('remote_persona_pk_seq'::regcl
 nome            | text    |
 cognome         | text    |
 codice_fiscale | text    | not null
Server: another_pg
FDW Options: (schema_name 'public', table_name 'persona')
```

Interrogare la tabella remota

```
# SELECT * FROM remote_persona;
 pk | nome  | cognome |  codice_fiscale
----+-------+---------+------------------
  1 | Luca  | Ferrari | FRRLCU78L19F257B
  2 | Diego | Ferrari | FRRDGI78L19F257B

# EXPLAIN SELECT * FROM remote_persona;
                                QUERY PLAN
-------------------------------------------------------------------------
 Foreign Scan on remote_persona  (cost=100.00..129.80 rows=660 width=100)
```

Join fra tabella remota e locale

Si immagini di avere una tabella di indirizzi locale:

```
# CREATE TABLE address( pk SERIAL PRIMARY KEY,
                        persona_pk integer NOT NULL,
                        address text );
# INSERT INTO address( persona_pk, address )
  VALUES( 1, 'Formigine' );
# INSERT INTO address( persona_pk, address )
  VALUES( 2, 'Modena' );
```

e di voler fare il join con la tabella remota delle persone:

```
# SELECT nome, cognome, address
  FROM remote_persona rp JOIN address a ON a.persona_pk = rp.pk
  ORDER BY cognome, nome;
 nome  | cognome |  address
-------+---------+-----------
 Diego | Ferrari | Modena
 Luca  | Ferrari | Formigine
```

Join fra tabella remota e locale: piano di esecuzione

Le tabelle sono piccole: *merge join*!

```
# EXPLAIN SELECT nome, cognome, address
  FROM remote_persona rp JOIN address a ON a.persona_pk = rp.pk
  ORDER BY cognome, nome;
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------
 Sort  (cost=678.84..692.19 rows=5340 width=96)
   Sort Key: rp.cognome, rp.nome
   ->  Merge Join  (cost=263.67..348.22 rows=5340 width=96)
         Merge Cond: (rp.pk = a.persona_pk)
         ->  Sort  (cost=180.30..182.52 rows=890 width=68)
               Sort Key: rp.pk
               ->  Foreign Scan on remote_persona rp  (cost=100.00..136.70 rows=890
         ->  Sort  (cost=83.37..86.37 rows=1200 width=36)
               Sort Key: a.persona_pk
               ->  Seq Scan on address a  (cost=0.00..22.00 rows=1200 width=36)
```

Inserimento di un nuovo record

Siccome la sequenza è locale, non si riesce ad inserire il record per chiave duplicata:

```
# INSERT INTO remote_persona( nome, cognome, codice_fiscale )
  VALUES( 'Emanuela', 'Santunione', 'SNTMNL11M11A944U' );
ERROR:  duplicate key value violates unique constraint "persona_pkey"
DETAIL:  Key (pk)=(1) already exists.
STATEMENT:  INSERT INTO public.persona(pk, nome, cognome, codice_fiscale) VALUES ($
ERROR:  duplicate key value violates unique constraint "persona_pkey"
DETAIL:  Key (pk)=(1) already exists.
CONTEXT:  Remote SQL command: INSERT INTO public.persona(pk, nome, cognome, codice_
```

Inserimento di un nuovo record: prima soluzione

Un possibile *workaround* consiste nel definire la tabella locale come sottoinsieme della tabella remote, escludendo i campi *autogenerati*. Questo potrebbe rompere alcuni vincoli (ad esempio il join con address che va riprogettato): le tabelle remote devono avere delle chiavi naturali e le chiavi surrogate possono essere escluse il piu' delle volte quando si *importano* localmente.

```
# DROP FOREIGN TABLE remote_persona;
# CREATE FOREIGN TABLE remote_persona (
  -- pk SERIAL,
  nome text,
  cognome text,
  codice_fiscale text NOT NULL )
  SERVER another_pg
  OPTIONS ( schema_name 'public', table_name 'persona' );
```

e adesso

```
# INSERT INTO remote_persona( nome, cognome, codice_fiscale )
  VALUES( 'Emanuela', 'Santunione', 'SNTMNL11M11A944U' );

INSERT 0 1
```

Inserimento di un nuovo record: seconda soluzione

Occorre in qualche modo esportare la sequenza remota da usare come valore per la tabella esterna. Sulla seconda instanza (remota) occorre definire una tabella che fornisca il valore della sequenza (questo perché è possibile esportare solo tabelle!):

```
> CREATE VIEW persona_seq_wrapper
  AS SELECT nextval( 'persona_pk_seq'::regclass ) AS val;
```

poi sul server locale si importa la vista remota:

```
# CREATE FOREIGN TABLE remote_persona_seq ( val integer )
  SERVER another_pg
  OPTIONS ( schema_name 'public',
            table_name 'persona_seq_wrapper' );
```

Inserimento di un nuovo record: seconda soluzione (2)

Sul server locale si definisce una funzione che preleva un valore dalla foreign table, che a sua volta preleva un valore dalla sequenza sul server remoto:

```
# CREATE FUNCTION remote_persona_seq_nextval()
  RETURNS integer
  AS $BODY$ SELECT val FROM remote_persona_seq; $BODY$
  LANGUAGE SQL;
```

ora si può definire la tabella remota assegnando come valore di default quello ritornato dalla funzione appena creata:

```
# CREATE FOREIGN TABLE remote_persona (
  pk integer DEFAULT remote_persona_seq_nextval(),
  nome text,
  cognome text,
  codice_fiscale text NOT NULL )
  SERVER another_pg
  OPTIONS ( schema_name 'public', table_name 'persona' );
```

Inserimento di un nuovo record: seconda soluzione (3)

A qeusto punto è possibile inserire un nuovo record nella tabella:

```
# INSERT INTO remote_persona( nome, cognome, codice_fiscale )
  VALUES( 'Emanuela', 'Santunione', 'SNTMNL11M11A944K' );
```

ATTENZIONE: in questo caso viene aperto un cursore per leggere la tabella remota, che in realtà legge la sequenza remota, ne consegue che si ha un calo di performance che in questo caso non giustifica il lavoro extra.

Foreign keys & foreign tables

Le `FOREIGN TABLE` non sono delle tabelle vere e proprie, quindi non possono supportare tutti i vincoli di una tabella regolare:

```
# ALTER TABLE address
   ADD FOREIGN KEY (persona_pk)
   REFERENCES remote_persona(pk);
ERROR:  referenced relation "remote_persona" is not a table
```

Introduzione

Il modulo `GitFdw` è un *Multicorn* e dipende dal modulo Python `brigit`.

Fase 1: creazione del server

```
# CREATE SERVER git_server
  FOREIGN DATA WRAPPER multicorn
  OPTIONS ( wrapper 'multicorn.gitfdw.GitFdw' );
```

Fase 2: creazione di una tabella

```
# CREATE FOREIGN TABLE github(
   author_name text,
   author_email text,
   message text,
   hash text,
   date date )
 SERVER git_server
 OPTIONS( path '/home/luca/fluca1978.github.io' );
```

path è l'unica opzione disponibile per questo tipo di wrapper! Il repository deve essere locale (è forse una limitazione di `brigit`?)

Fase 3: interrogare la tabella

```
# SELECT substring( hash from 1 for 10 ),
         date, message
  FROM github LIMIT 2;

 substring  |    date    |                message
------------+------------+---------------------------------------
 426dd99ac3 | 2017-11-04 | Fix some typos in a post
 868980d676 | 2017-11-02 | FreeBSD and PostgreSQL rc blog post.
```

Introduzione

Il `GoogleFdw` è un modulo *Multicorn*.

Fase 1: definizione del server

```
# CREATE SERVER google_server
  FOREIGN DATA WRAPPER multicorn
  OPTIONS( wrapper 'multicorn.googlefdw.GoogleFdw' );
```

Fase 2: creazione della tabella

```
# CREATE FOREIGN TABLE google_search(
    url text,
    title text,
    search text )
SERVER google_server;
```

Fase 3: effettuare una ricerca

Non funziona! Vedere: issue 199.

```
SELECT * FROM google_search WHERE search = 'postgresql';
ERROR:  Error in python: TypeError
DETAIL:  'NoneType' object has no attribute '__getitem__'
```

Installazione

Occorre avere le librerie SQLite3 installate e clonare un repository git:

```
% sudo pkg install sqlite3-3.22.0_2
% git clone https://github.com/gleu/sqlite_fdw.git
% cd sqlite_fdw
% gmake
% sudo gmake install
```

E' un wrapper in sola lettura!

Fase 1: definizione del server

```
# CREATE EXTENSION sqlite_fdw;
# CREATE SERVER digikam_sqlite_server
    FOREIGN DATA WRAPPER sqlite_fdw
    OPTIONS( database '/home/luca/digikam4.db' );
```

Fase 2: creazione della tabella

```
# CREATE FOREIGN TABLE digikam_albums(
    id INTEGER ,
    albumRoot INTEGER NOT NULL,
    relativePath TEXT NOT NULL,
    date DATE,
    caption TEXT,
    collection TEXT,
    icon INTEGER  )
  SERVER digikam_sqlite_server
  OPTIONS( table 'Albums' );
```

Fase 3: effettuare una ricerca

```
# SELECT relativePath, date FROM digikam_albums ORDER BY date DESC LIMIT 3;

        relativepath         |    date
-----------------------------+------------
 /                           | 2018-02-15
 /2018_FESTE                 | 2018-02-15
 /2018_FESTE/2018_CARNEVALE | 2018-02-11
```

pg_config

Il comando shell pg_config fornisce molte informazioni circa l'installazione corrente di PostgreSQL: dove si trovano i file eseguibili, le directory di configurazione, gli header, ecc.
Senza alcun argomento pg_config mostra tutte le opzioni di compilazione e installazione di PostgreSQL (output molto verbose).
Con specifiche opzioni il comando mostra come è stato compilato PostgreSQL. Ad esempio:

```
# con quali lib è stato linkato PostgreSQL?
% /usr/local/bin/pg_config --libs
-lpgcommon -lpgport -lintl -lssl -lcrypto -lz -lreadline -lcrypt -lm  -lpthread

# dove sono gli eseguibili?
% /usr/local/bin/pg_config --bindir
/usr/local/bin

# dove si trova il makefile per pgxs?
% /usr/local/bin/pg_config --pgxs
/usr/local/lib/postgresql/pgxs/src/makefiles/pgxs.mk
```

pg_controldata

Il comando pg_controldata fornisce lo stato attuale del cluster, con diverse informazioni utili quali lo stato dei WAL, la versione del catalogo di sistema, la versione del cluster, ecc. E' solitamente molto utile quando si deve riportare un bug:

```
% sudo  pg_controldata -D /mnt/data1/pgdata
pg_control version number:            1002
Catalog version number:               201707211
Database system identifier:           6509403694562149197
Database cluster state:               in production
pg_control last modified:             Mon Jan 29 13:38:59 2018
Latest checkpoint location:           0/B6000028
Prior checkpoint location:            0/B5000028
Latest checkpoint's REDO location:    0/B6000028
Latest checkpoint's REDO WAL file:    000000010000000000000B6
Latest checkpoint's TimeLineID:       1
Latest checkpoint's PrevTimeLineID:   1
Latest checkpoint's full_page_writes: on
...
Time of latest checkpoint:            Tue Jan 16 13:51:30 2018
...
wal_level setting:                    minimal
...
Database block size:                  8192
```

postgresql.conf

Il file principale di configurazione è `postgresql.conf`, contenuto solitamente in `$PGDATA` (alcune distribuzioni lo inseriscono nell'albero `/etc`).
Il file contiene dei parametri in formato `chiave = valore` con valori booleani (`on`, `off`), numerici, stringa.
Il carattere # identifica un commento.

Estendere `postgresql.conf`

E' possibile usare delle direttive per *spezzare* la configurazione su piu' file:

- `include` permette di includere un file specifico;
- `include_if_exists` include un file solo se esiste;
- `include_dir` include tutti i file della directory specificata (è importante nominarli correttamente per avere un ordine naturale di caricamento).

L'ultima definizione di un valore trovato nella catena degli include è quello a cui viene assegnato!

In questo modo è possibile gestire configurazioni multiple agilmente.

Estendere postgresql.conf: esempio

In postgresql.conf:

```
work_mem = 4MB
include_if_exists = 'conf.d/mem.conf'
```

e in conf.d/mem.conf:

```
work_mem = 16MB
```

L'ultima definizione vince:

```
# SELECT name, setting, category, sourcefile, sourceline
  FROM pg_settings WHERE name like 'work_mem%';
-[ RECORD 1 ]---------------------------------
name       | work_mem
setting    | 16384
category   | Resource Usage / Memory
sourcefile | /mnt/data1/pgdata/conf.d/mem.conf
sourceline | 1
```

Estendere postgresql.conf: contro-esempio

Se si inverte l'ordine di caricamento delle variabili l'ultima definizione è quella che *vince*:

```
# postgresql.conf
include_if_exists = 'conf.d/mem.conf'
work_mem = 4MB
```

```
# SELECT name, setting, category, sourcefile, sourceline
  FROM pg_settings WHERE name like 'work_mem%';
-[ RECORD 1 ]--------------------------------
name       | work_mem
setting    | 4096
category   | Resource Usage / Memory
sourcefile | /mnt/data1/pgdata/postgresql.conf
sourceline | 123
```

I valori *scartati* possono essere individuati con pg_file_settings.

Vedere la configurazione a run-time: SHOW

Il comando speciale SHOW consente di vedere un parametro a run-time:

```
> SHOW shared_buffers;
 shared_buffers
----------------
 256MB
```

Vedere la configurazione a run-time: pg_settings

La vista speciale pg_settings consente di valutare ogni singolo parametro con i relativi valori di default, minimo, massimo, descrizione, ecc.

```
# SELECT name, setting, min_val, max_val, unit,
  sourcefile, sourceline,
  short_desc,
  pending_restart,
  context
  FROM pg_settings
  WHERE name like 'shared_buffer%';
-[ RECORD 1 ]---+-----------------------------------------------------
name            | shared_buffers
setting         | 32768
min_val         | 16
max_val         | 1073741823
unit            | 8kB
sourcefile      |
sourceline      |
short_desc      | Sets the number of shared memory buffers used by the server.
pending_restart | f
context         | postmaster
```

Contesti di configurazione

Ogni parametro ha un *contesto* di configurazione che ne determina l'istante di modifica (`pg_settings.context`):

- `internal` modificabile solo ricompilando il server;
- `postmaster` modificabile solo riavviando il server;
- `sighup` modificabile inviando un `SIGHUP` al processo postmaster;
- `superuser-backend`, `backend` modificabile a livello di sessione per superutenti o utenti normali rispettivamente (prima dell'inizio della sessione!);
- `superuser`, `user` modificabile da un superutente o utente normale.

SET e pg_settings

Per i parametri che hanno un contesto di sessione è possibile usare il comando SET per impostare il relativo valore, o equivalentemente fare un UPDATE su pg_settings:

```
> SET enable_seqscan TO off;

> UPDATE pg_settings
  SET setting = 'off'
  WHERE name = 'enable_seqscan';
```

Vedere la configurazione a run-time: note su pg_settings

I dati sul file di configurazione (`sourcefile`, `sourceline`) non sono visualizzati per utenti non amministratori. Il flag `pending_restart` indica se il valore è stato modificato nel file di configurazione ma non ancora *riletto* dal sistema (ossia occorre un riavvio). Il valore `unit` indica l'unità di misura del valore corrente `setting`, nell'esempio di cui sopra il valore è impostato a: `8 kB * 16384 = 131072 kB = 128 MB` corrispondente al seguente valore nel file di configurazione:

```
% sudo grep shared_buffers /mnt/data1/pgdata/postgresql.conf
shared_buffers = 128MB
```

Trovare gli errori di configurazione (nei file di configurazione)

La vista speciale `pg_file_settings` consente di individuare eventuali errori nei file di configurazione:

- se `applied` è false allora il parametro è stato ignorato;
- se `error` è true allora il parametro contiene errori (es. valori non ammessi).

Ad esempio, se si fosse definito due volte la variabile `log_directory`:

```
# SELECT *
  FROM pg_file_settings
  WHERE name = 'log_directory';
-[ RECORD 1 ]----------------------------------
sourcefile | /mnt/data1/pgdata/postgresql.conf
sourceline | 361
seqno      | 11
name       | log_directory
setting    | pg_logging
applied    | f
error      |
-[ RECORD 2 ]----------------------------------
sourcefile | /mnt/data1/pgdata/postgresql.conf
sourceline | 362
seqno      | 12
name       | log_directory
```

Non tutti i parametri si trovano in pg_settings, ma in pg_file_settings!

Non tutti i parametri del file di configurazione si trovano in pg_settings ma sicuramente, se sono stati processati, si trovano in pg_file_settings:

```
# SELECT * FROM pg_settings WHERE name = 'datestyle';
(0 rows)

# SELECT * FROM pg_file_settings WHERE name = 'datestyle';

sourcefile | /mnt/data1/pgdata/postgresql.conf
sourceline | 574
seqno      | 20
name       | datestyle
setting    | iso, mdy
applied    | t
error      |
```

Come impostare un parametro di configurazione

Ci sono tre modi principali per impostare un parametro di configurazione:

- *editare* `postgresql.conf`, l'unico metodo che funziona sempre;
- effettuare (come superutente) un `ALTER SYSTEM SET` per il parametro specificato, che scrive il parametro nel file `postgresql.auto.conf` che viene *aggiunto* all'ordinario `postgresql.conf` al prossimo ricaricamento della configurazione o riavvio del server

```
# ALTER SYSTEM SET shared_buffers TO '128MB';
 -- ALTER SYSTEM RESET shared_buffers;
 -- ALTER SYSTEM SET shared_buffers TO DEFAULT;
```

```
% sudo cat /mnt/data1/pgdata/postgresql.auto.conf
# Do not edit this file manually!
# It will be overwritten by ALTER SYSTEM command.
shared_buffers = '128MB'
```

- specificare un flag all'avvio del server.

Ricarica la configurazione a run-time

Ci sono diversi modi per *forzare un refresh della configurazione*:

- restart del server (non molto pratico);
- inviare un SIGHUP al postmaster;

```
% sudo kill HUP 1126
```

- eseguire, come superutente del database, la stored procedure pg_reload_conf;

```
# SELECT pg_reload_conf();
 pg_reload_conf
----------------
 t
```

- eseguire, come owner del processo, pg_ctl che è lo strumento ufficiale per controllare un cluter:

```
% sudo -u postgres pg_ctl reload -D /mnt/data1/pgdata
server signaled
```

- eseguire uno degli script a disposizione del sistema operativo, ad esempio pg_ctlcluster (debian-like):

```
% sudo pg_ctlcluster 9.5 main reload
```

Processi

I seguenti parametri permettono di modificare il nome dei processi in esecuzione:

- `cluster_name` una stringa di massimo 64 bytes che indica ulteriorermente il nome del cluster;
- `update_process_title` riporta nel nome del processo lo stato del processo stesso.

Questi parametri su alcuni sistemi operativi rappresentano un collo di bottiglia (es. FreeBSD).
Ad esempio con questa configurazione:

```
cluster_name = 'pg9.6.5-FreeBSD'
update_process_title = on
```

E una connessione utente viene quindi visualizzata come:

```
02072 postgres postgres: pg9.6.5-FreeBSD: luca testdb 127.0.0.1(21451) idle (postgr
```

e quando l'utente cambia la query il *process title* si aggiorna di conseguenza:

```
02072 postgres postgres: pg9.6.5-FreeBSD: luca testdb 127.0.0.1(21451) VACUUM (post
```

Logging: dove

- `log_destination` indica dove inviare i log del server:
  - `syslog` invia al sistema di logging tramite `syslog_facility`, ma richiede solitamente modifiche al demone *syslogd* per accettare i log;
  - `csv` memorizza i log in modo *Comma Separated Values* (utile per analizzare i log in modo automatico;
  - `stderr` invia i log direttamente allo standard error;
- `log_collector` abilita la cattura dei log tramite un processo *collector* che li invia ai file di log (usato con `stderr` e `csv`);
  - `log_directory` e `log_filename` indica la directory (relativa a `$PGDATA` o assoluta) e il nome di file da usare per i log (come stringa sottoposta a escape come `strftime(2)`);
  - `log_rotation_age` e `log_rotation_size` indicano dopo quanto tempo o dimensione il file di log deve essere ruotato;
- `syslog_ident` e `syslog_facility` indicano la stringa di identity (`postgres`) e il livello di logging verso *syslogd*.

Logging: quando

- `log_min_messages` e `client_min_messages` indica la soglia di logging dei messaggi di log del server e delle connessioni utente;
- `log_min_duration_statement` indica di inserire nei log tutte le query di durata superiore (in millisecondi) superiore a quella stabilita;
- `log_duration` inserisce la durata di ogni statement eseguito (ma non la query eseguita!);
- `log_autovacuum_min_duration` inserisce nei log le azioni di autovacuum che sono durate piu' dei millisecondi specificati.

Logging: cosa

- `log_checkpoints` inserisce una riga di log ad ogni checkpoint;
- `log_connections` e `log_disconnections` inserisce una riga di log per ogni connessione e disconnessione;
- `log_line_prefix` permette di specificare la linea di log con delle sequenze di escape;
- `log_statement` indica se loggare gli statement di *DDL* o di *DML* o tutti (`off`, `ddl`, `mod`, `all`).

Esempio:

```
LOG:  connection received: host=127.0.0.1 port=36828
LOG:  connection authorized: user=postgres database=testdb
```

Parametri di networking

- `listen_address` indica su quale indirizzo IPv4/IPv6 o nome host il server deve attendere connessioni (sistemi multi-home);
- `port` indica la porta su cui rimanere in ascolto di connessioni (default 5432);
- `max_connections` stabilisce il massimo numero di connessioni concorrenti;
- `superuser_reserved_connections` è la parte di `max_connections` che deve essere riservata per il collegamento dei super-utenti;
- `tcp_keepalives_idle`, `tcp_keepalives_interval`, `tcp_keepalives_count` specificano il comportamento dei *TCP KEEPALIVES* specificando dopo quanti secondi di connessione idle deve essere inviato un pacchetto keepalive, dopo quanti secondi deve essere ritrasmesso in caso di mancato acknowledgment e dopo quanti pacchetti keepalive senza risposta la connessione è da considerarsi *morta*;
- `authentication_timeout` numero di secondi per terminare la fase di autenticazione di un client;

shared_buffers

E' uno dei parametri chiave di una istanza.
PostgreSQL mantiene un'area di memoria ove vengono *trasferite* le pagine dati lette da disco. Questo spazio di memoria è denominato *shared buffers*.
Solitamente non va incrementato oltre il 40% della memoria disponibile sulla macchina (dedicata) per consentire anche alla cache del sistema operativo di funzionare, e generalmente lo si tiene al 25% della memoria totale disponibile.

work_mem e maintanance_work_mem

La work_mem specifica quanta memoria viene messa a disposizione di operazioni di *sorting* e *hashing* prima che le query inizi a scrivere i dati su disco (temporaneamente). Va tenuto presente che ogni sessione parallela può usare questa quantità di memoria. Non importa aumentarla oltre 128MB se non per carichi di lavoro intensivi (reporting). Ha spesso senso impostarla a <memoria RAM>/ (max_connections/2).
maintanance_work_mem indica la memoria assegnata ai comandi di gestione dei dati, quali ad esempio VACUUM, CREATE INDEX. Anche qui il parametro indica la memoria usabile per ogni sessione in parallelo, anche se una sessione può eseguire solo uno di tali comandi in un dato momento. Inoltre con *autovacuum* attivo, la memoria usata può diventare moltiplicata per autovacuum_max_workers. Generalmente la si imposta a 1/32 della memoria totale.

Parametri relativi a VACUUM e ANALYZE

Siccome `VACUUM` e `ANALYZE` sono operazioni intense per l'I/O, i seguenti parametri cercano di contenere l'impatto:

- `vacuum_cost_limit` indica il *punteggio* di soglia per un `VACUUM` affinché si sospenda il processo per un certo periodo;
- `vacuum_cost_delay` (multipli di 10 ms) indica per quanto tempo il processo di `VACUUM` (manuale) sarà interrotto al raggiungimento della soglia di costo;
- `vacuum_cost_page_hit`, `vacuum_cost_page_miss`, `vacuum_cost_page_dirty` indicano rispettivamente il punteggio di costo di una pagina trovata nello shared buffers, non trovata e modificata (mentre prima era pulita);
- `vacuum_freeze_table_age` indica a quale età il `VACUUM` deve percorrere anche le pagine dati con tuple non congelate;
- `vacuum_freeze_min_age` numero di transazioni prima di forzare `VACUUM` a percorrere anche le pagine dati con tuple non congelate.

Autovacuum

- `autovacuum` indica se il demone di autovacuum deve essere abilitato (anche se impostato ad `off` un autovacuum sarà lanciato per evitare lo xid wraparound);
- `autovacuum_max_workers` indica il numero di processi di autovacuum che possono essere al massimo attivi contemporaneamente (oltre all'autovacuum launcher);
- `autovacuum_work_mem` indica la memoria da usare per ogni *worker* di autovacuum. Se vale `-1` indica di usare `maintanance_work_mem`;
- `autovacuum_naptime` indica il tempo di pausa fra un ciclo di autovacuum e l'altro;
- `autovacuum_vacuum_threshold` e `autovacuum_analyze_threshold` indicano il numero di tuple aggiunte/modificate prima di lanciare un processo di `VACUUM` o `ANALYZE` su quella determinata tabella;
- `autovacuum_vacuum_scale_factor` e `autovacuum_analyze_scale_factor` indicano una percentuale di dimensione della tabella da aggiungere a ai relativi parametri di

Background Worker

Il processo *Background Worker* è in carico di prelevare le pagine *sporche* (modificate) nello shared buffers e renderle persistenti su disco. Il background worker esegue ciclicamente scaricando le *Least Recently Used* pagine su disco, stimando anche quanti nuovi buffer sono necessari (deve fare spazio per nuovi dati). I parametri principali sono:

- `bgwriter_delay` indica, in multipli di 10 ms, di quanto sospendere il Background Writer fra un ciclo e un altro.
- `bgwriter_lru_maxpgaes` indica quante pagine modificate il background worker deve scaricare ad ogni ciclo;
- `bgwriter_lru_multiplier` indica di quanto moltiplicare il numero di pagine stimate come *nuove* per il prossimo ciclo;
- `bgwriter_flush_after` indica di forzare un *flush* del sistema operativo dopo che sono stati scritti dal Background Writer questi bytes di dati;
- `effective_io_concurrency` indica quanti processi di I/O paralleli possono essere generati al massimo. Ad esempio su un RAID-0 si può aumentare questo valore al numero di dischi che compongono l'array.

WAL Levels

Il parametro `wal_level` stabilisce il tipo di funzionamento per l'archiviazione dei WAL. In base al livello sono aggiunte piu' o meno informazioni nei WAL. I valori possibili sono:

- `minimal` scrive il numero di dati minimo, necessari solo per riprendersi da un crash;
- `replica` scrive informazioni sufficienti per il ripristino dei dati da parte di un *base backup*;
- `logical` aggiunge informazioni per il logical decoding.

Fsync etc.

Il parametro `fsync` stabilisce come rendere persistenti le modifiche. Disabilitarlo è altamente rischioso e andrebbe fatto solo quando si può ripristinare completamente il database senza perdita di dati (es. caricamento iniziale).
Il parametro `wal_sync_method` è comandato solo da `fsync = on` e indica quale metodo si deve usare per forzare il flush dei dati dei WAL. Il parametro di default è `fsync` e comunque i valori dipendono dal sistema operativo e dalla system calls messe a disposizione. Il metodo `fdatasync` è disponibile su Linux per evitare aggiornamenti dei metadati.
Il parametro `syncrhonous_commit` indica come viene gestito il commit di una transazione. Questo parametro può essere impostato per ogni singola transazione e non produce perdita di consistenza. I valori possibili sono:

- `on` (default) si attende che il sistema operativo dia conferma della scrittura (non significa che i dati siano persistenti!);
- `off` non si attende nessuna conferma, e si possono perdere transazioni per un certo periodo (3 volte il delay del WAL writer);
- `remote_apply` e `remote_write` sono usati con la replica sincrona e

Parametri dei WAL

- `full_page_writes` indica se ad ogni checkpoint una pagina modifica va scritta interamente nei WAL, aumenta la sicurezza di crash recovery al costo di maggior spazio di archiviazione;
- `wal_compression` abilita la compressione delle *full pages* ad ogni checkpoint;
- `commit_delay` indica il tempo di delay (microsecondi) per un commit, in modo da raggruppare transazioni tutte assieme;
- `wal_buffers` indica la quantita di memoria dello *shared buffers* (default 1/32) usabili per le pagine WAL ancora da scaricare su disco, utile per transazioni che committano contemporaneamente;
- `wal_writer_delay` indica quanti millisecondi attendere fra un flush verso il sistema operativo dei WAL e il successivo;
- `wal_writer_flush_after` indica dopo quanti dati forzare un flush verso il sistema operativo.

Checkpoints

- `checkpoint_timeout` numero di secondi prima di forzare un checkpoint automatico;
- `max_wal_size` dimensione dei dati prima di forzare un checkpoint automatico;
- `checkpoint_completion_target` indica il target di completamento del checkpoint per il throttling;
- `min_wal_size` impedisce a `pg_wal` di scendere al di sotto di tale dimensione, così da garantire una occupazione minima costante.

In particolare il sistema cerca di completare il checkpoint in un tempo pari a: `checkpoint_completion_target * checkpoint_timeout` e quindi valori prossimi a 1 per `checkpoint_completion_target` *rallentano* l'attività di checkpointing lasciando il database abile di rispondere ad altre query. Tuttavia, all'aumentare del `checkpoint_completion_target` aumentano anche i WAL da conservare (perché il checkpoint impiega piu' tempo a completare).

Metodi di accesso ai dati

I seguenti parametri abilitano o disabilitano la gestione di alcuni metodi di accesso ai dati: `enable_bitmapscan`, `enable_hashagg`, `enable_hashjoin`, `enable_indexonlyscan`, `enable_indexscan`, `enable_material`, `enable_mergejoin`, `enable_nestloop`, `enable_seqscan`, `enable_sort`, `enable_tidscan`

Costi dell'ottimizzatore

I costi dell'ottimizzatore sono in valori arbitrari e relativi: quello che conta è solo la loro differenza. I parametri sono:

- `seq_page_cost` costo per il recupero di una pagina dati in modalità sequenziale;
- `random_page_cost` costo per l'accesso non sequenziale ad una pagina dati, tramite indice (`4` per disco meccanico, `1.5` per disco SSD);
- `cpu_tuple_cost`, `cpu_index_tuple_cost`, `cpu_operator_cost` indicano il costo di *analisi* di una tupla, indice e operatore;
- `default_statistics_target` il valore di default per il campionamento di *Most Common Values* e degli istogrammi;
- `constraint_exclusion` permette di sfruttare le *constraint* di tabella per ottimizzare le query.

pgbench

Il sistema include un comando denominato `pgbench` che aiuta ad eseguire una serie di transazioni (anche definite dall'utente) su un determinato database al fine di calcolare le *Transaction Per Second (TPS)* e quindi valutare differenti scelte di configurazione.
Pgbench installa una manciata di tabelle con alcune tuple nel database specificato ed esegue alcuni test.

Scaling factor, clients e tipo di test

Ci sono alcune opzioni importanti quando si usa `pgbench`:

- *scaling factor* (`-s`): usabile solo in inizializzazione per indicare quanto "grossi" devono essere i dati (crescita lineare);
- *fill factor* (`-F`): usabile solo in inizializzazione per indica la percentuale di utilizzo delle tabelle create (default 100%);
- *number of clientes* (`-c`) usato in benchmarking, indica quanti client devono essere simulati (in default 1);
- *protocol* (`-M`): usato in benchmarking, indica se inviare le query al server usando prepared statements o meno.

Fase 1: inizializzazione

```
% pgbench -i -s 10 -h localhost -U luca testdb
creating tables...
100000 of 1000000 tuples (10%) done (elapsed 0.14 s, remaining 1.26 s)
200000 of 1000000 tuples (20%) done (elapsed 0.43 s, remaining 1.71 s)
300000 of 1000000 tuples (30%) done (elapsed 1.73 s, remaining 4.03 s)
400000 of 1000000 tuples (40%) done (elapsed 2.54 s, remaining 3.81 s)
500000 of 1000000 tuples (50%) done (elapsed 3.23 s, remaining 3.23 s)
600000 of 1000000 tuples (60%) done (elapsed 3.82 s, remaining 2.55 s)
700000 of 1000000 tuples (70%) done (elapsed 4.55 s, remaining 1.95 s)
800000 of 1000000 tuples (80%) done (elapsed 5.58 s, remaining 1.39 s)
900000 of 1000000 tuples (90%) done (elapsed 6.01 s, remaining 0.67 s)
1000000 of 1000000 tuples (100%) done (elapsed 7.18 s, remaining 0.00 s)
vacuum...
set primary keys...
done.
```

Fase 2: benchmarking

```
% pgbench  -h localhost -U luca -c 10 -M prepared testdb
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 10
query mode: prepared
number of clients: 10
number of threads: 1
number of transactions per client: 10
number of transactions actually processed: 100/100
latency average = 16.586 ms
tps = 602.921759 (including connections establishing)
tps = 616.187365 (excluding connections establishing)
```

Custom scripts

L'utente può creare i propri script di test da far eseguire a `pgbench`.
All'interno dello script occorre inserire le eventuali definizioni di variabili (`\set`) e la transazione da eseguire.
L'esecuzione dello script corrisponde ad una transazione.
Vengono fornite da `pgbench` due variabili speciali passate agli script custom:

- `:scale` lo scaling factor;
- `:client_id` un numero intero (che parte da zero) che identifica univocamente un client simulato.

Custom script: esempio (sbagliato)

```
 \set eta random(:scale , 50)
 \set anno random( 11, 99 )
 \set giorno random( 10, 50 )
BEGIN;
  INSERT INTO persona( nome, cognome, codice_fiscale, eta )
  VALUES( 'NomeTest', 'CognomeTest', 'TTTzzz' || :anno || 'A' || :giorno || 'XXXW',
END;
```

Custom script: esempio (sbagliato) e sua esecuzione

```
% pgbench  -h localhost -U luca -c 10 -f my_bench.sql testdb
starting vacuum...end.
client 4 aborted in state 4: ERROR:  duplicate key value violates unique constraint
DETAIL:  Key (codice_fiscale)=(TTTzzz96A37XXXW) already exists.
client 3 aborted in state 4: ERROR:  duplicate key value violates unique constraint
DETAIL:  Key (codice_fiscale)=(TTTzzz70A45XXXW) already exists.
transaction type: my_bench.sql
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 1
number of transactions per client: 10
number of transactions actually processed: 98/100
latency average = 19.566 ms
tps = 511.090134 (including connections establishing)
tps = 518.212526 (excluding connections establishing)
```

Siccome due client hanno fallito entrambi una transazione, e quindi in tutto sono fallite due transazioni, il sistema riporta che 98/100 transazioni sono state attualmente processate!

Introduzione a pgTAP

L'estensione `pgTAP` consente di caricare in un database una serie di funzioni di utilita' per *testing* di un database. L'idea ricalca quella dell'infrastruttura di test di Perl e utilizza il protocollo *TAP* (Test Anything Protocol).

Installazione di pgTAP

Mediante pgxn è possibile scaricare e installare il pacchetto:

```
% sudo pgxn install pgtap

INFO: best version: pgtap 0.98.0
...
```

e nel database che si vuole testare è possibile creare l'estensione:

```
> CREATE EXTENSION pgtap;
```

Verificare il funzionamento di pgTAP

E' possibile verificare il corretto funzionamento di pgTAP mediante la funzione pass(), che se eseguita senza un piano fallisce miseramente (ma almeno conferma l'installazione dell'estensione):

```
> SELECT pass( 'Hello World' );
ERROR:  You tried to run a test without a plan! Gotta have a plan
CONTEXT:  PL/pgSQL function _get_latest(text) line 10 at RAISE
PL/pgSQL function _todo() line 9 at assignment
PL/pgSQL function ok(boolean,text) line 9 at assignment
```

Principali funzioni messe a disposizioni da pgTAP (1)

Le funzioni utilizzabili in un test sono:

- `plan()` indica quanti test si andranno ad eseguire;
- `ok()` considera un valore booleano che deve essere *vero*;
- `is()` e `isnt()` controlla due valori fra loro che devono essere rispettivamente uguali o diversi;
- `matches()`, `imatches()`, `doesnt_match()` effettua una comparazione fra una stringa e una regexp (eventuale case insensitive);
- `alike()`, `ialike()`, `unalike()`, `unialike()` utilizza l'operatore `LIKE` eventualmente in modalità case insensitive;
- `cmp_ok()` confronta due valori mediante l'operatore specificato;
- `pass()` e `fail()` indicano rispettivamente il successo o l'insuccesso del test;
- `isa_ok()` controlla che un tipo sia quello specificato.

Principali funzioni messe a disposizione da pgTAP (2)

Per controllare la presenza di tabelle e altri oggetti:

- `has_tablespace()` e `hasnt_tablespace()`;
- `has_schema()` e `hasnt_schema()`;
- `has_function()` e `hasnt_function()`;
- `has_relation()` e `hasnt_relation()` che si specializzano in una delle seguenti:
  - `has_table()`, `hasnt_table()`;
  - `has_view()`, `hasnt_view()`;
  - `has_foreign_table()`, `hasnt_foreign_table()`;
  - `has_index()`, `hasnt_index()`;
  - `has_sequence()`, `hasnt_sequence()`
- `has_role()` e `hasnt_role()`;
- `has_language()` e `hasnt_language()`;
- `has_extension()` e `hasnt_extension()`.

Esempio di script di test

Anzitutto lo script è bene che visualizzi solo alcune informazioni (ad esempio rimuovendo gli header delle tabelle/funzioni). Inoltre è opportuno che tutte le funzioni siano eseguite in una transazione per dare modo a pgTAP di eseguire il controllo sul plan di esecuzione. Quindi un semplice script può essere:

```
-- parte fissa per fornire un output
-- conforme a TAP
\set ECHO none
\set QUIET 1
\pset format unaligned
\pset tuples_only true
\pset pager

-- test in una transazione
BEGIN;
SELECT plan( 1 ); -- quanti test?
SELECT has_role( 'luca', 'Ricerca utente LUCA' );
SELECT * FROM finish(); -- indica la fine del test
ROLLBACK; -- sempre un rollback di questa transazione
```

Esecuzione dello script di esempio

E' sufficiente lanciare l'interpretazione dello script da un terminale SQL per vedere il risultato come TAP:

```
> \i test.sql
1..1
ok 1 - Ricerca utente LUCA
```

Eseguire lo script di esempio da riga di comando

Senza entrare nel terminale `psql` è possibile lanciare il comando (sempre via `psql`) direttamente dalla shell corrente:

```
% psql -h localhost -U luca -X -f test.sql testdb
1..1
ok 1 - Ricerca utente LUCA
```

L'opzione `-f` indica di leggere ed eseguire il contenuto del file `test.sql`, mentre l'opzione `-X` indica di non caricare il file `.psqlrc`.

Eseguire lo script di esemprio da riga di comando: pg_prove

Esiste un comando simile a prove di Perl, denominato pg_prove che viene installato da un modulo CPAN:

```
% sudo cpan TAP::Parser::SourceHandler::pgTAP
```

mediante pg_prove è possibile eseguire una intera suite di test in un colpo solo:

```
% pg_prove -h localhost -U luca -d testdb test.sql
test.sql .. ok
All tests successful.
Files=1, Tests=1,  1 wallclock secs ( 0.02 usr  0.03 sys +  0.00 cusr  0.02 csys =
Result: PASS
```

Un esempio di test piu' complesso

Molte delle funzioni di test accettano lo schema e opzionalmente la descrizione del test, se lo schema è omesso si considera lo schema corrente, se la descrizione è omessa non viene stampato nulla di particolare in caso di fallimento.

```
BEGIN;
SELECT plan( 4 );

SELECT has_role( 'luca', 'Ricerca utente LUCA' );
SELECT has_table( 'public', 'persona', 'Tabella persona' );
SELECT has_column( 'public', 'persona',
                   'pk', 'Chiave surrogata persona' );
-- senza schema
SELECT has_column( 'persona', 'cognome',
                   'Colonna cognome tabella persona' );

Select * FROM finish();
ROLLBACK;
```

che produce come risultato:

```
% pg_prove -d testdb -h localhost -U luca test.sql
test.sql .. ok
All tests successful.
```

Esempio di fallimento di test

Usando pg_prove:

```
% pg_prove -d testdb -h localhost -U luca test.sql
test.sql .. 1/5
# Failed test 5: "Colonna indirizzo tabella persona"
# Looks like you failed 1 test of 5
...
Result: FAIL
```

mentre usando direttamente i test:

```
% psql -h localhost -U luca -X -f test.sql testdb
1..5
...
not ok 5 - Colonna indirizzo tabella persona
# Failed test 5: "Colonna indirizzo tabella persona"
# Looks like you failed 1 test of 5
```

Organizzare i test

In maniera analoga a quanto si fa con Perl e `prove` è possibile organizzare i test in directory e farli eseguire tutti in un colpo a `pg_prove`. Si noti che `pg_prove` non richiede il setup del terminale `psql` consigliato nei passi precedenti.
Ad esempio, supponendo di avere una directory `t/` che contiene degli script SQL con i vari test:

```
% pg_prove -d testdb -h localhost -U luca t/*.sql
t/evento.sql ... ok
t/persona.sql .. ok
All tests successful.
Files=2, Tests=7,  1 wallclock secs ( 0.02 usr  0.02 sys +  0.00 cusr  0.01 csys =
Result: PASS
```

Introduzione

PostgreSQL supporta diversi tipi di indici, ognuno specializzato per un algoritmo di *ordinamento* dei dati. E' possibile specificare quale indice usare al momento della creazione.
Un indice può essere multicolonna (fino a 32 colonne per indice).
L'indice userà la *collation* della colonna a cui si applica, ma è possibile specificarne una differente all'atto della creazione dell'indice. Ad ogni modo un indice usa una sola collation!

Cosa si può indicizzare?

PostgreSQL permette di indicizzare:

- colonne (fino a 32 per indice);
- espressioni di colonne;
- dati parziali.

Le espressioni di colonne sono calcolate ad ogni `INSERT` / `UPDATE` ma non ad ogni scansione dell'indice.
Gli indici parziali sono costruiti attorno a particolari valori dei dati (ad esempio `eta >= 18`). L'idea è quella di non indicizzare i valori piu' comuni perché non sarebbero comunque usati (l'indice non è filtrante su valori comuni!). Gli indici parziali possono anche essere `UNIQUE`: le colonne indicizzate devono essere `UNIQUE`, il predicato viene fatto su altre colonne.

Heap, Indice secondario, accessi

In PostgreSQL gli indici sono memorizzati separatamente dalla tabella:

- l'indice è formalmente detto *secondario*;
- lo spazio di memorizzazione della tabella è detto *heap*.

Questo significa che un accesso attraverso un indice richiede:

1. il caricamento e la valutazione (scan) delle pagine dati dell'indice;
2. il caricamento e l'accesso alla pagina dati della tabella individuata dall'indice.

Questo significa che l'accesso tramite indice può essere *costoso* in termini di risorse e I/O (random access). Per questo motivo un indice viene usato solo se il suo utilizzo non aggiunge penalità, altrimenti un accesso sequenziale all'heap viene preferito.

B-Tree

E' il tipo di default. Sono gli unici indici che possono essere definiti `UNIQUE`! Supporta l'*Index Only Scan*!
Funziona per operazioni di equalità e range su dati ordinabili in qualche modo, quindi operatori =, >, <, >=, <=, `BETWEEN`, `IN`, `IS NULL`.
Possono anche essere usati per ricerche `LIKE` a patto che:

- il pattern sia un literal;
- il pattern sia cercato all'inizio della stringa.

Ad esempio `LIKE 'Ferr%'`.
Nel caso di indici multicolonna l'indice viene usato in base a delle limitazioni: il fattore di filtro delle prime colonne con uguaglianza e la prima con disuguaglianza identificano la porzione di indice da "caricare in memoria e valutare", le colonne a seguire sono comunque usate ma non riducono la porzione di dati dell'indice da controllare.

Hash

Gli indici *Hash* sono usabili solo per uguaglianza, quindi non saranno mai utilizzati se non assieme all'operatore =.
ATTENZIONE: gli indici hash non sono attualmente salvati nei WAL.
Questo significa che gli indici hash non sopravvivono ad un crash e non sono nemmeno replicati (dalla version 10 entrambi i problemi sono risolti).

GIN

Sono indici *invertiti*, ovvero invece che "puntare" ad una tupla puntano a dati multipli (es. array).
Sono usati solitamente per testare l'esistenza della chiave fra i possibili valori, ad esempio nell Full Text Search.

BRIN

Gli indici *BRIN* (Block Range INdexes) memorizzano dati sommari circa ogni blocco fisico di dati su disco. Ad esempio, se i dati sono ordinabili, il BRIN memorizza il valore minimo e massimo di ogni *blocco fisico* di dati.

GIST

GIST (Generalized Index Search Tree) non è un vero tipo di indice ma una "piattaforma" per la costruzione di indici personalizzati.
Tali indici vengono costruiti definendo gli operatori applicabili e il relativo comportamento (ad esempio indici spaziali).

SP-GIST

E' una infrastruttura per la costruzione di indici non bilanciati.

Quando sicuramente non viene usato un indice

Una query che non filtri i dati non userà mai un indice.
Si vogliono reperire tutti i dati, che senso avrebbe caricare in memoria prima l'indice per poi accederli tutti?

ORDER BY

Nel caso di un `ORDER BY` è possibile sia utilizzato l'indice, a patto che:

- le colonne dell'indice comprendano quelle dell'`ORDER BY`;
- la tabella sia molto piccola, nel qual caso l'I/O per un sort esterno risulterebbe piu' costoso che la visita dell'indice.

In questo caso i B-Tree sono utili: memorizzano i dati in ordine *ascendente* con i `NULL` alla fine.
Un altro caso in cui si "forza" l'uso dell'indice è `ORDER BY ... LIMIT`, poiché l'indice solitamente è piu' veloce nel restituire le prime n-tuple di `LIMIT`.

Agggregazione di indici

PostgreSQL è capace di unire piu' indici al fine di ottimizzare l'esecuzione della query.
Solitamente la query viene "smontata" in piu' piani di esecuzione, ciascuno colelgato ad un indice (a volte anche allo stesso con clausole diverse).
Dopo aver percorso un indice il sistema costruisce una bitmap che contiene la mappa di match trovati su quella condizione/indice. Le varie bitmap (una per indice) sono poi aggregate logicamente con `AND` o `OR` per costruire la mappa finale delle tuple che hanno passato ogni indice.
La mappa finale è ordinata naturalmente per posizione *fisica* delle tuple, quindi se c'è un `ORDER BY` sarà necessario un ulteriore passaggio di sorting.

Quali sono le tabelle che potenzialmente richiedono un indice?

La vista `pg_stat_user_tables` può fornire delle indicazione sull'utilizzo delle tabelle e dei relativi indici. Ove ci sono dei *sequential scan* elevati si può ipotizzare sia necessario un indice:

```
> SELECT relname, seq_scan, idx_scan,
         n_tup_ins, n_tup_upd, n_tup_del,
         n_live_tup, n_dead_tup, last_vacuum
  FROM pg_stat_user_tables
  ORDER BY seq_scan DESC;
       relname          | seq_scan | idx_scan | n_tup_ins | n_tup_upd | n_tup_del |
------------------------+----------+----------+-----------+-----------+-----------+
 usr_local_etc_snapshot |       11 |          |         0 |         0 |         0 |
 log_table              |        4 |        0 |        22 |         0 |         0 |
 persona                |        2 |        0 |         0 |         0 |         0 |
 address                |        1 |        0 |         2 |         0 |         0 |
 pgbench_accounts       |        1 |        0 |   1000000 |         0 |         0 |
 pgbench_branches       |        1 |        0 |        10 |         0 |         0 |
 pgbench_tellers        |        1 |        0 |       100 |         0 |         0 |
 evento                 |        1 |        0 |   2110336 |         0 |         0 |
 persona_femmina        |        0 |          |      5000 |         0 |         0 |
 persona_maschio        |        0 |          |      5001 |         0 |         0 |
```

Qual'è lo stato degli indici?

La vista speciale `pg_user_stat_indexes` fornisce indicazioni sull'utilizzo degli indici:

```
> SELECT relname, indexrelname,
         idx_scan, idx_tup_read, idx_tup_fetch
   FROM pg_stat_user_indexes;
     relname      |         indexrelname          | idx_scan | idx_tup_read | idx_tup_
------------------+-------------------------------+----------+--------------+---------
 address          | address_pkey                  |        0 |            0 |
 evento           | idx_desc                      |        2 |            1 |
 evento           | evento_pkey                   |        0 |            0 |
```

Reset delle statisitche

Con la funzione speciale `pg_stats_reset()` (da eseguire da superutente) le statistiche `pg_stat_xxxx` sono azzerate!

Introduzione

`EXPLAIN` è un comando di utilita' che consente di interagire con l'ottimizzatore.
PostgreSQL usa un ottimizzatore basato sul *costo*: la query viene spezzata in un albero di *nodi* da eseguire, ogni nodo avrà un costo e la somma dei costi fornisce il costo della query stessa. Esistono piu' *percorsi* per estrarre i dati da una query (es. con o senza indice) e il piano di esecuzione che risulta avere il costo inferiore è quello che PostgreSQL sceglie per l'esecuzione.
Il costo di un nodo/piano è espresso in una misura aleatoria relativa e fine a se stessa. Talve valore dipende fortemente dal tempo di esecuzione ma anche dall'impiego delle risorse.

EXPLAIN fornisce un solo piano di uscita

Il comando EXPLAIN visualizza il piano migliore scelto fra quelli possibili, e di conseguenza NON indica perché un indice non viene usato, NON suggerisce la riscrittura di query per ottimizzare il lavoro, NON indica come ottenere risultati migliori.
Occorre prendere dimestichezza con lo strumento e fare dei tentativi per migliorare una query lenta!
Esiste anche il modulo `autoexplain` che consente di inserire nei log automaticamente un `EXPLAIN` delle query che sono durate piu' di un certo tempo.

EXPLAIN e ANALYZE

Il comando `EXPLAIN` fornisce indicazione sul piano che sarà usato per fare l'accesso alle tuple.
Il comando `EXPLAIN` non effettua la query, simula solo l'accesso ai dati.
Può essere accompagnato da `ANALYZE` per eseguire effettivamente la query e aggiornare anche le statistiche di sistema.
Solitamente si vuole usare `EXPLAIN ANALYZE` a meno che ci siano effetti sui dati o la query esegua per troppo/infinito tempo. *ATTENZIONE: siccome `EXPLAIN ANALYZE` eseguirà la query, se questa modifica le tuple (es. `INSERT`, `UPDATE`, `DELETE`) è bene inserire il blocco in una transazione e fare un rollback!*

Output di EXPLAIN

Il comando EXPLAIN fornisce come output una serie di nodi, ciascuno che indica:

- *tipo di accesso* ossia come vengono percorsi i dati;
- *costo* in unità imparziali, diviso in costo iniziale (setup) e finale (ossia per il completamento del nodo);
- *numero di tuple* quante tuple sono estratte da ogni nodo;
- *dimensione delle tuple* dimensione dei dati estratti.

Esempio di EXPLAIN

Si consideri:

```
>  EXPLAIN SELECT * FROM persona WHERE eta > 48;
                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on persona  (cost=0.00..219.00 rows=175 width=46)
   Filter: (eta > 48)
```

C'è un solo nodo, di tipo Seq Scan (accesso sequenziale) che ha un costo iniziale nullo (non è richiesto alcun setup) e costo finale di 219. Saranno estratte 175 tuple, ciascuna di 46 bytes.

Esempio di EXPLAIN ANALYZE

Con l'aggiunta di ANALYZE viene effettivamente eseguita la query e si può verificare se le statistiche erano aggiornate: la query riporta effettivo tempo di esecuzione e dati sulle tuple realmente trovate.

```
>  EXPLAIN ANALYZE SELECT * FROM persona WHERE eta > 48;
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------
 Seq Scan on persona  (cost=0.00..219.00 rows=175 width=46)
                      (actual time=0.027..3.886 rows=175 loops=1)
   Filter: (eta > 48)
   Rows Removed by Filter: 9825
 Planning time: 0.098 ms
 Execution time: 4.174 ms
```

La query non effettua un ANALYZE!

Formati di output di EXPLAIN

E' possibile ottenere diversi formati di output di EXPLAIN mediante l'opzione FORMAT. Tali formati possono essere usati per una migliore interpretazione da parte di programmi automatici e includono:

- *text* output di default;
- *json*;
- *xml*;
- *yaml*.

```
> EXPLAIN ( FORMAT JSON )
  SELECT * FROM persona WHERE eta > 48;
              QUERY PLAN
-----------------------------------
 [                                 +
   {                               +
     "Plan": {                     +
       "Node Type": "Seq Scan",    +
       "Relation Name": "persona",+
       "Alias": "persona",         +
       "Startup Cost": 0.00,       +
       "Total Cost": 219.00,       +
       "Plan Rows": 175,           +
       "Plan Width": 46,           +
```

Buffers di EXPLAIN

Il comando EXPLAIN ANALYZE accetta l'opzione particolare buffers che consente di indicare dove sono state trovate le informazioni:

```
> EXPLAIN (ANALYZE, BUFFERS)
   SELECT * FROM evento WHERE description like '%10%';
                                                        QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on evento  (cost=0.00..39806.00 rows=107668 width=18) (actual time=0.122.
   Filter: (description ~~ '%10%'::text)
   Rows Removed by Filter: 1996586
   Buffers: shared hit=64 read=13371
 Planning time: 1.679 ms
 Execution time: 285.588 ms
```

Si ha che 64 buffers erano già in memoria e 13371 sono stati riletti da disco.

Effetti collaterali di un sistema in produzione

Quando si esegue un `EXPLAIN ANALYZE` si deve tenere in considerazione che alcuni dati potrebbero essere già presenti o assenti dallo shared buffer, e questo dipende dal resto delle query che stanno lavorando sul sistema. Ad esempio se si esegue ricorsivamente la query di explain precedente (ad esempio con `\watch`) si ottiene:

```
> EXPLAIN (ANALYZE, BUFFERS)
  SELECT * FROM evento WHERE description like '%10%';
                                                        QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on evento  (cost=0.00..39806.00 rows=107668 width=18) (actual time=0.000.
   Filter: (description ~~ '%10%'::text)
   Rows Removed by Filter: 1996586
   Buffers: shared hit=13435
 Planning time: 0.085 ms
 Execution time: 242.981 ms
```

ossia tutti i dati sono già in memoria! Si noti che il tempo è calato, anche se non drasticamente.

Verbosità di EXPLAIN

L'opzione verbose consente di visualizzare alcuni dati in piu' in uscita, come ad esempio il nome delle colonne restituite da ogni nodo e il nome completo di ogni relazione in alias (utile ad esempio con i join).

Nodi annidati in EXPLAIN

```
> EXPLAIN
  SELECT * FROM persona
  WHERE eta > 48 AND sesso IN (
      SELECT sesso FROM persona
      GROUP BY sesso HAVING COUNT(*) > 20 );
                                          QUERY PLAN
-----------------------------------------------------------------------------
 Hash Semi Join  (cost=244.07..465.48 rows=175 width=46)
   Hash Cond: (persona.sesso = persona_1.sesso)
   ->  Seq Scan on persona  (cost=0.00..219.00 rows=175 width=46)
         Filter: (eta > 48)
   ->  Hash  (cost=244.05..244.05 rows=2 width=2)
         ->  HashAggregate  (cost=244.00..244.03 rows=2 width=2)
               Group Key: persona_1.sesso
               Filter: (count(*) > 20)
               ->  Seq Scan on persona persona_1  (cost=0.00..194.00 rows=10000 wid
```

Output di EXPLAIN: come si legge?

Tendenzialmente si legge da destra verso sinistra, dal basso verso l'alto; i nodi con costo minori sono eseguiti per primi. Nell'esempio di prima si parte con un `Seq Scan` su `persona_1` (costo nullo), query inner. Successivamente viene eseguito un `HashAggregate` che implementa il `GROUP BY`; tutto questo viene racchiuso in un nodo `Hash` che fornisce due tuple e che ha un costo di 244. Parallelamente si esegue un `Seq Scan` sulla tabella `persona` (outer) per ottnere un `Hash` sulla condizione di `eta`. Infine i due hash sono uniti in un `Semi Join`, si noti che il costo di partenza di questo nodo è 244 pari al massimo costo dei due nodi intermedi (un nodo non può partire prima dei suoi figli).

Sequential Scan (Seq Scan)

E' il tipo di scansione di default (ossia quando non vi sono indici o il loro utilizzo è inutile e/o costoso).
Viene letta la tabella una pagina dati alla volta fino alla fine.

Bitmap Index Scan (`Bitmap Index Scan`)

Viene usato quando si hanno dei valori non troppo frequenti e delle clausole di uguaglianza.
Si costruisce una mappa hash di tuple che soddisfano o meno la condizione e si ripercorre questa condizione quando si leggono le pagine dati.

Index Scan (`Index Scan`)

Viene usato quando si ha un indice altamente filtrante (ossia pochi valori da estrarre).
E' l'accesso classico con indice:

1. si percorre l'indice;
2. si estraggono le tuple visibili dall'heap della tabella.

Index Only Scan (`Index Only Scan`)

E' una funzionalità introdotta nelle ultime versione di PostgreSQL. Viene usata se:

- l'indice supporta l'*Index Only Scan* (es. B-Tree);
- la query referenzia solo colonne presenti nell'indice.

Tutti i dati sono quindi accessibili solo tramite l'indice, e quindi è possibile accedere all'indice direttamente senza accedere all'heap.
C'è però una complicazione: gli indici non memorizzano i dati di visibilità, quindi l'indice potrebbe referenziare tuple invisibili (MVCC). Per risolvere questo problema ogni pagina heap memorizza un bit che indica se tutte le sue tuple sono visibili. Se non tutte le tuple sono visibili questo accesso si trasforma di fatto in un *Index Scan*.

Nested Loop (Nested Loop)

E' il meccanismo classico di join. La tabella di destra viene percorsa con due strategie:

- *inner* `Seq Scan`
  1. la tabella *outer* (sinistra) viene percorsa sequenzialmente;
  2. la tabella *inner* (destra) viene percorsa sequenzialmente per ogni valore della tabella inner. Funziona solo per piccole dimensioni della *inner*.
- *inner* `Index Scan`
  1. la tabella *outer* (sinistra) viene letta sequenzialmente (o anche con indice);
  2. per ogni valore della *outer* si cerca nell'indice della *inner* (destra) il match e si accede tramite l'indice alle tuple.

Altri tipi di join

- `Hash Join`
  1. la tabella *inner* (destra) viene decomposta in un hash con valori multipli per bucket;
  2. la tabella *outer* (sinistra) viene percorsa sequenzialmente, per ogni valore si calcola la chiave e si cerca nell'array di valori hash della *inner*
- `Merge Join`
  1. entrambe le tabelle sono prima ordinate (`Sort`) a seguito di una lettura sequenziale (`Seq Scan`);
  2. la tabella *outer* (sinistra) viene percorsa sequenzialmente;
  3. per ogni valore della *outer* si estraggono tutti i valori della *inner* che fanno match (sono ordinati);
  4. al primo valore della *outer* che non fa match si avanza al prossimo valore della *inner* e si riparte dalla stessa posizione della *inner*.
- `Lateral Join` viene usato quando ci sono delle espressioni da calcolare dinamicamente;
- `Semi Join` e `Anti Join` sono dei join *parziali* usati per verificare l'esistenza o meno di chiavi fra le tabelle, ad esempio clausole `EXISTS` e `IN` e relative negazioni.

Aggregazioni

Quando si usa una funzione di aggregazione si possono incontrare i seguenti nodi:

- `GroupAggregate` è il caso classico con `GROUP BY`;
- `HashAggregate` raggruppamento in memoria mediante una tabella hash;
- `WindowAgg` usato quando si ha una window function.

Nodi *di servizio*

Ci sono una serie di nodi per le operazioni principali:

- `Sort` usato per il sorting esplicito o implicito;
- `Limit` quando compare una clausola `LIMIT`;
- `Unique` usato per `DISTINCT` oppure `UNION`.

Nodi per l'unione di query

- `SubqueryScan` effettua il join fra una sottoquery e quella principale;
- `CTEScan` effettua il join fra una CTE e la query principale;
- `Materialize` crea un recordset in memoria partendo da una porzione di risultati di un nodo;
- `Append` usata con `UNION ALL`.

Dove sono le statistische?

Il catalogo `pg_statistic` contiene le informazioni statistiche su ogni attributo di ogni tabella, ad sempio:

```
# SELECT pc.relname, ps.stanullfrac, ps.stadistinct, pa.attname, pa.attstattarget
  FROM pg_class pc JOIN pg_statistic ps ON ps.starelid = pc.oid
  JOIN pg_attribute pa ON pa.attnum = ps.staattnum
  WHERE pc.relname = 'software' AND pa.attrelid = pc.oid;
 relname  | stanullfrac | stadistinct | attname | attstattarget
----------+-------------+-------------+---------+---------------
 software |           0 |          -1 | pk      |            -1
 software |           0 |       -0.25 | name    |            -1
 software |           0 |          -1 | version |            -1
 software |           0 |       -0.25 | valid   |            -1
```

che mostra i valori null e il numero di valori distinti per le varie colonne.

I valori delle statistiche

I valori delle statistiche *solitamente* seguono questa regola:

- se sono valori positivi allora sono valori esatti;
- se sono valori negativi hanno un significato speciale.

Ad esempio:

```
 relname  | stanullfrac | stadistinct | attname | attstattarget
----------+-------------+-------------+---------+---------------
 software |           0 |          -1 | pk      |            -1
 software |           0 |       -0.25 | name    |            -1
```

- `stadistinct` indica il numero esatto di valori distinti o un valore negativo che il moltiplicatore del numero di tuple, ovvero `name` ha valori distinti ogni 4 righe (`-0.25`), mentre `pk` ogni riga (è chiave!);
- `attstattarget` indica la granularilarità delle statistiche collezionate da `ANALYZE`. Il valore dipende dal dominio del dato:
  - se negativo indica che si usa il default per quel tipo di dato;
  - se positivo indica che si userà quel valore;
  - se zero allora nessuna statistica sarà raccolta su quella colonna.

Per valori scalari solitamente il target indica quanti *valori comuni devono essere considerati*.

ANALYZE

PostgreSQL aggiorna il catalogo delle statistiche con il comando `ANALYZE` (che può essere eseguito da solo o assieme a `VACUUM`).

```
# ANALYZE VERBOSE software;
INFO:  analyzing "public.software"
INFO:  "software": scanned 1 of 1 pages,
       containing 8 live rows and 0 dead rows;
       8 rows in sample, 8 estimated total rows
```

Il livello di statistiche (granularità) da raccogliere con `ANALYZE` può essere specificato colonna per colonna con:

```
> ALTER TABLE software
  ALTER COLUMN name
  SET STATISTICS 200;
```

Tale valore viene definito statistic target e rappresenta il massimo numero di valori memorizzabili nell'istogramma e nei Most Common Values di `pg_statistic` per quella colonna.

Tuple e pagine dati

Il catalogo `pg_class` contiene il numero di tuple e pagine dati di una relazione, questo viene usato dall'ottimizzatore per stimare la quantità di lavoro da fare. Il numero di pagine viene controllato rispetto a quello di `pg_class` (è un'operazione non costosa e non richiede I/O) e se i due valori non coincidono si aggiorna il numero delle tuple di conseguenza.

```
> SELECT relpages, reltuples
  FROM pg_class
  WHERE relname = 'persona';
 relpages | reltuples
----------+-----------
       94 |     10000
```

Statistiche piu' comode

La vista speciale pg_stats sul catalogo pg_statistic (e tabelle collegate) fornisce una visione piu' "comoda" di accesso alle statistiche:

```
> SELECT null_frac,
  n_distinct,
  most_common_vals,
  most_common_freqs,
  histogram_bounds
  FROM pg_stats
  WHERE tablename = 'persona'
  AND attname IN ( 'sesso', 'nome' );
-[ RECORD 1 ]-
null_frac          | 0
n_distinct         | -1
most_common_vals   |
most_common_freqs  |
histogram_bounds   | {Nome1,Nome1087,Nome1177,Nome1267,Nome1357,Nome1447,
                     Nome1537,Nome1627,Nome1717,Nome1807,Nome1898,Nome1988,
                     Nome2077,Nome2167,Nome2257,Nome2347,Nome2437,Nome2527,
                     Nome2617,Nome2707,Nome2798,Nome2888,Nome2978,
                     Nome3067, ... }
-[ RECORD 2 ]-
null_frac          | 0
n_distinct         | 2
```

Analizzare MCVs e MCFs in una tabella

E' possibile utilizzare un doppio cast, `unnest` e `rows from` per ottenere una tabella di correlazione fra un valore e la sua frequenza:

```
> SELECT  mcv, mcf
  FROM pg_stats,
   ROWS FROM ( unnest( most_common_vals::text::text[] ),
               unnest( most_common_freqs ) )
             r( mcv, mcf )
  WHERE tablename = 'evento'
  AND attname IN ( 'description' );
```

Query di esempio

Si supponga di avere la tabella persona popolata con 10000 nomi a caso, come nell'esempio di partizionamento.

```
> EXPLAIN
  SELECT *
  FROM persona
  WHERE sesso = 'M'
  AND nome < 'Nome1177';
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on persona  (cost=0.00..244.00 rows=100 width=46)
   Filter: ((nome < 'Nome1177'::text) AND (sesso = 'M'::bpchar))
```

Il sistema indica che verranno estratte 100 tuple, ma come fa a saperlo?

Calcolare la selettività

Quando viene effettuata una query l'ottimizzatore cerca di comprendere quale sia la selettività di una clausola `WHERE`. In particolare per ogni clausola si esamina l'*operatore* nel catalogo di sistema `pg_operator`. In `pg_operator` si ha che:

- `oprname` corrisponde al nome dell'operatore;
- `oprkind` vale `b` per operatori infissi, `l` per unari a sinistra, `r` per unari a destra;
- `oprrest` indica la funzione di *restrizione di selettività* dell'operatore;

. `oprleft`, `oprright`, `oprresult` indicano it `pg_type` tipo dell'operando o risultato.

```
> SELECT oprname, oprkind, oprleft, oprright, oprrest
   FROM pg_operator
   WHERE ( oprname = '<' OR oprname = '=' )
   AND oprleft = ( SELECT oid
                   FROM pg_type
                   WHERE typname = 'char' )
   AND oprright = oprleft;
-[ RECORD 1 ]---------
oprname  | =
oprkind  | b
```

Most Common Values (MCVs)

La funzione `eqsel` controlla i valori piu' comuni per le colonne specificate, ossia `most_common_vals` e `most_common_freqs`. Dai valori si ottiene che nel caso della colonna `sesso`:

```
most_common_vals  | {F,M}
most_common_freqs | {0.5,0.5}
```

ossia la selettività della colonna su uno dei due valori è del 50% (`0.5`).

Histograms

Nel caso della colonna `nome`, e quindi della funzione `scalarltsel`, i valori da usare sono in un *istogramma* che riporta dei "bucket" di valori ipotizzandoli a distribuzione costante (ossia a pari frequenza):

```
histogram_bounds  | {Nome1,Nome1087,Nome1177,Nome1267,Nome1357,Nome1447,
                     Nome1537,Nome1627,Nome1717,Nome1807,Nome1898,Nome1988,
                     Nome2077,Nome2167,Nome2257,Nome2347,Nome2437,Nome2527,
                     Nome2617,Nome2707,Nome2798,Nome2888,Nome2978,
                     Nome3067, ... }
```

L'istogramma divide i valori in *bucket* con pari frequenza.
In particolare il valore di filtro della clausola `WHERE ... nome < 'Nome1177'` individua appunto il terzo bucket, quindi ci sono 2 bucket che soddisfano la condizione. I bucket in totale sono 100 (*statistic target*), quindi la selettività è data da `2/100 = 0.02`.

Selettività complessiva

La selettività complessiva della clausola in `AND` logico è data dalla moltiplicazione dei valori di selettività individuale, ovvero:

```
10000 tuple * ( 0.5 * 0.02 ) = 100
```

che è appunto il risultato fornito dal comando `EXPLAIN`.

Altro esempio analogo

Si supponga di variare la query come segue:

```
> EXPLAIN  SELECT * FROM persona WHERE sesso = 'F' AND nome like 'Nome1%';
                            QUERY PLAN
------------------------------------------------------------------
 Seq Scan on persona  (cost=0.00..244.00 rows=556 width=42)
   Filter: ((nome ~~ 'Nome1%'::text) AND (sesso = 'F'::bpchar))
```

Il procedimento è lo stesso descritto in precedenza, e la selettività ora risulta:

- 0.5 per la colonna sesso;
- 12/100 = 0.12 per la colonna nome;

ovvero 10000 tuple * 0.5 * 0.12 = 600

Controesempio

Si supponga di usare la query:

```
> EXPLAIN
   SELECT *
   FROM persona
   WHERE sesso <> 'F'
   AND nome > 'Nome1177';
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on persona  (cost=0.00..244.00 rows=4900 width=46)
   Filter: ((sesso <> 'F'::bpchar) AND (nome > 'Nome1177'::text))
```

che è l'esatto opposto della precedente. Gli operatori hanno funzioni di selettività differente che ragionano in modo opposto:

```
 SELECT oprname, oprkind, oprleft, oprright, oprrest
   FROM pg_operator
   WHERE ( oprname = '>' OR oprname = '<>' )
   AND oprleft = ( SELECT oid
                   FROM pg_type
                   WHERE typname = 'char' )
   AND oprright = oprleft;
-[ RECORD 1 ]---------
oprname  | <>
oprkind  | b
oprleft  | 18
```

Calcolare la selettività complessiva

Le selettività risultano ora date da:

- *la selettività restante nel caso di* `sesso`, ovvero `1 - 0.5 = 0.5`;
- *la selettività restante nel caso di* `nome`, ovvero `(100 - 2 ) bucket / 100 = 0.98`.

La selettività complessiva è data dalla composizione delle selettività individuali, quindi il numero delle tuple può essere calcolato come:

```
10000 tuple * ( 0.5 * 0.98 ) = 4900
```

che è appunto il risultato fornito da `EXPLAIN`.

Caso analogo ma con valori interi

Aggiungiamo una età random (compresa fra 0 e 50 anni):

```
> ALTER TABLE persona
   ADD COLUMN eta integer
   DEFAULT mod( (random() * 1000)::int, 50 );

> ANALYZE persona;
```

Le statistiche di sistema riportano:

```
> SELECT * FROM pg_stats
  WHERE tablename = 'persona'
  AND attname = 'eta';
-[ RECORD 1 ]--
...
null_frac              | 0
avg_width              | 4
n_distinct             | 50
most_common_vals       | {32,8,28,15,14,38,22,16,37,41,
                         19,26,25,12,34,24,4,20,13,17,3,
                         44,2,7,36,48,30,31,39,0,6,45,
                         33,43,1,11,46,35,5,40,18,27,
                         21,47,42,29,10,9,49,23}
most_common_freqs      | {0.0222,0.0219,0.0218,0.0217,0.0216,
                         0.0216,0.0214,0.0213,0.0213,0.0212,
                          ... }
```

Query di esempio

```
>  EXPLAIN SELECT *
    FROM persona
    WHERE eta = 28;
                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on persona  (cost=0.00..219.00 rows=218 width=46)
   Filter: (eta = 28)
```

Calcolo della selettività

Il valore 28 è uno dei valori frequenti per la colonna `eta`, listato come terzo valore piu' frequente nei `most_common_vals`, in particolare la sua frequenza è di `0.0218` in `most_common_freqs`.
A questo punto l'ottimizzatore assume che la selettività della condizione sia di `0.0128`, e quindi le tuple in uscita siano date dal calcolo:
`10000 * 0.0218 = 218`

Ulteriore esempio

```
 EXPLAIN SELECT * FROM persona WHERE eta < 48;
                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on persona  (cost=0.00..219.00 rows=9623 width=46)
   Filter: (eta < 48)
```

Calcolo della selettività

Ci sono due bucket che soddisfano la condizione di diseguaglianza:

- 48 con frequenza `0.0202`;
- 49 con frequenza `0.0175`.

La frequenza dei bucket che rimangono (e che soddisfano la condizione) è quindi data dalla somma dei relativi valori di frequenza, o anche dalla differenza di quelli sopra che non soddsifano la condizione, ossia:

```
10000 tuple * ( 1 - ( 0.0202 + 0.0175 ) ) = 9623
```

Controesempio

```
>  EXPLAIN SELECT * FROM persona WHERE eta > 48;
                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on persona  (cost=0.00..219.00 rows=175 width=46)
   Filter: (eta > 48)
```

C'è un solo bucket che verifica la condizione `eta > 48`, ossia quello con 49 e frequenza `0.0175`, e quindi le tuple in uscita sono:

```
10000 tuple * 0.0175 = 175
```

Calcolo della selettività: esempio piu' complesso

```
> EXPLAIN SELECT *
   FROM persona
   WHERE eta > 48
   AND sesso = 'M'
   AND nome > 'Nome1177';
                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on persona  (cost=0.00..269.00 rows=86 width=46)
   Filter: ((eta > 48) AND (nome > 'Nome1177'::text) AND (sesso = 'M'::bpchar))
```

Le singole selettività di filtro sono:

- 0.0175 per la colonna età;
- 0.5 per la colonna sesso;
- 0.98 per la colonna nome;

```
10000 tuple * ( 0.0175 * 0.5 * 0.98 ) = 85.75 = 86
```

Selettività: riassunto

L'ottimizzatore esegue alcuni passaggi fondamentali per comprendere quanto una clausola `WHERE` sia selettiva sui dati:

1. controlla da `pg_stat` quante sono le tuple e quante le pagine dati, se i due numeri non combcaciano si aggiusta il numero delle tuple;
2. affida alla funzione stabilita dall'operatore (`pg_operator`) il compito di restituire l'indice di selettività:
   - nel caso di *uguaglianza* si controllano i *Most Common Values*:
     - se il valore è fra quelli comuni la selettività è data dalla frequenza del valore;
     - se il valore non è fra quelli comuni si sommano le frequenze dei MCVs che soddisfano la condizione, tale somma è la selettività (lineare);
   - nel caso di disigueglianza si controlla l'*istogramma* dei valori, supporti a frequenza costante:
     - si cerca di capire in quale/i bucket cade il valore;
     - se cade in un solo bucket si calcola l'andamento lineare: ampiezza del bucket rispetto al numero di bucket totali;
     - se cade in piu' bucket si calcola la percentuale di bucket colpiti.

Introduzione

Con il termine *statistic collector* si intende una parte di PostgreSQL dedicata a monitorare l'attività, intesa come accesso per numero di tuple, tabelle, indici, attività utente, ecc.
Le statistche sono visibili tramite una serie di viste speciali, denominate `pg_stat_xxx` e sono mantenute in file temporanei all'interno di `pg_stat_tmp`, per poi essere consolidate allo shutdown in `pg_stat` (questo garantisce la possibilita' di mantenere le statisiche fra avvi differenti del server). In caso di crash le statistische sono azzerate!

Configurazione dello statistic collector

In `postgresql.conf` vi sono una serie di parametri `track_xxx` che abilitano o disabilitano la collezione delle statistiche. Tali parametri possono anche essere impostati a livello di sessione (ma solo per i superutenti, per evitare utilizzi maliziosi). I parametri sono:

- `track_activities` informazioni sulle attività utente correntemente in atto (visibile solo dal proprietario e dai superutenti);
- `tack_counts` informazioni sulle attività di database, necessario per autovacuum;
- `tack_functions` informazioni sulle chiamate di funzione (bisogna specificare quale tipo di linguaggio va tracciato, es `pl`);
- `track_io_timing` informazioni sulle operazioni di I/O, molto "costoso" e solitamente disabilitato (può servire per monitorare nuovo hardware o configurazioni aggressive).

Interazione con lo statistic collector

Occorre tenere presente che le statistiche non sono aggiornate in *real time*:

- vengono aggiornate ogni 500 ms;
- ogni processo invia le proprie statistiche al collector solo quando diventa *idle*;
- le statistiche sono *congelate* all'interno di una stessa transazione.

In particolare l'ultimo punto significa che una transazione che interroghi le statistiche vedrà gli stessi dati statistici per tutta la durata della transazione, essendo quindi abilitata a effettuare correlazioni fra i dati senza "paura" che questi cambino.

Vedere cosa fanno gli utenti: pg_stat_activity

La vista pg_stat_activity contiene una tupla per ogni processo di backend, consentendo di vedere cosa i singoli processi stanno facendo:

```
 > SELECT usename, client_addr, application_name,
          backend_start, query_start,
          state, backend_xid, query
   FROM pg_stat_activity;
-[ RECORD 1 ]----+---------------------------------
usename          | luca
client_addr      | 127.0.0.1
application_name | psql
backend_start    | 2017-11-13 09:45:38.589523+01
query_start      | 2017-11-13 11:09:09.484301+01
state            | active
backend_xid      |
query            | FETCH FORWARD 20 FROM _psql_cursor
```

Vedere come stanno i database: pg_stat_database

La vista pg_stat_database fornisce una indicazione di massima sull'utilizzo dei database:

```
> SELECT datname, xact_commit, xact_rollback, blks_read,
         tup_fetched, tup_inserted, tup_updated, tup_deleted
FROM pg_stat_database;

datname       | testdb
xact_commit   | 1600
xact_rollback | 13
blks_read     | 111456
tup_fetched   | 27087
tup_inserted  | 3121466
tup_updated   | 242
tup_deleted   | 119
```

In particolare se il rapporto fra xact_committed e xact_rollback si avvicina all'uguaglianza (o xact_rollback supera il numero di xact_committed è probabile che ci sia un errore applicativo o una query entrata in loop.

Vedere come stanno i database: pg_stat_database_conflicts

Questa vistra mostra le indicazioni circa eventuali conflitti nei database: lock in attesa, deadlocks e conflitti su snapshot.

Informazioni sulle tabelle: pg_stat_user_tables

Le viste speciali pg_stat_user_tables, pg_stat_sys_tables e l'unione pg_stat_all_tables forniscono dati sull'utilizzo delle tabelle utente, di sistema (catalogo) o complessive:

```
> SELECT relname, seq_scan, idx_scan,
         n_tup_ins, n_tup_del, n_tup_upd, n_tup_hot_upd,
         n_live_tup, n_dead_tup,
         last_vacuum, last_autovacuum,
         last_analyze, last_autoanalyze
         FROM pg_stat_user_tables;
```

```
relname           | evento
seq_scan          | 3
idx_scan          | 2
n_tup_ins         | 2110336
n_tup_del         | 0
n_tup_upd         | 0
n_tup_hot_upd     | 0
n_live_tup        | 2110336
n_dead_tup        | 0
last_vacuum       |
last_autovacuum   |
last_analyze      | 2017-11-13 09:54:49.332696+01
last_autoanalyze  | 2017-11-08 14:13:15.982416+01
```

Informazioni sugli indici: pg_stat_user_indexes

Le viste speciali pg_stat_user_indexes, pg_stat_sys_indexes e l'unione pg_stat_all_indexes forniscono informazioni sull'utilizzo degli indici utente, di sistema e complessivi:

```
> SELECT relname, indexrelname,
         idx_scan, idx_tup_read, idx_tup_fetch
         FROM pg_stat_user_indexes;
```

```
relname       | evento
indexrelname  | idx_desc
idx_scan      | 2
idx_tup_read  | 1
idx_tup_fetch | 1
```

Si noti che idx_tup_read sono le tuple di indice lette, mentre idx_tup_fetch sono le tuple di relazione estratte.

Informazioni di transazione: pg_stat_xact_user_tables

Le viste speciali `pg_stat_xact_user_tables`, `pg_stat_xact_sys_tables` e `pg_stat_xact_all_tables` sono strutturate in maniera identica a `pg_stat_user_tables` e similari, ma considerano le statistiche solo per la transazione corrente.
Sono quindi da considerarsi un sottoinsieme limitato alla transazione corrente delle relative informazioni statistiche di tabella.

Informazioni sulle funzioni: pg_stat_user_functions

La vista speciale pg_stat_user_functions fonrisce indicazioni circa le *chiamate* alle funzioni utente:

```
> SELECT funcname, calls, total_time, self_time
   FROM pg_stat_user_functions;
 funcname | calls | total_time | self_time
----------+-------+------------+-----------
 somma    |     2 |       0.43 |      0.43
```

Il self_time è il tempo speso solo nel codice della funzione, il total_time include il tempo speso anche nelle altre funzioni eventualmente richiamate.

Informazioni di I/O: pg_statio_xxx

Le viste speciali pg_statio_user_tables, pg_statio_user_indexes e le similari versioni di sistema e complessive, forniscono informazioni sulle attività di input/output:

```
> SELECT relname, heap_blks_read, heap_blks_hit,
         idx_blks_read, idx_blks_hit
   FROM pg_statio_user_tables;
-[ RECORD 1 ]--+----------------------
relname        | evento
heap_blks_read | 77359
heap_blks_hit  | 31156
idx_blks_read  | 5
idx_blks_hit   | 5
```

Informazioni sui WAL: pg_stat_archiver

La vista pg_stat_archiver contiene esattamente una tupla con le informazioni circa lo stato di archiviazione dei WAL:

```
> SELECT * FROM pg_stat_archiver;
-[ RECORD 1 ]------+------------------------------
archived_count     | 27
last_archived_wal  | 000000010000000300000FA
last_archived_time | 2017-11-02 10:38:25.898059+01
failed_count       | 0
last_failed_wal    |
last_failed_time   |
stats_reset        | 2017-10-31 13:53:29.891094+01
```

Informazioni sul demone di scrittura: pg_stat_bgwriter

La vista pg_stat_bgwriter contiene esattamente una tupla che indica lo stato di scrittura su disco dei dati:

```
SELECT * FROM pg_stat_bgwriter;
-[ RECORD 1 ]---------+------------------------------
checkpoints_timed     | 273
checkpoints_req       | 16
checkpoint_write_time | 995400
checkpoint_sync_time  | 3340
buffers_checkpoint    | 22220
buffers_clean         | 0
maxwritten_clean      | 0
buffers_backend       | 506196
buffers_backend_fsync | 0
buffers_alloc         | 58248
stats_reset           | 2017-10-31 13:53:29.891094+01
```

dove checkpoints_timed indica quanti checkpoint schedulati sono stati eseguiti e checkpoint_req quanti ne sono richiesti (es. manualmente), e le colonne checkpoint_write_time e checkpoint_sync_time indicano in millisecondi il tempo speso per la scrittura su disco e la sincronizzazione (due fasi interne del checkpoint).

pg_stat_statements

Il modulo pg_stat_statements consente di tracciare l'attività di ogni statement sul database, permettendo quindi di "scoprire" query lente o problematiche.
Il modulo richiede la configurazione di una *shared library* (e quindi il riavvio del server) e la creazione di una estensione nel database ove si vogliono monitorare le statistiche.
Il modulo raccoglie le informazioni a livello di cluster, ma le esporta solo in un database specifico!

Installazione di pg_stat_statements

1. aggiungere il modulo a `shared_preload_libraries` in `postgresql.conf`

```
shared_preload_libraries = 'pg_stat_statements'
```

1. riavviare il servizio

```
% sudo service postgresql restart
```

1. creare l'estensione nel database ove si vorranno vedere le statistiche

```
# CREATE EXTENSION pg_stat_statements;
```

Usare pg_stat_statements

Il modulo mette a disposizione una vista speciale, pg_stat_statements, che raccoglie le statistiche di utilizzo dei database e delle query. E' bene interrogare la vista come superutente per poter vedere tutti i dati, inclusi i join per la decodifica dell'utente:

```
# SELECT auth.rolname,query, db.datname, calls, min_time, max_time,
         shared_blks_hit, shared_blks_read, shared_blks_written,
         blk_read_time, blk_write_time
 FROM pg_stat_statements
      JOIN pg_authid auth ON auth.oid = userid
      JOIN pg_database db ON db.oid = dbid
 ORDER BY calls DESC;

rolname              | luca
query                | DECLARE _psql_cursor NO SCROLL CURSOR FOR
                     | SELECT * FROM pg_stat_statements;
datname              | testdb
calls                | 3
min_time             | 0.181
max_time             | 1.119
shared_blks_hit      | 6
shared_blks_read     | 0
shared_blks_written  | 0
blk_read_time        | 0
```

Funzioni di pg_stat_statements

Il modulo mette a disposizione due funzioni:

- `pg_stat_statements_reset()` cancella tutte le statistiche collezionate fino ad ora;
- `pg_stat_statements()` una funzione che ritorna le tuple che popolano la vista (in realtà la vista è l'output di questa funzione).

Parametri di pg_stat_statements

Ci sono diversi parametri di configurazione da inserire in `postgresql.conf`, in particolare:

- `pg_stat_statements.max` il massimo numero di query da tenere nelle statistiche;
- `pg_stat_statements.save` indica se le statistiche vanno salvate fra un riavvio del cluster e l'altro (in default si).

pgstattuple

Il modulo pgstattuple fornisce informazioni su una tabella o indice in termine di dimensioni, tuple visibili, espirate, ecc. Viene installata come estensione in un database, poi permettere di accedere (come superutente) ai dati di ogni relazione qualificata:

```
# CREATE EXTENSION pgstattuple;
```

Funzioni di pgstattuple

Sono messe a disposizione due funzioni principali:

- `pgstattuple()` accetta il nome (qualificato) della relazione e ritorna le informazioni necessarie;
- `pgstattindex()` analoga, ma per gli indici.

ATTENZIONE: si richiede uno scan dell'intera relazione per ottenere le informazioni!

Usare pgstattuple

```
# SELECT * FROM pgstattuple( 'public.evento' );
-[ RECORD 1 ]------+----------
table_len          | 110092288
tuple_count        | 2110336
tuple_len          | 90411135
tuple_percent      | 82.12
dead_tuple_count   | 0
dead_tuple_len     | 0
dead_tuple_percent | 0
free_space         | 2500
free_percent       | 0
```

pg_buffercache

Il modulo pg_buffercache consente di ispezionare i dati degli shared buffers.
Viene caricato come estensione e mette a disposizione una vista che consente di verificare lo stato degli shared buffers per ogni relazione:

```
# CREATE EXTENSION pg_buffercache;
```

La vista pg_buffercache fornisce una tupla per ogni buffer nello shared buffer space.

Usare pg_buffercache

```
# SELECT c.relname, b.*
    FROM pg_buffercache b INNER JOIN pg_class c
         ON b.relfilenode = pg_relation_filenode(c.oid)
         AND b.reldatabase IN (0, (
                     SELECT oid FROM pg_database
                     WHERE datname = current_database()));
-[ RECORD 283 ]--+-------
relname          | evento
bufferid         | 477
relfilenode      | 19584
reltablespace    | 1663
reldatabase      | 19554
relforknumber    | 0
relblocknumber   | 3
isdirty          | f
usagecount       | 1
pinning_backends | 0
-[ RECORD 284 ]--+-------
relname          | evento
bufferid         | 478
relfilenode      | 19584
reltablespace    | 1663
reldatabase      | 19554
relforknumber    | 0
```

I Write Ahead Log sono un componente fondamentale per il funzionamento del database.
L'idea dietro ai `WAL` è questa:

- ogni volta che una transazione effettua un `COMMIT` i dati NON vengono scritti immediatamente su disco, per ottimizzare l'I/O.
- il `COMMIT` viene registrato nei WAL che contengono *le differenze* (binarie) dei dati;
- il WAL viene reso persistente su disco.

Siccome il WAL viene usato in modo sequenziale l'impatto di I/O è basso e si garantisce in questo modo di avere uno strato di persistenza. I WAL vengono usati principalmente in caso di crash: se il database riparte a seguito di un crash, il sistema ripercorre i segmenti di WAL e riapplica le modifiche ai dati.
Durante questa fase di *recovery* il database si trova in uno stato *inconsistente*, i WAL garantiscono appunto l'allineamento dei dati a quanto si trova su memoria persistente e quindi la *consistenza* del database. I segmenti di WAL sono fondamentali per il funzionamento del database, e quindi occorrono alcuni accorgimenti per assicurare un buon funzionamento dei WAL:

1. la dimensione dei log è fissa (16MB), così come il loro numero (con qualche piccola variazione); questo garantisce che lo spazio di archiviazione dei WAL non si riempirà mai!
2. ogni dato nel WAL viene memorizzato come *record* in linked list al record precedente; questo permette di risalire all'ultimo record "buono" nella catena dei segmenti;
3. ogni record inserito nei segmenti ha un checksum CRC32 che ne garantisce l'integrità;
4. i *signals* sono interrotti durante la scrittura dei WAL, così da garantire che il processo non sia interrotto per errore.

PostgreSQL memorizza il Write Ahead Log nella directory `pg_xlog`. Ogni *segmento* è di 16 MB e ha un nome in esadecimale.

```
% sudo  ls -lh /mnt/data1/pgdata/pg_xlog
 16M Oct 27 10:21 000000010000000000000082
 16M Oct 18 10:32 000000010000000000000083
 16M Oct 18 10:32 000000010000000000000084
```

Un checkpoint è un ponto di consolidamento fra WAL e dati fisici.

Il sistema non può registrare modifiche indefinitamente nei WAL:

- i segmenti aumenterebbe all'infinito;

- il tempo di recovery (ripercorrere i WAL) aumenterebbe di conseguenza.

Per evitare tali problemi si definiscono degli istanti periodici detti *checkpoint* ove PostgreSQL si preoccupa di consolidare i dati fisici su disco, scartando quindi i WAL precedenti quel checkpoint.
Di fatto un crash recovery ripercorre i segmenti di WAL dall'ultimo checkpoint conosciuto!

Come si fa a sapere quali pagine dati sono *sicure* per essere scaricate su memoria persistente? La risposta è LSN: Log Sequence Number.
Ogni pagina dati contiene un numero LSN che indica quale segmento di log contiene le modifiche a quella pagina. Una pagina dati non può quindi essere resa persistente se i log fino a quel `lsn` non sono stati a loro volta resi persistenti.

Vedere i LSN

I *Log Sequence Number* vengono espressi nella forma a/b in esadecimale, ad esempio:

```
4/42C2D9C8
```

Esiste una funzione apposita, `pg_xlogfile_name()`, che dato un LSN indica in quale WAL si trova il relativo record:

```
> SELECT pg_xlogfile_name( '4/42C2D9C8' );
     pg_xlogfile_name
--------------------------
 000000010000000400000042
```

Dove si trovano i LSN nei WAL?

La funzione `pg_xlogfile_name_offset()` consente non solo di avere il nome del file WAL ove si trova un record relativo ad un LSN, ma anche a che offset si trova nel file quel record:

```
> SELECT * FROM pg_xlogfile_name_offset( '4/42C2D9C8' );
         file_name          | file_offset
----------------------------+-------------
 000000010000000400000042 |    12769736
```

Distanza nei WAL

La funzione `pg_xlog_location_diff()` consente di valutare la differenza nei WAL fra due LSN (ad esempio per capire a che punto si è nella replication):

```
> SELECT pg_xlog_location_diff( '4/42C2D9C8', '2/57FFE0C8' );
 pg_xlog_location_diff
-----------------------
            8233613568
```

I checkpoint sono configurabili attraverso due parametri:

- `max_wal_size` indica quanti segmenti (da 16MB l'uno) sono tollerati fra due checkpoint consecutivi;
- `checkpoint_timeout` indica quanti secondi sono tollerati fra due checkpoint consecutivi.

Un checkpoint viene forzato aumaticamente ogni volta che si raggiunge una quantità di *dati sporchi* pari a `max_wal_size * 16 MB` o dopo che `checkpoint_timeout` secondi sono trascorsi. La prima condizione che si verifica fa scattare un checkpoint! Siccome al raggiungimento di un *checkpoint* il database deve forzare un *flush* di tutte le pagine dati "sporche", il sottosistema di I/O viene messo a dura prova.
Per evitare dei colli di bottiglia è stato introdotto un parametro che consente di stabilire entro quanto tempo il checkpoint deve essere completato: `checkpoint_completion_target`. Il suo valore è compreso fra 0 e 1 e indica la frazione di tempo (`checkpoint_timeout`) da usare per completare il checkpoint stesso. L'idea di `checkpoint_completion_target` è di rallentare l'attività di I/O di scarto delle pagine dati sporche. In sostanza le pagine dati devono essere tutte scritte entro `checkpoint_completion_target * checkpoint_timeout`

secondi che con la configurazione di default divente: `0.5 * 5 min = 2.5 min` e quindi il sistema ha circa 2.5 minuti per completare l'I/O. Il parametro offre una indicazione di come verrà *sparso* l'I/O nel tempo, non è una stima esatta. Ne consegue che valori prossimi a 1 rallenteranno le scritture su disco causando I/O costante (bassa banda), di contro valori prossimi a 0 forzano un picco di I/O rapido (alta banda). Il rischio è che valori prossimi ad 1 producano un *inseguimento* dei checkpoint continuo. Il valore corretto dipende dal carico di lavoro! Un superutente può forzare un checkpoint immediato con il comando `CHECKPOINT`. Il comando è pensato per le fasi di archiviazione e recovery, difficilmente è da usarsi nell'ultilizzo regolare del database. Stabilito che i segmenti di WAL sono *riciclati* dopo un checkpoint, è possibile stimare l'occupazione della directory `pg_xlog` che è pressoché costante nel tempo e vale:

```
( 2 + checkpoint_completion_target ) * ( max_wal_size / 16 ) + 1
```

ossia è dominata da `max_wal_segments`. Il valore sopra calcolato indica quanti segmenti di 16 MB ciascuno saranno presenti in `pg_xlog`. Durante un uso intenso potrebbe capitare che il numero di segmenti aumenti oltre `checkpoint_segments`, ma il sistema provvederà all'eliminazione del residuo una volta ripristinato l'ordine. e' quindi consigliato mantenere una

certa quota disco per consentire qualche segmento ulteriore a quanto calcolato sopra. Il parametro `wal_level` specifica quale livello di WAL si vuole usare:

- `minimal` (default), è il livello usato per garantire il normale funzionamento del cluster;
- `archive` consente di archiviare i WAL su memoria o dispositivo alternativo, utile per tecniche di backup avanzate (es. PITR) o per replica *warm standby*;
- `hot_standby` consente di rendere le repliche funzionanti da subito (in sola lettura!);
- `logical` consente di avviare la *replica logica*.

Introduzione

PostgreSQL supporta la replicazione logica dalla versione 10, ma intanto l'infrastruttura per la *decodifica* dei WAL da binario a logico (ossia verso statement SQL) è possibile.
La decodifica logica avviene attraverso dei *plugin*. Ogni plugin è responsabile di operare sui WAL e tradurli nel formato appropriato, ad esempio un formato SQL.
Per interagire con la decodifica logica occorre quindi avere un opportuno plugin per poter gestire i WAL segment. Uno di questi plugin è `test_decoding`, usato per ottenere statement SQL.
L'interazione fra plugin e WAL segment avviene attraverso i *replication slot*.

Perché i replication slot?

In PostgreSQL un *replication slot* è uno stream univoco di modifiche ordinate che un *consumer* può ripercorrere a applicare (o decodificare).
Lo slot non è al corrente di cosa sta facendo il consumer, e neanche di quale consumer vi è attaccato. Ogni consumer è responsabile di richiedere le modifiche allo slot a partire da un certo punto.
Gli slot usati per la decodifica logica sono leggermente differenti da quelli di streaming replication (detti anche *fisici*): le differenze sono nel protocollo di scambio di dati, ma concettualmente entrambi rappresentano uno stream ordinato di modifiche.

Logical Decoding: passi fondamentali

Per ottenere le informazioni logiche occorre:

1. abilitare il liello `logical` per `wal_level`;
2. usare degli *slot* di replicazione (quindi occorre abilitare almeno uno slot di replicazione);
3. interfacciarsi con i cataloghi di sistemi o appositi programmi del cluster.

ATTENZIONE: i comandi DDL non sono decodificati (ma ovviamente *esistono*)!

Configurazione del server

Nel file `postgresql.conf`:

```
wal_level = logical
max_replication_slots = 1 # almeno uno!
```

Creazione di uno slot

E' possibile usare il plugin `test_decoding`:

```
# SELECT *
  FROM pg_create_logical_replication_slot( 'logical_decoding_slot',
                                           'test_decoding');
-[ RECORD 1 ]-+----------------------
slot_name     | logical_decoding_slot
xlog_position | 3/FB0004F8
```

E' possibile creare lo slot anche tramite `pg_recvlogical`!

Interazione con lo slot e con la decodifica: SQL

```
# INSERT INTO persona( nome, cognome, codice_fiscale )
  VALUES( 'Mario', 'Rossi', 'RSSMRA71M68F357T' );
```

e interrogando la vista pg_logical_slot_get_changes si ottiene:

```
# SELECT * FROM pg_logical_slot_get_changes('logical_decoding_slot', NULL, NULL);
-[ RECORD 1 ]--------
location | 3/FB000530
xid      | 1950
data     | BEGIN 1950
-[ RECORD 2 ]--------
location | 3/FB000600
xid      | 1950
data     | table public.persona:
           INSERT: pk[integer]:16
           nome[character varying]:'Mario'
           cognome[character varying]:'Rossi'
           codice_fiscale[character varying]:'RSSMRA71M68F357T'
           eta[integer]:null
           valid[boolean]:true
-[ RECORD 3 ]--------
location | 3/FB000B18
xid      | 1950
data     | COMMIT 1950
```

Interazione con lo slot e la decodifica: pg_recvlogical

Da un terminale:

```
% pg_recvlogical -h localhost \
  -d testdb \
  -U postgres \
  --slot=logical_decoding_slot \
  --start -f -
Password:

BEGIN 1951
table public.persona: INSERT:
   pk[integer]:17
   nome[character varying]:'Giovanni'
   cognome[character varying]:'Verdi'
   codice_fiscale[character varying]:'VRDGVN71M18F357K'
   eta[integer]:null
   valid[boolean]:true
COMMIT 1951
```

che corrisponde allo stament SQL

```
# INSERT INTO persona( nome, cognome, codice_fiscale )
  VALUES( 'Giovanni', 'Verdi', 'VRDGVN71M18F357K' );
```

Slot da cancellare!

Se si vuole interrompere la replicazione occorre anche cancellare gli slot dal sistema, altrimenti al prossimo riavvio il sistema andrà in *PANIC*:

```
[3-1] PANIC:  too many replication slots active before shutdown
[3-2] HINT:  Increase max_replication_slots and try again.
```

```
# SELECT pg_drop_replication_slot( 'logical_decoding_slot' );
```

Introduzione al PITR

L'idea è abbastanza semplice ma molto potente: avendo i WAL segments + possibile ripristinare il sistema ad un istante temporale nel passato predefinito!
In sostanza si simula un *crash*, forzando il sistema a ripercorrere i WAL; invece che lasciare il sistema terminare di srotolare tutti i WAL lo si ferma ad un istante consistente nel passato.

> *Utilità (esempi): recuperare il database subito prima una transazione che ha fatto drop di tutto un database; visualizzare i dati come erano prima di una lunga elaborazione, debug di applicazioni che possono aver sporcato i dati, ecc.*

Se viene specificato un istante nel *futuro* (ovvero che non è presente nei WAL, PostgreSQL effettuerà un normale avvio in recovery. Non si può rompere PostgreSQL per un errore del DBA!

Preparasi al PITR

Le fasi da seguire sono sostanzialmente:

- configurazione del cluster affinché *esporti* i WAL segments;
- eseguire il backup dei file di dati (non importa siano *corrotti*, i WAL aggiusteranno la situazione);
- eseguire il backup dei segmenti di WAL.

Esistono due metodologie di archiviazione per PITR:

- *manuale*: si effettua un backup dei file di dati con strumenti a livello di filesystem, ma il cluster deve essere informato dell'inizio e della fine del backup per allineare opportunamento i WAL;
- *integrato*: si usa lo strumento `pg_basebackup` che consente di ottenere i file di dati senza richiedere l'utilizzo di tool a livello di filesystem.

Premessa per il PITR: WAL Archiving

Non dipende il modo che si sceglie per effettuare l'archiviazione dell'istanza, la cosa che realmente conta è che il WAL Archiving sia abilitato!
Il *Wal Archiving* è la tecnica mediante la quale PostgreSQL non ricicla automaticamente i WAL ma li mantiene *offline* per elaborazioni future.
Ovviamente questo significa che si deve avere lo spazio di archiviazione (locale, remoto) per mantenere la quantità di WAL (ossia di dati) da *ri-eseguire*.

Come impostare il WAL Archiving

Occorre specificare, nal file `postgresql.conf`, il comando shell (o lo script, o il programma) da eseguire ogni volta che un nuovo segmento di WAL viene pronto per l'archiviazione (ossia è stato scritto in `pg_xlog`).
Il comando può essere un semplice `cp` o un complesso programma, tenendo conto che i marcaposto speciali sono espansi come segue:

- `%f` è il nome relativo del file di log da archiviare;
- `%p` è il nome del file da archiviare relativo a `$PGDATA` (ossia `pg_wal/<file>`).

I file di WAL sono di proprietà dell'utente che esegue il cluster. Si tenga presente che i WAL contengono effettivamente i dati contenuti nel database (e quindi è opportuno siano protetti).

WAL Archiving: impostazioni pratiche

Si deve definire dove archiviare i WAL (può essere uno spazio locale o remoto):

```
# mkdir /mnt/data2/wal_archive \
  && chown postgres:postgres /mnt/data2/wal_archive \
  && chmod 700 /mnt/data2/wal_archive
```

Poi in `postgresql.conf` si abilita l'archiving:

```
archive_mode = on
# archive_command = 'test ! -f /mnt/server/archivedir/%f && cp %p /mnt/server/archi
archive_command = '/usr/local/bin/my_wal_archiver.sh %f %p'
wal_level = replica
```

e infine si *riavvia* il cluster (`wal_level` richiede un restart).

WAL Archiving: semplice script di archiviazione

```
% cat /usr/local/bin/my_wal_archiver.sh
#!/bin/sh

ARCHIVING_PATH=/mnt/data2/wal_archive
LOG=${ARCHIVING_PATH}/log.txt
NOW=$( date -R )

SRC_FILE=$2 # file with path
DST_FILE=${ARCHIVING_PATH}/$1

# non sovrascrivo il file se esiste gia!
if [ -f ${DST_FILE} ]
then
    echo "KO $NOW: $1 esiste gia'" >> $LOG
    exit 1
fi

cp ${SRC_FILE} ${DST_FILE} \
  && echo "OK $NOW: WAL segment $1 copiato" >> $LOG \
  && exit 0
```

Cosa succede a regime?

Man mano che i dati vengono immessi nel database (o un checkpoint viene raggiunto) lo spazio di archiviazione dei WAL si inizia a popolare:

```
% sudo -u postgres ls -lh /mnt/data2/wal_archive
...
-rw-------  1 postgres  postgres    16M Jan 17 13:30 000000010000000000000017
-rw-------  1 postgres  postgres    16M Jan 17 13:30 000000010000000000000018
-rw-------  1 postgres  postgres    16M Jan 17 13:30 000000010000000000000019
-rw-------  1 postgres  postgres   405B Jan 17 13:30 log.txt
```

Nota: il file `log.txt` è stato generato dallo script di archiviazione, non da PostgreSQL!

Forzare l'archiviazione

Il database archivia i WAL segment ogni volta che ha terminato di prepararne uno (es. al checkpoint). Questo può rappresentare un problema poiché se ci sono pochi dati o il tempo fra due checkpoint è alto si possono perdere dei segmenti di WAL e, conseguentemente, non riuscire a ripristinare tutti i dati (ossia la granularità dell'ultimo periodo si riduce). L'opzione `archive_timeout` permette di specificare un numero di secondi dopo il quale forzare un nuovo segmento di WAL (e la conseguente archiviazione). Ad esempio:

```
archive_timeout = 20
```

e i log sono archiviati ogni 20 secondi (circa):

```
% sudo -u postgres cat /mnt/data2/wal_archive/log.txt
OK Wed, 17 Jan 2018 13:39:27 +0100: WAL segment 00000001000000000000001C copiato
OK Wed, 17 Jan 2018 13:42:08 +0100: WAL segment 00000001000000000000001D copiato
OK Wed, 17 Jan 2018 13:42:28 +0100: WAL segment 00000001000000000000001E copiato
```

Una volta che l'archiviazione dei WAL è attiva occorre:

1. informare il cluster che il backup sta iniziando con `pg_start_backup()`;
2. effettuare il backup dei file di dati (attenzione a non copiare la configurazione!);
3. informare il cluster che il backup è terminato con `pg_stop_backup()`.

Non importa il tempo che passa fra il punto 1 e 2, il comportamento del cluster non viene alterato!

Fase 1: informare il cluster che il backup sta iniziando

Come superutente del cluster avviare il backup:

```
# SELECT pg_start_backup( 'PITR_BACKUP_MANUAL', true, false );
 pg_start_backup
-----------------
 0/37000108
```

I parametri di `pg_start_backup` sono:

- una etichetta testuale che identifica il backup (*label*);
- l'avvio immediato (`true`) di un checkpoint senza onorare il `checkpoint_completion_target` (e quindi usando tutto l'I/O disponibile), se si vuole far usare meglio il database lasciarlo a `false`;
- l'eventuale indicazione di un backup esclusivo (ossia impedisce la concorrenza di backup).

Fase 2: effettuare il backup fisico di PGDATA

E' ora possibile usare lo strumento a livello di filesystem per effettuare la copia:

```
% sudo -u postgres rsync -a /mnt/data1/pgdata/ /mnt/data3/pgdata/
% sudo -u postgres rm /mnt/data3/pgdata/postmaster.pid \
                      /mnt/data3/pgdata/postmaster.opts
```

Fase 3: informare il cluster che il backup è terminato

Nella stessa connessione come utente amministratore eseguire pg_stop_backup:

```
# SELECT pg_stop_backup( false );
NOTICE:  pg_stop_backup complete, all required WAL segments have been archived
-[ RECORD 1 ]--+-------------------------------------------------------------
pg_stop_backup | (0/38000088,"START WAL LOCATION: 0/37000108 (file 000000010000000
               | CHECKPOINT LOCATION: 0/37000140
               | BACKUP METHOD: streamed
               | BACKUP FROM: master
               | START TIME: 2018-01-17 13:46:42 CET
               | LABEL: PITR_BACKUP_MANUAL
               | ","")
```

Questo produrrà un nuovo WAL segment switch per garantire che anche l'ultimo segmento è stato reso persistente e archiviato.

Fase 3: conseguenze

Nello spazio di archiviazione viene creato un file con suffisso .backup che contiene le informazioni riportate da pg_stop_backup e che sono vitali per il backup stesso:

```
% sudo cat /mnt/data2/wal_archive/000000010000000000000037.00000108.backup
START WAL LOCATION: 0/37000108 (file 000000010000000000000037)
STOP WAL LOCATION: 0/38000088 (file 000000010000000000000038)
CHECKPOINT LOCATION: 0/37000140
BACKUP METHOD: streamed
BACKUP FROM: master
START TIME: 2018-01-17 13:46:42 CET
LABEL: PITR_BACKUP_MANUAL
STOP TIME: 2018-01-17 13:48:27 CET
```

Si noti che il nome file è concatenato dal nome del segmento WAL (0x37), dal punto di inizio del backup come riportato da pg_start_backup (0x370000108).

Problema di esempio: nuke di una tabella

Si cancellano dei dati da una tabella:

```
> BEGIN;
> INSERT INTO evento( description )
  SELECT 'Evento ' || generate_series( 1, 2000000 );
> SELECT txid_current(), current_timestamp;
 txid_current |       current_timestamp
--------------+-------------------------------
          973 | 2018-01-17 14:07:05.806106+01
> COMMIT;

-- si bevono alcune birre e...

> BEGIN;
> DROP TABLE evento;
> SELECT txid_current(), current_timestamp;
 txid_current |       current_timestamp
--------------+-------------------------------
          975 | 2018-01-17 14:09:30.041652+01
> COMMIT;
```

Far ripartire un nuovo cluster

Occorre creare un file `recovery.conf`:

- `restore_command` è l'opposto di `archive_command` e informa il cluster su come reperire i WAL segment dall'archivio;
- `recovery_end_command` viene eseguito una volta che tutti i WAL sono stati riprodotti (ripristino terminato);
- `recovery_target_time` o `recovery_target_xid` indicano l'istante temporale o il numero di transazione a cui fermare il recovery.

Nello specifico:

```
restore_command = 'cp /mnt/data2/wal_archive/%f "%p"'
recovery_target_time = '2018-01-17 14:09:00'
# opzionale
#recovery_target_action = promote
```

Avviare il sistema in PITR

```
% sudo -u postgres pg_ctl -D /mnt/data3/pgdata -o "-p 5433" start
waiting for server to start....
LOG:  listening on IPv6 address "::1", port 5433
LOG:  listening on IPv4 address "127.0.0.1", port 5433
LOG:  listening on Unix socket "/tmp/.s.PGSQL.5433"
LOG:  database system was shut down in recovery at 2018-01-17 14:10:41 CET
LOG:  starting point-in-time recovery to 2018-01-17 14:09:00+01
LOG:  restored log file "000000010000000000000039" from archive
LOG:  redo starts at 0/39000060
LOG:  restored log file "00000001000000000000003A" from archive
LOG:  restored log file "00000001000000000000003B" from archive
LOG:  restored log file "00000001000000000000003C" from archive
...
LOG:  consistent recovery state reached at 0/4E000000
LOG:  database system is ready to accept read only connections
...
LOG:  recovery stopping before commit of transaction 975, time 2018-01-17 14:09:38.
LOG:  recovery has paused
HINT:  Execute pg_wal_replay_resume() to continue.
server started
```

Abilitare il sistema in PITR

Il sistema automaticamente si mette in pausa prima di accettare connessioni read/write: questo permette di analizzare lo stato del database per capire se tutto è corretto. Occorre abilitare in manuale il sistema indicando con `pg_wal_replay_resume()` che il recovery è terminato:

```
% psql -h localhost -U postgres -p 5433 testdb
# SELECT pg_wal_replay_resume();
```

e quindi il database risponde con:

```
2018-01-17 14:18:39.500 CET [1527] LOG:  redo done at 0/64000F58
2018-01-17 14:18:39.500 CET [1527] LOG:  last completed transaction was at log time
2018-01-17 14:18:39.536 CET [1527] LOG:  selected new timeline ID: 2
2018-01-17 14:18:39.640 CET [1527] LOG:  archive recovery complete
2018-01-17 14:18:41.014 CET [1526] LOG:  database system is ready to accept connect
```

Se non si vuole agire in manuale occorre impostare nel `recovery.conf` una delle seguenti opzioni:

- non specificare un `recovery_target` (di qualunque tipo), non possibile in PITR;
- impostare `recovery_target_action` a `promote`.

PITR: risultato

```
% psql -h localhost -U luca -p 5433 testdb
> SELECT count(*) FROM evento;
-[ RECORD 1 ]--
count | 2000000
```

Il file `recovery.conf` viene rinominato in `recovery.done` (per evitare altri avvi in *recovery mode*). Il file `backup_label` viene rimosso.

Usare pg_basebackup

E' un programma che realizza il backup in modo quasi automatico, ma richiede una riga di configurazione nel file `pg_hba.conf` per consentire una connessione al database sorgente:

```
host    replication     postgres  127.0.0.1/32   trust
```

E' inoltre indispensabile assicurarsi di avere almeno due *WAL Sender* attivi in `postgresql.conf`:

```
max_wal_senders = 2
```

pg_basebackup in azione

```
% sudo -u postgres pg_basebackup \
   -D /mnt/data3/pgdata \
   -l 'PITR BACKUP 2' -v \
   -h localhost -p 5432 -U postgres

pg_basebackup: initiating base backup, waiting for checkpoint to complete
pg_basebackup: checkpoint completed
pg_basebackup: write-ahead log start point: 0/6A000028 on timeline 1
pg_basebackup: starting background WAL receiver
pg_basebackup: write-ahead log end point: 0/6A0000F8
pg_basebackup: waiting for background process to finish streaming ...
pg_basebackup: base backup completed
```

ATTENZIONE: verificare `postgresql.conf`!

pg_basebackup: backup_label

Il file backup_label contiene le informazioni necessarie per raccapezzarsi con il backup:

```
% sudo cat /mnt/data3/pgdata/backup_label
START WAL LOCATION: 0/6A000028 (file 000000010000000000000006A)
CHECKPOINT LOCATION: 0/6A000060
BACKUP METHOD: streamed
BACKUP FROM: master
START TIME: 2018-01-17 14:29:26 CET
LABEL: PITR BACKUP 2
```

Costruire il recovery.conf

Si consdiera lo stesso esempio visto in precedenza con i seguenti dati:

- transazione di inserimento tuple: `xid = 978`, `current_timestamp = 2018-01-17 14:32:50.604495+01`
- transazione di nuke: `xid = 981`, `current_timestamp = 2018-01-17 14:35:30.640968+01`.

Il file `recovery.conf` si costruisce in maniera *identica* al caso visto in precedenza, o per variare ci si basa sulla transazione invece che sul tempo:

```
restore_command = 'cp /mnt/data2/wal_archive/%f "%p"'
recovery_target_xid = 981           # transazione di nuke
recovery_target_inclusive = false   # non vogliamo il nuke!
recovery_target_action = promote    # siamo sicuri di voler riprendere!
```

Lo stato di archiviazione

Man mano che una istanza ripercorre i WAL inserisce nella directory `pg_wal/archive_status` dei file `.done` che rappresentano il segmento di WAL ripristinato, nonché da che backup si è partiti:

```
# ls /mnt/data3/pgdata/pg_wal/archive_status/
000000010000000000000006A.done   000000010000000000000070.done
000000010000000000000006B.done   000000010000000000000071.done
000000010000000000000006C.done   000000010000000000000072.done
000000010000000000000006D.done   000000010000000000000073.done
000000010000000000000006E.done   000000010000000000000074.done
000000010000000000000006F.done   000000010000000000000075.done
```

Timelines

Quando una istanza finisce di ripercorrere dei segmenti di WAL e diventa nuovamente usabile, essa crea una nuova *timeline*:

```
2018-01-17 14:18:39.536 CET [1527] LOG:  selected new timeline ID: 2
```

L'idea è che da lì in avanti il database deve essere capace di distinguere fra i segmenti di WAL generati dal cluster (riavviato) e quelli provenienti dal cluster originale. Per fare questo la *timeline* fa parte del nome del file di ogni WAL:

```
% sudo ls /mnt/data3/pgdata/pg_xlog/
...
000000010000000000000086
00000002.history
000000020000000000000087
000000020000000000000088
```

I file con prefisso 00000002 appartengono alla seconda timeline, che è un *branch* della timeline 00000001 del primo cluster.

Timeline history file

Le timeline sono mantenute nella directory `pg_wal` e sono in formato testo, con suffisso `.history`. Tali file contengono le informazioni sulla timeline di partenza e il momento (ossia il segmento di WAL) che ha generato il branch. Ad esempio per un recovery:

```
LOG:  redo done at 0/87000028
LOG:  last completed transaction was at log time 2018-01-17 14:35:14.034293+01
LOG:  restored log file "000000010000000000000086" from archive
LOG:  selected new timeline ID: 2
```

si ha una timeline history

```
% sudo cat /mnt/data3/pgdata/pg_wal/00000002.history
1       0/87000000      before transaction 981
```

Il `recovery.conf` lavora implicitamente sulla stessa timeline del server di origine. E' tuttavia possibile specificare una timeline differente.

Concetti e terminologia

La *replication* è la capacità di *sincronizzare* un cluster master con uno o piu' slave che replicano i dati (ossia ripetono le transazioni). Il nodo master è l'unico che accetta le query DML (ossia l'unico abilitato alle scritture). Gli slave, detti anche *stand-by*, possono funzionare in replica isolata (warm standby) o in replica funzionante (hot standby): in questo caso essi accettano solo query di lettura (e quindi possono essere usati per il bilanciamento del carico).

Tipologia di replica

Esistono fondamentalmente tre tipi di *replication*:

1. log shipping: il master archivia i WAL segment che lo slave ripercorre indefinitamente;
2. streaming: il master invia, su richiesta, i WAL segment allo slave che li richiede. Questo tipo di replica si suddivide ulteriormente in:
   - sincrona il master attende che anche lo slave abbia eseguito la transazione (molto sicuro, ma molto "costoso");
   - asincrona il master si "fida" che lo slave porterà a termine la transazione.
3. logical: supportata solo dalla versione 10, consente di inviare un flusso di *statement* per la modifica e quindi abilita lo scenario bidirezionale.

Ogni slave può a sua volta diventare un punto di "avvio" di replicazione per un successivo slave.

Fase 1: configurazione del master

Occore che il master consenta l'archiviazione dei WAL segment e supporti la connessione per replica (solo se si usa pg_basebackup). In postgresql.conf:

```
wal_level       = replica
archive_mode    = on
archive_command = '/usr/local/bin/my_wal_archiver.sh %f %p'
max_wal_senders = 2 # solo per usare pg_basebackup
```

e se si usa pg_basebackup in pg_hba.conf:

```
host  replication  postgres    127.0.0.1/32  trust
```

Fase 2: creazione dello slave

```
% sudo -u postgres pg_basebackup
   -D /mnt/data3/pgdata
   -l 'LOG_SHIPPING_REPLICA' -v
   -h localhost -p 5432 -U postgres
```

Editare il file `postgresql.conf` dello slave affinché:

```
wal_level       = minimal
archive_mode    = off
max_wal_senders = 0
hot_standby     = on        # solo per fare query
port            = 5433      # solo per test locale!!
# logging_collector = off # per vedere i log!
```

Fase 3: creare il `recovery.conf`

Anche nel caso di replication occorre generare un file `recovery.conf` che istruisca lo slave su come "riavviarsi":

```
standby_mode = 'on'
restore_command = 'cp /mnt/data2/wal_archive/%f "%p"'
```

In sostanza non si specifica a quale istante temporale fermare il recovery e si abilita la funzione `standby_mode` che, appunto, istruisce lo slave a *seguire* le transazioni del master.

Fase 4: avvio dello slave

```
% sudo -u postgres pg_ctl -D /mnt/data3/pgdata start
waiting for server to start....2018-01-18 12:00:49.966 CET [1183] LOG:  listening o
LOG:  listening on IPv4 address "127.0.0.1", port 5433
LOG:  listening on Unix socket "/tmp/.s.PGSQL.5433"
LOG:  database system was shut down in recovery at 2018-01-18 11:59:58 CET
LOG:  entering standby mode
LOG:  restored log file "000000010000000000000008C" from archive
LOG:  redo starts at 0/8C000060
LOG:  consistent recovery state reached at 0/8D000000
LOG:  database system is ready to accept read only connections
...
```

Come si nota il database accetta solo query in lettura ed è pronto.

Interazione con lo slave

Come già detto, lo slave è in sola lettura:

```
% psql -h localhost -U luca -p 5433 testdb
> INSERT INTO persona( nome, cognome, codice_fiscale )
  VALUES( 'Luca', 'Stantunione', 'SNTLCU02A20F257T' );
ERROR:  cannot execute INSERT in a read-only transaction
```

Perdita di dati sullo slave ?

Sul master:

```
% psql -h localhost -U luca -p 5432 testdb
> CREATE TABLE evento( pk SERIAL PRIMARY KEY, description TEXT);
> INSERT INTO evento( description )
  SELECT 'replication ' || generate_series(1, 2000 );
```

ma sullo slave non è ancora visibile:

```
% psql -h localhost -U luca -p 5433 testdb
> SELECT * FROM evento;
ERROR:  relation "evento" does not exist
```

Perdita di dati sullo slave, perché?

Non è una perdita di dati: lo slave non ha ancora ricevuto il file WAL da ripercorrere.

```
cp: /mnt/data2/wal_archive/00000001000000000000008D: No such file or directory
cp: /mnt/data2/wal_archive/00000001000000000000008D: No such file or directory
LOG:  restored log file "00000001000000000000008D" from archive
LOG:  restored log file "00000001000000000000008E" from archive
LOG:  restored log file "00000001000000000000008F" from archive
LOG:  restored log file "000000010000000000000090" from archive
```

ma quando finalmente il master produce i nuovi WAL, in particolare l'atteso 0x10x08D allora ecco che lo slave si "allinea":

```
% psql -h localhost -U luca -p 5433 testdb
> SELECT count(*) FROM evento;
 count
-------
  2000
```

Perdita di dati sullo slave, che fare?

Non è una perdita di dati: lo slave rimane indietro dei segmenti di WAL non ancora archiviati. Occorre allora agire sul master impostando il parametro:

```
archive_timeout = 10
```

che forza una archiviazione ogni 10 secondi (ad esempio). In alternativa, manualmente, come utente amministratore è possibile invocare la funzione `pg_switch_wal()`:

```
# SELECT pg_switch_wal();
 pg_switch_wal
---------------
 0/FD000158
```

Perdita di dati: che fare? La strada sbagliata!

Non ci si deve fidare delle date di modifica dei WAL sul master!
PostgreSQL sa esattamente quando un WAL è completo, e quando non lo è. Se un WAL è stato scritto parzialmente le sue date sul file system saranno aggiornate, ma questo non lo rende idoneo all'archiviazione.
Se si forza l'archiviazione si combina un macello: il sistema di archiving deve impedire la copia di uno stesso segmento WAL due volte e quindi forzarlo manualmente è sbagliato!

Perdita di dati: che fare? La strada sbagliata (esempio)

Se il master ha generato come ultimo WAL il 0x010x95, e lo salve continua a dire che non ha ricevuto tale wal, di fatto il sistema di archiviazione non ha inviato all'archivio tale WAL.

Lo slave si lamenta:

```
LOG:  restored log file "000000010000000000000094" from archive
cp: /mnt/data2/wal_archive/000000010000000000000095: No such file or directory
```

il sistema di archiviazione dice di non aver copiato l'ultimo WAL che lo slave si aspetta:

```
OK Thu, 18 Jan 2018 12:11:56 +0100: WAL segment 000000010000000000000094 copiato
```

E' possibile forzare una archiviazione manuale del nuovo WAL:

```
% sudo -u postgres /usr/local/bin/my_wal_archiver.sh \
        000000010000000000000095 \
        /mnt/data1/pgdata/pg_wal/000000010000000000000095
```

e lo slave riceve subito il WAL:

```
LOG:  restored log file "000000010000000000000094" from archive
LOG:  restored log file "000000010000000000000095" from archive
LOG:  invalid record length at 0/957BD6D0: wanted 24, got 0
LOG:  restored log file "000000010000000000000095" from archive
LOG:  invalid record length at 0/957BD6D0: wanted 24, got 0
```

Ma da qui in poi lo slave rimarrà indietro perché lo script di archiviazione

Perdita di spazio, che fare?

L'archiviazione dei WAL non può avvenire all'infinito. Il `recovery.conf` consente di specificare un comando da effettuare una volta che il segmento è stato ripercoso (a patto che non serva per altri scopi!). Il comando `pg_archivecleanup` effettua la cancellazione dei file recuperati. Nel file `recovery.conf` il placeholder `%r` indica un segmento recuperato, quindi:

```
standby_mode = 'on'
restore_command = 'cp /mnt/data2/wal_archive/%f "%p"'
archive_cleanup_command = 'pg_archivecleanup /mnt/data2/wal_archive %r'
```

Fase 1: configurazione del master

Occorre che il master consenta un numero di processi di invio dei WAL (*WAL Senders*) pari o superiore al numero di slave che si connetteranno. Inoltre il master deve sapere che i WAL saranno sottoposti a replication, quindi in `postgresql.conf`:

```
wal_level       = replica
max_wal_senders = 1 # quanti slave?
```

Siccome lo slave si collegherà al master, deve essere possibile l'autenticazione, quindi in `pg_hba.conf`:

```
host  replication  postgres    127.0.0.1/32  md5
```

Fase 2: creazione dello slave

```
% sudo -u postgres pg_basebackup
   -D /mnt/data3/pgdata
   -l 'LOG_SHIPPING_REPLICA' -v
   -h localhost -p 5432 -U postgres
```

Editare il file `postgresql.conf` dello slave affinché:

```
wal_level        = minimal
archive_mode     = off
max_wal_senders  = 0
hot_standby      = on # solo per fare query
port             = 5433 # solo per test locale!!
```

Fase 3: creare il `recovery.conf`

Anche nel caso di replication occorre generare un file `recovery.conf` che istruisca lo slave su come "riavviarsi":

```
standby_mode = 'on'
primary_conninfo = 'host=localhost port=5432 user=postgres password=xxxxx'
```

In sostanza si specifica a quale server remoto collegarsi per ottenere il flusso di WAL.

Fase 4: avvio dello slave

```
% sudo -u postgres pg_ctl -D /mnt/data3/pgdata start
waiting for server to start....
LOG:  listening on IPv6 address "::1", port 5433
LOG:  listening on IPv4 address "127.0.0.1", port 5433
LOG:  listening on Unix socket "/tmp/.s.PGSQL.5433"
LOG:  database system was interrupted; last known up at 2018-01-25 15:04:18 CET
LOG:  entering standby mode
LOG:  redo starts at 0/9A000060
LOG:  consistent recovery state reached at 0/9A000130
LOG:  database system is ready to accept read only connections
LOG:  started streaming WAL from primary at 0/9B000000 on timeline 1
 done
server started
```

Come si nota il database accetta solo query in lettura ed è pronto.

Monitoring

La vista pg_stat_replication fornisce informazioni utili sullo stato di ogni slave (occorre essere sul nodo master e interrogarla come superutente):

```
# SELECT * FROM pg_stat_replication;
-[ RECORD 1 ]----+------------------------------
pid              | 1135
usesysid         | 10
usename          | postgres
application_name | walreceiver
client_addr      | 127.0.0.1
client_hostname  |
client_port      | 25829
backend_start    | 2018-01-25 15:09:52.833939+01
backend_xmin     |
state            | streaming
sent_lsn         | 0/9B000140
write_lsn        | 0/9B000140
flush_lsn        | 0/9B000140
replay_lsn       | 0/9B000140
write_lag        |
flush_lag        |
replay_lag       |
sync_priority    | 0
sync_state       | async
```

Monitoring: successivamente...

Man mano che passa il tempo e vengono prodotti dati sul nodo master la situazione di sincronizzazione varia:

```
# SELECT * FROM pg_stat_replication;
-[ RECORD 1 ]----+------------------------------
pid              | 1135
usesysid         | 10
usename          | postgres
application_name | walreceiver
client_addr      | 127.0.0.1
client_hostname  |
client_port      | 25829
backend_start    | 2018-01-25 15:09:52.833939+01
backend_xmin     |
state            | streaming
sent_lsn         | 0/A4479AD0
write_lsn        | 0/A4479AD0
flush_lsn        | 0/A4479AD0
replay_lsn       | 0/A4479AD0
write_lag        | 00:00:00.000211
flush_lag        | 00:00:00.000693
replay_lag       | 00:00:00.000961
sync_priority    | 0
sync_state       | async
```

Replication Slot

Cosa succede se lo slave si disconnette e il master inizia a riciclare i WAL?
Lo slave sarà impossibilitato a recuperare il suo stato!
Possibili soluzioni:

- archiviazione dei WAL (ma potrebbe richiedere molto spazio);
- replication slot.

I replication slot informano il master di tenere i WAL segment qualora non siano stati inviati agli slave relativi.
Un replication slot è uno stream ordinato e univoco di modifiche!

Replication Slot: configurazione dello slave

Si deve impostare il nome dello slot nel `recovery.conf`:

```
standby_mode = 'on'
primary_conninfo = 'host=localhost port=5432 user=postgres password=xxxxx'
primary_slot_name = 'slave_a_slot'
```

E' necessario fermare e riavviare lo slave, qualora non sia stato configurato da subito per usare gli slot di replicazione!

Replication Slot: configurazione del master

Nel file `postgresql.conf` si deve specificare quanti slot si vogliono mantenere:

```
max_replication_slots = 1
```

e si possono quindi creare gli slot con il nome identico a quello fornito agli slave:

```
# SELECT *
  FROM pg_create_physical_replication_slot( 'slave_a_slot' );
-[ RECORD 1 ]-+-------------
slot_name     | slave_a_slot
xlog_position |
```

Replication Slot: monitoring

Una volta avviato lo slave è possibile verificare (dal master) in che stato sono i replication slot:

```
# SELECT * FROM pg_replication_slots;
-[ RECORD 1 ]-------+-------------
slot_name           | slave_a_slot
plugin              |
slot_type           | physical
datoid              |
database            |
temporary           | f
active              | t
active_pid          | 1323
xmin                |
catalog_xmin        |
restart_lsn         | 0/A4480848
confirmed_flush_lsn |
```

Replication Slot: esperimento

Se lo slave viene fermato (svanisce) la vista pg_stat_replication non riporterà alcun risultato, mentre lo slot sarà ancora visibile. Se lo slave riprende la sua attività:

```
% sudo -u postgres pg_ctl -D /mnt/data3/pgdata start
waiting for server to start....
...
LOG:  invalid record length at 0/A4480848: wanted 24, got 0
 done
server started
LOG:  started streaming WAL from primary at 0/A4000000 on timeline 1
```

e le viste mostrano la situazione aggiornata:

```
# SELECT * FROM pg_replication_slots;
-[ RECORD 1 ]-------+-------------
slot_name           | slave_a_slot
plugin              |
slot_type           | physical
datoid              |
database            |
temporary           | f
active              | t
active_pid          | 1411
xmin                |
```

Application Name

La replica sincrona passa per il concetto di *application name*. L'idea è semplice: ogni connessione viene associata ad un *nome applicazione*. Qualora la connessione (di replica dallo slave) abbia un nome di connessione riconosciuto fra quelli che devono comportarsi come replica sincrona, il sistema viene istruito di comportarsi come replica sincrona.
La replica sincrona è costosa: si deve aspettare che ogni slave abbia effettuato la transazione!
Nel caso di replica asincrona l'application name di default è `walreceiver`*!*

Fase 1: configurazione del master

Il master deve consentire un opportuno numero di slave (e quindi di *WAL Senders*), nonché sapere quali slave si collegheranno in modo sincrono (ossia la *application name* deve essere nota fra tutti i sistemi!). Nel file `postgresql.conf`:

```
synchronous_standby_names = 'slave_a, slave_b'
wal_level        = replica
max_wal_senders = 2 # quanti slave?
```

Come per la replica asincrona, occorre che sia concessa la possibilità di connessione agli slave, quindi in `pg_hba.conf`:

```
host  replication  postgres    127.0.0.1/32  md5
```

e righe relative ai vari slave.

Fase 2: creazione dello slave

```
% sudo -u postgres pg_basebackup \
   -D /mnt/data3/pgdata \
   -l 'LOG_SHIPPING_REPLICA' -v \
   -h localhost -p 5432 -U postgres
```

Editare il file `postgresql.conf` dello slave affinché:

```
wal_level        = minimal
archive_mode     = off
max_wal_senders  = 0
hot_standby      = on # solo per fare query
port             = 5433 # solo per test locale!!
```

Fase 3: creare il `recovery.conf`

L'application name deve essere inserito nella stringa di connessione affinché lo slave si identifichi per la replica sincrona:

```
standby_mode = 'on'
primary_conninfo = 'host=localhost port=5432 user=postgres password=xxxx applicatio
```

Fase 4: avviare lo slave

```
% sudo -u postgres pg_ctl -D /mnt/data3/pgdata start
server starting
LOG:  database system was shut down in recovery at 2017-10-31 13:17:10 CET
LOG:  entering standby mode
LOG:  redo starts at 3/92000060
LOG:  consistent recovery state reached at 3/9A000000
LOG:  database system is ready to accept read only connections
LOG:  unexpected pageaddr 3/6B000000 in log segment 000000010000000300000009A, offse
LOG:  started streaming WAL from primary at 3/9A000000 on timeline 1
```

Monitoring in replica

Su uno slave è possibile vedere l'ultimo segment WAL ricevuto con `pg_last_xlog_received()`:

```
> SELECT pg_last_xlog_receive_location();
-[ RECORD 1 ]-----------------+-----------
pg_last_xlog_receive_location | 3/A3000000
```

mentre sul master `pg_current_xlog_location()` indica dove si è:

```
> SELECT PG_CURRENT_XLOG_LOCATION();
-[ RECORD 1 ]------------+-----------
pg_current_xlog_location | 3/A8000060
```

Monitoring in replica (2)

Sul nodo master, come superutenti, è possibile avere un'idea dei singoli slave collegati:

```
# SELECT * FROM pg_stat_replication;
-[ RECORD 1 ]----+------------------------------
pid              | 1648
usesysid         | 10
usename          | postgres
application_name | slave_a
client_addr      | 127.0.0.1
client_hostname  |
client_port      | 63390
backend_start    | 2017-10-31 13:23:49.874223+01
backend_xmin     |
state            | streaming
sent_location    | 3/AC000000
write_location   | 3/AC000000
flush_location   | 3/AC000000
replay_location  | 3/AC000000
sync_priority    | 1
sync_state       | sync
```

Promote

Uno slave può diventare un master qualora sia necessario.
Diventare master (promote) significa che lo slave inizierà a percorrere la sua stessa timeline e si sgancerà dal vecchio master (ossia sarà completato il recovery).
Ci sono due modi per promuovere uno slave:

- fare una *promote* esplicita;
- usare un trigger file.

Promote esplicita

Usando pg_ctl si può effettuare il promote di un database:

```
% sudo -u postgres pg_ctl -D /mnt/data3/pgdata promote
server promoting
LOG:  received promote request
FATAL:  terminating walreceiver process due to administrator command
LOG:  invalid record length at 3/F7000140: wanted 24, got 0
LOG:  redo done at 3/F7000108
LOG:  selected new timeline ID: 2
LOG:  archive recovery complete
LOG:  MultiXact member wraparound protections are now enabled
LOG:  database system is ready to accept connections
LOG:  autovacuum launcher started
```

Promote via trigger file

Si deve specificare un'opzione nel file `recovery.conf`:

```
standby_mode = 'on'
primary_conninfo = 'host=localhost port=5432 user=postgres password=xxx'
trigger_file = '/tmp/promote_to_master'
```

Quando sarà presente il file `/tmp/promote_to_master` (non importa il contenuto!) lo slave si sgancerà dal master.

Promote via trigger file: esempio

```
% echo ‘date‘ > /tmp/promote_to_master
```

e lo slave reagisce con

```
LOG:  trigger file found: /tmp/promote_to_master
FATAL:  terminating walreceiver process due to administrator command
LOG:  invalid record length at 3/F9000060: wanted 24, got 0
LOG:  redo done at 3/F9000028
LOG:  selected new timeline ID: 2
LOG:  archive recovery complete
LOG:  MultiXact member wraparound protections are now enabled
LOG:  database system is ready to accept connections
LOG:  autovacuum launcher started
```

Il problema

Nella streaming replication (o analoga soluzione), il database clone ha una serie di wal che non sono piu' necessari. Il file `.backup` contiene infatti tali informazioni:

```
% sudo cat /mnt/data2/wal_archive/000000010000000000000037.00000108.backup
START WAL LOCATION: 0/37000108 (file 000000010000000000000037)
...
```

ovvero tutti i file piu' vecchi di 0x0037 possono essere eliminati per creare spazio.

pg_archivecleanup

Il comando pg_archivecleanup può essere usato come comando stand-alone o come punto finale di un recovery per eliminare i wal non necessari.

```
% sudo -u postgres pg_archivecleanup /mnt/data3/pgdata
          /mnt/data2/wal_archive/000000010000000000000037.00000108.backup
```

In alternativa, nel recovery.conf può usare un comando di fine recovery per fare pulizia:

```
# recovery.conf
...
restore_command = 'cp /mnt/data2/wal_archive/%f "%p"'
archive_cleanup_command = 'pg_archivecleanup /mnt/data2/wal_archive/ %r'
```

ove %r indica l'ultimo WAL da tenere, ovvero il restart point.

Introduzione alla replicazione logica

Consente di inviare una parte ridotta di informazioni di replicazione, tipicamente basandosi su un identificativo dei dati (es. la chiave primaria).
Si differenzia da quella fisica perché non prevede un flusso di informazioni binarie (quale pagina dati, a quale offset, quali valori) ma prevede un flusso di dati *logici* per eseguire l'aggiornamento di una specifica tupla.
Prevede un meccanismo di publish & subscribe dei componenti: il master *pubblica* i dati di replicazione, i *subscribers* li consumano.
Il *subscriber* può essere a sua volta un *publisher* (quindi si supporta la *cascading replication*). Inoltre il *subscriber* può essere usato in scrittura, ma in questo scenario possono nascere dei conflitti.

Limitazioni della replica logica

- i comandi *DDL* non sono replicati;
- `TRUNCATE` non viene replicato;
- replicare tabelle prive di chiave primaria è una pessima idea;

Configurazione del master

```
wal_level = logical
max_replication_slots = 5 # numero di subscribers, meglio se superiore!
max_wal_senders = 2
```

Clonazione dello slave

```
% sudo -u postgres pg_basebackup
       -D /mnt/data3/pgdata
       -l 'LOGICAL_REPLICA' -v
       -h localhost -p 5432 -U postgres

pg_basebackup: initiating base backup, waiting for checkpoint to complete
pg_basebackup: checkpoint completed
pg_basebackup: write-ahead log start point: 0/B2000028 on timeline 1
pg_basebackup: starting background WAL receiver
pg_basebackup: write-ahead log end point: 0/B2000130
pg_basebackup: waiting for background process to finish streaming ...
pg_basebackup: base backup completed
```

Configurazione dello slave

```
wal_level = minimal
port = 5433
max_wal_senders = 0
max_replication_slot = 2
```

Pubblicazione dal master

Sul nodo master:

```
> CREATE TABLE replicated( pk serial, description text );

> CREATE PUBLICATION replication_test FOR TABLE replicated;
```

Ricezione sullo slave

Sul nodo slave:

```
# CREATE SUBSCRIPTION replication_sub_test
  CONNECTION 'host=localhost port=5432 dbname=testdb'
  PUBLICATION replication_test;
```

e nei log dello slave appare qualcosa del tipo:

```
LOG:  logical replication apply worker for subscription "replication_sub_test" has
LOG:  logical replication table synchronization worker
      for subscription "replication_sub_test",
      table "replicated" has started
LOG:  logical replication table synchronization worker
      for subscription "replication_sub_test",
      table "replicated" has finished
```

Monitorare l'andamento

Sul nodo master:

```
# SELECT * FROM pg_replication_slots;
slot_name           | replication_sub_test
plugin              | pgoutput
slot_type           | logical
datoid              | 16386
database            | testdb
temporary           | f
active              | t
active_pid          | 1739
xmin                |
catalog_xmin        | 1275
restart_lsn         | 0/FCA3D610
confirmed_flush_lsn | 0/FCA3D648
```

mentre sul nodo slave:

```
# SELECT * FROM pg_replication_origin_status;
local_id    | 1
external_id | pg_25954
remote_lsn  | 0/FCA3D468
local_lsn   | 0/E53A8528
```

UPDATE/DELETE e REPLICA IDENTITY

La tabella creata come esempio non ha una chiave primaria, e quindi il sistema non sa come fare a riconoscere le tuple *non* aggiunte!

```
# DELETE FROM replicated;
ERROR:  cannot delete from table "replicated" because it does not have a replica id
HINT:  To enable deleting from the table, set REPLICA IDENTITY using ALTER TABLE.
```

Si può ovviare al problema assegnando una `REPLICA IDENTITY` alla tabella:

```
# ALTER TABLE replicated REPLICA IDENTITY FULL;
```

e quindi i comandi di `INSERT` e `UPDATE` iniziano a funzionare. La `REPLICA IDENTITY` può essere `DEFAULT` (usa i vincoli `UNIQYE` e quindi le chiavi trovate) o `FULL` (fa un flush dell'intera tupla).

Background Workers

I *background workers* sono processi esterni lanciati e gestiti da PostgreSQL.
A tali processi sono consentiti accessi alla shared memory, ai database e al sistema interno di PostgreSQL.
Devono aderire ad una specifica interfaccia affinché il cluster li possa gestire correttamente.
Introducono un potenziale rischio di stabilità e sicurezza!

Quando inizializzare un backgroud worker

I processi worker possono essere inizializzati all'avvio del cluster o in una fase successiva. La differenza è nella chiamata di funzione per la registrazione del worker stesso:

- `RegisterBackgroundWorker(BackgroundWorker *worker)` nel caso di avvio assieme al cluster;
- `RegisterDynamicBackgroundWorker(BackgroundWorker *worker, BackgroundWorkerHandle **handle)` per inizializzazione in un qualunque momento di vita del cluster.

La struttura BackgroundWorker

La struttura C `BackgroundWorker` fornisce le informazioni di gestione del processo worker stesso, quali ad esempio:

- nome del processo;
- flags (ad esempio `BGWORKER_SHMEM_ACCESS` per accedere alla shared memory, `BGWORKER_BACKEND_DATABASE_CONNECTION` per richiedere connessione ad un database e la capacità di eseguire transazioni);
- istante di avvio:
  - `BgWorkerStart_PostmasterStart` avvia il prima possibile;
  - `BgWorkerStart_ConsistentState` avvia quando sono possibili le connessioni utente (ad esempio in *hot standby*);
  - `BgWorkerStart_RecoveryFinished` avvia quando è possibile accedere in lettura/scrittura al cluster.
- istante di riavvio: indica dopo quanti secondi PostgreSQL deve riavviare il processo a seguito di un crash del processo stesso, o `BGW_NEVER_RESTART`;
- funzione da richiamare (e `Datum` come argomenti);
- eventuale pid di un processo PostgreSQL a cui inviare `SIGUSR1` quando il worker viene avviato o terminato

alive_worker

`alive_worker` è uno *scheletro* di background worker ispirato da `worker_spi` (disponibile nell'albero dei sorgenti di PostgreSQL) il cui scopo è quello di inserire una riga di *log* in una *log table* ogni 10 secondi, simulando quindi il logging di una attività continua.

Makefile

Makefile:

```
MODULES = alive_worker

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

Installare e attivare il worker

Occorre compilarlo, installarlo, listarlo nelle shared libraries e riavviare il cluster:

```
% gmake && sudo gmake install
```

Nel file `postgresql.conf` occorre inserire la libreria condivisa `alive_worker` (corrispondente ad `alive_worker.so`) e almeno un worker abilitato:

```
shared_preload_libraries = 'alive_worker'
max_worker_processes = 8
```

Avvio del worker

Quando viene riavviato il cluster, il worker si registra e inizia il suo lavoro. Nei log di PostgreSQL si può trovare qualcosa di simile:

```
INFO:   alive_worker::_PG_init registering worker [alive worker]
DEBUG:  registering background worker "alive worker"
DEBUG:  loaded library "alive_worker"
DEBUG:  starting alive_worker::worker_main
DEBUG:  alive_worker in main loop
DEBUG:  alive_worker executing query [INSERT INTO public.log_table( message ) VALUE
DEBUG:  alive_worker query OK!
DEBUG:  alive_worker in main loop
DEBUG:  alive_worker executing query [INSERT INTO public.log_table( message ) VALUE
DEBUG:  alive_worker query OK!
DEBUG:  alive_worker in main loop
```

Worker in funzione

Durante la sua vita il worker andrà ad inserire in una apposita tabella `log_table` un messaggio di log con incremento:

```
 > SELECT * FROM log_table;
 pk |            message             |             ts
----|--------------------------------|----------------------------
  4 | STILL ALIVE and counting! 1    | 2017-11-09 15:05:09.446884
  5 | STILL ALIVE and counting! 2    | 2017-11-09 15:05:19.44873
  6 | STILL ALIVE and counting! 3    | 2017-11-09 15:05:29.52069
  7 | STILL ALIVE and counting! 4    | 2017-11-09 15:05:39.527419
  8 | STILL ALIVE and counting! 5    | 2017-11-09 15:05:49.557702
  9 | STILL ALIVE and counting! 6    | 2017-11-09 15:05:59.624813
```

Interagire con il worker

Il worker si registrerà nella process table del sistema operativo con il nome `alive_worker` e può quindi essere terminato con un segnale `SIGTERM`:

```
% ps -auxw | grep alive_worker
 postgres 3312   0.0  5.4 201368 26636  -  Ss   15:07     0:00.01 postgres: bgworke

% sudo kill -TERM 3312
```

Un estratto di codice del worker

Si crea una struttura BackgroundWorker con i relativi dati:

```
void _PG_init(void)
{
  BackgroundWorker worker;
  worker.bgw_flags        = BGWORKER_SHMEM_ACCESS
                          | BGWORKER_BACKEND_DATABASE_CONNECTION;
  worker.bgw_start_time   = BgWorkerStart_RecoveryFinished;
  worker.bgw_restart_time = BGW_NEVER_RESTART;
  worker.bgw_main         = worker_main;
  snprintf(worker.bgw_name, BGW_MAXLEN, "alive worker");
  worker.bgw_notify_pid   = 0;
  ...
  RegisterBackgroundWorker( &worker );
}
```

Un estratto del codice del worker

Il loop principale del worker nella funzione `worker_main` effettua:

```
while ( ! sigterm_activated )
{
  int         ret;
  int         rc;

  rc = WaitLatch( &MyProc->procLatch,
                  WL_LATCH_SET | WL_TIMEOUT | WL_POSTMASTER_DEATH,
                  10000L ); /* 10 secondi */
  ResetLatch( &MyProc->procLatch );

  SetCurrentStatementStartTimestamp();
  StartTransactionCommand();
  SPI_connect();
  PushActiveSnapshot( GetTransactionSnapshot() );
  pgstat_report_activity( STATE_RUNNING, queryStringData.data );

  resetStringInfo( &queryStringData );
  appendStringInfo( &queryStringData,
                    "INSERT INTO %s.%s( message ) "
                    "VALUES( '%s %d' ) ",
                    log_table_schema,
                    log_table_name,
                    log_table_message,
```

Installazione

1. scaricare il pacchetto *wheel*;
2. installare con pip dopo aver creato un *virtual environment*:

```
% sudo pip install virtualenv virtualenvwrapper
% source /usr/local/bin/virtualenvwrapper.sh
% mkvirtualenv pgadmin4
% pip install ./pgadmin4-2.0-py2.py3-none-any.whl
```

1. avviare il demone:

```
% sudo python ~/.virtualenvs/pgadmin4/lib/python2.7/site-packages/pgadmin4/pgAdmin4
NOTE: Configuring authentication for SERVER mode.

Enter the email address and password to use for the initial pgAdmin user account:

Email address: fluca1978@gmail.com
Password:
Retype password:
pgAdmin 4 - Application Initialisation
======================================

Starting pgAdmin 4. Please navigate to http://127.0.0.1:5050 in your browser.
```

Usare pgadmin4

Occorre collegarsi con un browser all'indirizzo web
`http://localhost:5050`.
Si forniscono le credenziali specificate all'atto del primo avvio di pgadmin4 e si entra nell'interfaccia grafica.

Installazione

Esiste un fork di PgAdminIII.

1. scaricare l'albero dei sorgenti:

```
% git clone https://bitbucket.org/openscg/pgadmin3-lts
```

1. installare *wxWidgets* almeno versione 2.8:

```
% tar xjvf wxWidgets-3.1.0.tar.bz2 && cd wxWidgets-3.1.0
% ./configure --prefix=/opt/pgadmin310lts \
              --with-wx=/opt/wxwidgets310 \
              --with-wx-version=3.1
% make && sudo make install
```

Buona fortuna! #+begni$_{src}$ shell Makefile:478: recipe for target 'all' failed make: * [all] Error 2 #+end$_{src}$

Page Checksum

Se il cluster è stato inizializzato opportunamente (opzione `-k` di `initdb`) ogni pagina dati ha associato un *checksum*. Tale checksum viene controllato ogni volta che la pagina dati viene portata da disco nello shared buffer, se il checksum non corrisponde PostgreSQL segnala l'anomalia e si rifiuta di leggere i dati da quella pagina.
Se l'opzione non è abilitata PostgreSQL non verifica l'integrità della pagina e quindi può inserire in memoria dati sporchi.

Individuare una tabella da corrompere

La seguente query identifica una tabella utente (non pg*) ordinando per numero di pagine dati:

```
# SELECT relname, relpages, reltuples, relfilenode
FROM pg_class WHERE relkind = 'r' AND relname NOT LIKE 'pg%'
Order BY relpages DESC;
-[ RECORD 1 ]------------------------
relname     | evento
relpages    | 13439
reltuples   | 2.11034e+06
relfilenode | 19584
```

La tabella evento ha quindi 13439 pagine dati, è una buona candidata per una possibile corruzione.

```
# select pg_relation_filepath( 'evento'::regclass );
-[ RECORD 1 ]--------+-----------------
pg_relation_filepath | base/19554/19584
```

La tabella si trova quindi nel file $PGDATA/base/19554/19584.

Corrompere una tabella

Il seguente semplice programma Perl corrompe i dati oltre ai primi 8kB (quindi la seconda pagina):

```
#!env perl

open my $db_file, "+<", $ARGV[ 0 ] || die "Impossibile aprire il file!\n\n";
seek $db_file, ( 8 * 1024 ) + 20, 0;

print { $db_file } "Hello Corrupted Database!";
close $db_file;
```

E quindi per corrompere il database è sufficiente:

```
% sudo service postgresql stop
% sudo perl corrupt.pl /mnt/data1/pgdata/base/19554/19584
% sudo service postgresql start
```

Vedere la tabella corrotta

Quando il sistema si trova a fronteggiare la pagina corrotta informa del problema:

```
> SELECT * FROM evento;
...
ERROR:  invalid page in block 1 of relation base/19554/19584
```

Come recuperare i dati corrotti?

Non c'è modo!

Ad ogni modo PostgreSQL consente una opzione particolare (solitamente non inclusa in `postgresql.conf`) denominate:

- `zero_damaged_pages` azzera una pagina dati il cui checksum non è corretto. Questo consente di riutilizzare la pagina come "pulita";
- `ignore_checksum_failure` se impostata a `false` non visualizza l'errore quando si incontra una pagina dati corrotta, e questo *è molto periocoloso perché impedisce di accorgersi del danneggiamento dei dati*.

Nel caso specifico basta:

```
# SET zero_damaged_pages TO 'on';
-- interrogare di nuovo la tabella, questa
-- volta viene lanciato un warning ma la query non
-- viene interrotta
```

Nei log PostgreSQL segnala che ha azzerato la pagina dati:

```
WARNING:  page verification failed, calculated checksum 61489 but expected 61452
WARNING:  invalid page in block 1 of relation base/19554/19584; zeroing out page
```

Risultato di aver azzerato la pagina corrotta

Se si esegue un `VACUUM` (o si attende che scatti l'autovacuum) e si controlla la dimensione della tabella si ottiene che:

```
# SELECT relname, relpages, reltuples, relfilenode
   FROM pg_class
   WHERE relkind = 'r' AND relname NOT LIKE 'pg%'
Order BY relpages DESC;
-[ RECORD 1 ]------------------------
relname     | evento
relpages    | 13438
reltuples   | 2.11015e+06
relfilenode | 19841
```

Ovvero la tabella è stata diminuita di esattamente una pagina dati.

Vacuum ripulisce le pagine corrotte

Se l'opzione zero_damaged_pages è attivata un `VACUUM FULL` procede a ripulire le pagine dati:

```
# SET zero_damaged_pages TO 'on';

# VACUUM FULL VERBOSE evento;
INFO:  vacuuming "public.evento"
WARNING:  page verification failed, calculated checksum 22447 but expected 19660
WARNING:  invalid page in block 1 of relation base/19554/19857; zeroing out page
INFO:  "evento": found 0 removable, 2109837 nonremovable row versions in 13437 page
```

Autovacuum non ripulisce le pagine corrotte

Il processo di autovacuum, contrariamente a `VACUUM`, ignora l'opzione `zero_damaged_pages` anche se impostato. Questo perché l'ozione è considerata pericolosa, e sbirciando nel codice sorgente di autovacuum si trova che:

```
/*
 * Force zero_damaged_pages OFF in the autovac process, even if it is set
 * in postgresql.conf.  We don't really want such a dangerous option being
 * applied non-interactively.
 */
SetConfigOption("zero_damaged_pages", "false", PGC_SUSET, PGC_S_OVERRIDE);
```

Installazione

Ad esempio su una macchina FreeBSD:

```
% sudo pkg install pgbouncer
```

La configurazione si trova in un file `pgbouncer.ini` (ad esempio `/usr/localt/etc/pgbouncer.ini`).

Configurazione: pgbouncer.ini

Il fine di configurazione principale è pgbouncer.ini.
La sezione pgbouncer contiene informazioni *globali* per l'amministrazione stessa.
La sezione databases contiene le configurazioni dei singoli database.

Esecuzione di pgbouncer

Occorre ricordare che:

1. è sempre obbligatorio passare il file di configurazione da usare con l'opzione `-d`;
2. non può essere eseguito come `root`.

Occorre che l'utente che eseguirà `pgbouncer` abbia i diritti sulle directory di log e pid!

```
% sudo -u postgres pgbouncer -d /usr/local/etc/pgbouncer.ini
```

Riavvio di pgbouncer

Per riavviare il demone di pgbouncer occorre usare l'opzione `-R`:

```
% sudo -u postgres pgbouncer -d /usr/local/etc/pgbouncer.ini -R
```

Configurazione della sezione pgbouncer

```
[pgbouncer]
logfile = /var/log/pgbouncer/pgbouncer.log
pidfile = /var/run/pgbouncer/pgbouncer.pid

listen_addr = 127.0.0.1
listen_port = 6432

auth_type = trust
auth_file = /usr/local/etc/pgbouncer.userlist.txt

pool_mode = session
server_reset_query = DISCARD ALL
max_client_conn = 20
default_pool_size = 20
```

Autenticazione

`pgbouncer` usa un file di autenticazione esterno, identificato dalla variabile `auth_file` che contiene per ogni riga lo username e la password, entrambi racchiusi fra doppi apici. E' possibile non specificare la password se la modalità è `trust`.
Ad esempio (`/usr/local/etc/pgbouncer.userlist.txt`):

```
"luca" ""
"boss" ""
```

Il tipo di autenticazione `auth_type` può essere principalmente:

- `trust` (accetta l'utente se è listato);
- `md5` la password deve essere la stessa nel file di autenticazione;

Pool Mode

`pgbouncer` supporta tre modalità di *pooling*, specificate come `pool_mode`:

- `session` indica che una connessione viene rilasciata quando il client si disconnette;
- `transaction` al termine di ogni transazione viene rilasciata la connessione;
- `statement` al termine di ogni singolo statement.

Tipicamente si usa `session` o `transaction`. Il numero di connessioni da tenere nel pool è indicato da `default_pool_size`, ma fino a `max_client_conn` possono essere create.
Ogni volta che una connessione viene restituita al pool viene eseguita la `server_reset_query` al fine di resettare variabili di sessioni, cursori, ecc.

Lo pseudo database pgbouncer

E' possibile collegarsi ad un database fittizio denominato pgbouncer. Questo consente di usare statement SQL per comandare il sistema di pooling.
Per potersi connettere a tale database occorre listare gli utenti nella variabile admin_users. Questi non devono essere utenti reali, basta che siano listati nel file di autenticazione di pgbouncer stesso:

```
# pgbouncer.ini
admin_users = luca, pgbouncer_admin
```

Nel database pgbouncer è possibile dare il comando SHOW HELP per ottenere informazioni su quali comandi sono messi a disposizione:

```
% psql -h localhost -p 6432 -U pgbouncer_admin pgbouncer
# SHOW HELP;
NOTICE:  Console usage
DETAIL:
        SHOW HELP|CONFIG|DATABASES|POOLS|CLIENTS|SERVERS|VERSION
        SHOW STATS|FDS|SOCKETS|ACTIVE_SOCKETS|LISTS|MEM
        SHOW DNS_HOSTS|DNS_ZONES
        SET key = arg
        RELOAD
        PAUSE [<db>]
```

SHOW CONFIG

Illustra tutta la configurazione così come è stata caricata dal file di configurazione.
Utile per introspezione.

SHOW DATABASES

Il comando SHOW DATABASES mostra i database configurati, lo stato del pool e a quale database puntano effettivamente:

```
# SHOW DATABASES;
name                | testdb
host                | localhost
port                | 5432
database            | devdb
force_user          |
pool_size           | 20
reserve_pool        | 0
pool_mode           |
max_connections     | 0
current_connections | 1
```

SHOW CLIENTS

Mostra quali client sono attualmente connessi, a cosa e da dove:

```
# SHOW CLIENTS;
type          | C
user          | luca
database      | testdb
state         | active
addr          | 127.0.0.1
port          | 39019
local_addr    | 127.0.0.1
local_port    | 6432
connect_time  | 2018-02-02 14:15:44
request_time  | 2018-02-02 14:17:55
ptr           | 0x801d15490
link          | 0x801cd1ed0
remote_pid    | 0
```

SHOW STATS

Mostra statistiche sulle query servite

```
# SHOW STATS;
database          | testdb
total_requests    | 1
total_received    | 838
total_sent        | 92737
total_query_time  | 3407336
avg_req           | 0
avg_recv          | 0
avg_sent          | 0
avg_query         | 0
```

ATTENZIONE: il `database` è quello a cui ci è connessi mediante `pgpool`, non quello effettivo!

SHOW POOLS

Mostra lo stato del pool per ogni database configurato:

```
# SHOW POOLS;
database    | testdb
user        | luca
cl_active   | 1
cl_waiting  | 0
sv_active   | 1
sv_idle     | 0
sv_used     | 0
sv_tested   | 0
sv_login    | 0
maxwait     | 0
pool_mode   | session
```

PAUSE e RESUME

Il comando `PAUSE` blocca l'esecuzione verso un database: dal momento in cui scatta il comando (non ci devono essere connessioni fuori dal pool!) ogni nuova connessione viene accettata, ma l'esecuzione non viene eseguita. Si può riprendere con `RESUME`.

pgbouncer	testdb
PAUSE testdb;	
	psql -h localhost -U luca -p 6432 testdb
	– connesso!
	SELECT * FROM pg_{class};
	– bloccato!
RESUME testdb;	– sbloccato!

DISABLE e ENABLE

Simili a PAUSE e RESUME ma bloccano le connessioni direttamente fornendo il messaggio che il database non esiste:

```
% psql -h localhost -p 6432 -U luca testdb
psql: ERROR:  database does not allow connections: testdb
```

KILL

Il modo drastico di terminare i backend: dopo un KILL i client ricevono il messaggio come se il server si fosse spento.

```
> select * from pg_class;
ERROR:  database removed
server closed the connection unexpectedly
        This probably means the server terminated abnormally
        before or while processing the request.
The connection to the server was lost. Attempting reset:
```

RELOAD

Ricarica la configurazione senza interrompere il servizio.

SHUTDOWN

Interrompe il servizio di pgpool.

```
# SHUTDOWN;
server closed the connection unexpectedly
        This probably means the server terminated abnormally
        before or while processing the request.
The connection to the server was lost. Attempting reset: Failed.
```

Rimbalzo da un database ad un altro

Si supponga di voler rimbalzare ogni connessione al database `testdb` a quello `devdb`:

```
# pgbouncer.ini
[databases]
testdb = host=localhost dbname=devdb
```

e la connessione:

```
% psql -h localhost -p 6432 -U luca testdb
[luca @ testdb on localhost] 1  > SELECT current_database();
 current_database
------------------
 devdb
```

Si noti che ci si collega a `testdb` e anche il prompt di psql riporta ciò, ma in realtà il database corrente è `devdb`!

Pooling su tutti i databaase

E' sufficiente usare il nome particolare * come database per effettuare il pooling su ogni database:

```
* = host=localhost port=5432
```

Connessioni con pool specifico

E' possibile specificare per ogni database parametri di pool che sovrascrivono quelli già presenti nel default:

```
testdb = host=localhost port=5432 pool_size=100
```

Rimappare utenti in connessione

E' possibile fare in modo che ogni utente che si collega al database sia *rimappato* in un altro utente:

```
testdb = host=localhost port=5432 user=postgres pool_size=100
```

e quindi

```
% psql -h localhost -p 6432 -U luca testdb
# SELECT current_user;
 current_user
--------------
 postgres
```