
HTTP://GIPSYLINUX.WORDPRESS.COM/

Panduan
Administrasi Database PostgreSQL

Versi 1.4

BUKU 1

Sangkalan

Pada Intinya Perintah Dasar SQL Termasuk Administrasi Database PostgreSQL Adalah Identik, Tetapi Untuk GUI dan Perintah Lanjut Buku Pada Buku Ini Mungkin Tidak Kompatibel Lagi Dengan Versi Database Terbaru atau versi yang Anda Gunakan

Artikel dan e-book lainnya dapat di peroleh di www.yuliardi.com

Written and Published by

Rofiq Yuliardi
Web : www.yuliardi.com
Email : rofiq@yuliardi.com , rofiqy@gmail.com
YahooID : rofiqy2000
Phone : 0852-160-88127

DAFTAR ISI

PENGANTAR POSTGRESQL

BAB 1 – Pengantar PostgreSQL

Deskripsi

Bab ini merupakan bab pertama yang akan membahas pengertian database PostgreSQL, instalasi dan konfigurasinya. Bab pertama ini juga akan membahas distribusi file sistem dalam PostgreSQL. Bab ini terdiri dari beberapa subbab yang membahas proses instalasi server database PostgreSQL secara detai baik pada sistem perasi windows maupun linux.

Obyektif

Tujuan dari bab ini adalah agar peserta dapat memahami konsep dasar tentang database PostgreSQL. Peserta juga diharapkan dapat memahami dan mengerti proses instalasi database server PostgreSQL baik untuk sistem operasi Windows maupun Linux, termasuk didalamnya melakukan konfigurasi pada sistem.

Outline

- PostgreSQL
- Kelebihan PostgreSQL
- Instalasi Database PostgreSQL
- Sistem Distribusi File
- Instalasi Paket RPM
- Instalasi Paket tarball
- Membuat Database Cluster
- Konfigurasi Skrip SysV
- Instalasi di Windows XP

Tentang PostgreSQL

PostgreSQL merupakan Sebuah Obyek-Relasional Data Base Management System (ORDBMS) yang dikembangkan oleh Berkeley Computer Science Department. System yang ditawarkan PostgreSQL diharapkan sanggup dan dapat mencukupi untuk kebutuhan proses aplikasi data masa depan. PostgreSQL juga menawarkan tambahan-tambahan yang cukup signifikan yaitu *class, inheritance, type, dan function*. Tambahan keistimewaan lain yang tidak dimiliki *database management system* yang lain berupa *constraint, triggers, rule, dan transaction integrity*, dengan adanya *feature* (keistimewaan) tersebut maka para pemakai dapat dengan mudah mengimplementasikan dan menyampaikan sistem ini. Sejak tahun 1996 PostgreSQL mengalami kemajuan yang sangat berarti, berbagai keistimewaan dari PostgreSQL sanggup membuat *database* ini melebihi *database* lain dari berbagai sudut pandang.

Pada awal pembuatannya di University of California Berkeley (1977-1985) postgresl masih mempunyai banyak kekurangan bila dibandingkan dengan *database* yang lain, namun seiring dengan berjalannya waktu tepatnya pada tahun 1996 PostgresSQL berubah menjadi sebuah *database* yang menawarkan standar melebihi standar ANSI-SQL92 dan sanggup memenuhi permintaan dunia *open source* akan *server database* SQL. Standar ANSI-SQL92 merupakan standar yang ditetapkan untuk sebuah *database* berskala besar seperti Oracle, Interbase, DB2 dan yang lainnya.

Kelebihan PostgreSQL

Berbeda dengan *database* lain, PostgreSQL menyediakan begitu banyak dokumentasi yang disertakan pada berbagai distribusi Linux, sehingga para pembaca bisa dengan mudah mempelajari bahkan mengimplementasikannya. Tidak hanya itu berbagai dokumentasi yang bertebaran di Internet maupun *mailing list* yang semuanya dapat kita ambil dan pelajari. PostgreSQL memiliki keluwesan dan kinerja yang tinggi, artinya sesuai dengan niatan awal para pembuat PostgreSQL bahwa *database* yang mereka buat harus melebihi *database* lain dan ini terbukti pada arsitekturnya. Dengan arsitektur yang luwes maka sebuah *user* PostgreSQL mampu mendefenisikan sendiri SQL-nya, inilah yang membuat *database* PostgreSQL berbeda dengan sistem relasional standar. Di samping mendefenisikan sendiri SQL-nya, PostgreSQL juga memungkinkan setiap *user* untuk membuat sendiri *object file* yang dapat diterapkan untuk mendefenisikan tipe data, fungsi dan bahasa pemrograman yang baru sehingga PostgreSQL sangat mudah dikembangkan maupun di implementasikan pada tingkat *user*. PostgreSQL versi 7.0.x dan versi di atasnya menyertakan dokumentasi maupun berbagai macam contoh pembuatan fungsi maupun sebuah prosedur. Dengan keluwesan dan fitur yang dimilikinya, PostgreSQL patut bahkan melebihi jika disandingkan dengan *database* yang berskala besar lainnya. Jika kita menggunkan sebuah *database*, tentunya tak lepas dari tujuan dan maksud apa yang ingin dicapai serta kelebihan yang bagaimana yang kita inginkan.

PostgreSQL juga mendukung beberapa fitur database modern, antar lain;

- complex queries
- foreign keys

- triggers
- views
- transactional integrity
- multiversion concurrency control

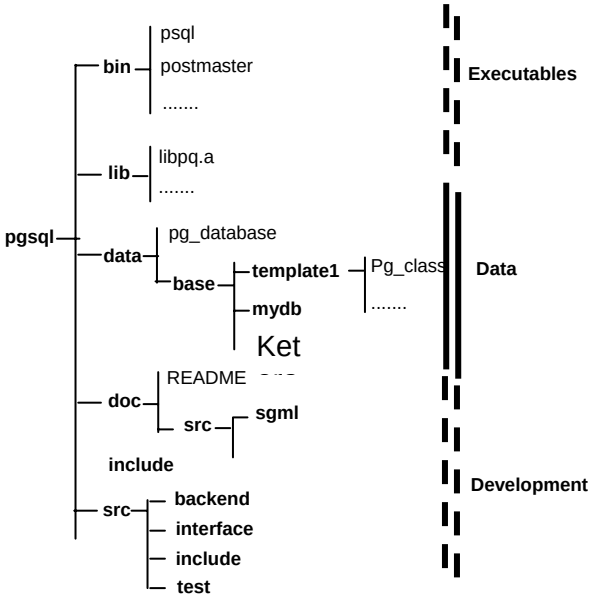
Selain itu PostgreSQL juga dapat di extend sesuai kebutuhan pengguna melalui beberapa metode dengan menambangkan obyek baru, seperti

- Penambahan Tipe Data
- Penambahan Fungsi
- Penambahan Operator
- Penambahan Fungsi Aggregate
- Metode Index
- Bahasa prosedural

Instalasi Database PostgreSQL

Pada bagian ini, akan dibahas cara instalasi PostgreSQL yaitu dengan menggunakan paket yang telah ada pada CD distribusi Linux. Biasanya paket tersebut telah dikompilasi dalam bentuk RPM, yang kedua dengan cara mengkompilasi paket *tarball* PostgreSQL dalam format **tar.gz** atau **tar.bz2** serta yang ketiga adalah instalasi PostgreSQL pada sistem operasi Windows.

Sistem Distribusi File



Gambar Pendistribusian File

Pada gambar diatas proses distribusi PostgreSQL ketika dinstall secara default. Untuk lebih mudahnya kita anggap bahwa PostgreSQL telah diinstall dan terdapat pada direktori **/usr/local/src/pgsql**. Semua perintah (*command*) PostgreSQL terinstall di dalam direktori **/usr/local/src/pgsql/bin**.

Instalasi Paket RPM

Saat ini banyak CD Linux telah beredar dengan berbagai merk Distribusinya. Baik itu RedHat, Mandrake maupun yang lainnya, sehingga kita bisa langsung memilih paket yang dibutuhkan dan menginstalnya.

Untuk menginstal PostgreSQL ada 3 paket yang dibutuhkan, antara lain :

Postgresql-8.0.1-6.i586.rpm

Postgresql-devel-8.0.1-6.i586.rpm

Postgresql-server-8.0.1-6.i586.rpm

Berikut ini langkah-langkah penginstalan-nya.

```
# rpm -ivh Postgresql - 8.0.1-6.i586.rpm

# rpm -ivh Postgresql - devel - 8.0.1-6.i586.rpm

# rpm -ivh Postgresql - server -8.0.1-6.i586.rpm
```

Setelah semua paket tersebut terinstal, jalankan server PostgreSQL dari konsole (harus sebagai root) tuliskan perintah seperti di bawah ini kemudian tekan enter.

```
#  /etc/rc.d/init.d/postgres stop
    Stopping postgresql service:              [  OK  ]

#  /etc/rc.d/init.d/postgres start
   Checking postgresql installation:          [  OK  ]
```

```
Starting postgresql service:                [  OK  ]
```

Jika pada layar komputer anda muncul hasil seperti di atas, proses instalasi berhasil dan PostgreSQL siap digunakan. Namun, jika proses di atas mengalami kegagalan, pastikan file /tmp/ .s.PGSQL.5432 dihapus terlebih dahulu kemudian ulangi lagi proses menjalankan postgres seperti langkah di atas.

PostgreSQLjuga dapat dijalankan pada saat *startup*, untuk itu kita perlu mengkonfigurasi *ntsysv*. Dari konsole (prompt Linux) ketikan ntsysv.

```
#  ntsysv
```

berikut tampilan dari perintah tersebut.

Gambar Konfigurasi Service saat booting.

Beri tanda bintang (asterik) dengan menekan spasi pada kotak postgresql, klik OK. Kemudian, booting ulang komputer. Selanjutnya, database postgres tidak perlu dijalankan secara manual lagi, karena pada saat booting kita telah menjalankan service PostgreSQL secara otomatis.

Instalasi Paket tarball

Mengkompilasi paket tarball postgres juga merupakan salah satu cara untuk menginstall PostgreSQL. Untuk cara yang satu ini sangat menguntungkan sebab dengan mengkompilasi *source code*-nya sendiri, kita akan lebih luas mengetahui apa yang diinginkan dan dapat juga menambah atau meng-*uppgrade* postgres sesuai dengan keinginan kita. Paket ini dapat diperoleh pada CD Linux atau men-*download*-nya dari Internet di **http://www.postgresql.org/**.

File-file yang perlu disediakan sebelum melakukan proses kompilasi antara lain :

postgresql-base-8.0.1.tar.gz

Paket ini berisikan source code wajib sebagai build-in.

postgresql-docs-8.0.1.tar.gz

Paket ini berisikan dokumentasi tentang postgres dengan format HTML. Sebagai catatan, pada saat penginstalan paket *base*, *man page* secara otomatis akan terinstal juga.

Postgresql-opt-8.0.1.tar.gz

Paket *opt* berisi beberapa pilihan ekstensi sebagai *interface* untuk C++ (libpq++), JDBC, ODBC, Perl, Python, dan tcl. Selain itu, juga berisi source wajib untuk *multibyte support*.

Postgresql-test-8.0.1.tar.gz

Paket ini berisi sederetan *regression test*, peket ini wajib ada jika kita ingin menjalankan *regression test* setelah melakukan kompilasi postgres.

Sebelum melakukan proses kompilasi, periksalah terlebih dulu ruang harddisk yang tersisa, untuk mengkompilasi paket tarball di butuhkan ruang harddisk kurang lebih 30 Mbytes dan sekitar 5 Mbytes untuk instalasi direktorinya. Gunakan perintah berikut ini untuk memeriksa ruang harddisk yang masih kosong.

```
$  df -m

Filesystem  1M-blocks  Used   Available  Use%   Mounted on
/dev/hda3   2015        1438    475         76%        /
 none             93              0     93           0%      /dev/shm
/dev/hda1   1002          639    363         64%     /mnt/win_c
/dev/hda5   1002          662    340          67%    /mnt/win_d
```

Buatlah user postgres sebelum melakukan kompilasi:

```
$  su - -c "useradd postgres"
```

Loginlah sebagai **root** dan kopikan paket postgres tersebut ke dalam direktori **/usr/local/src**

```
$  su  -
password : <masukkan password root kemudian tekan enter>
```

Dari direktori /usr/local/src, ekstrak paket file tersebut:

```
# cd /usr/local/src
# tar -xzvf postgresql-8.0.1.tar.gz
postgresql-8.0.1/
postgresql-8.0.1/ChangeLogs/
postgresql-8.0.1/ChangeLogs/ChangeLog-8.1-8.1.1
postgresql-8.0.1/ChangeLogs/ChangeLog-8.1RC1-to-8.1RC2
postgresql-8.0.1/ChangeLogs/ChangeLog-8.1RC2-to-8.1RC3
postgresql-8.0.1/ChangeLogs/ChangeLog-8.1RC3-to-8.1rc4
postgresql-8.0.1/ChangeLogs/ChangeLog-8.1beta1-to-8.1beta3
postgresql-8.0.1/ChangeLogs/ChangeLog-8.1beta3-to-8.1beta4
postgresql-8.0.1/ChangeLogs/ChangeLog-8.1beta4-to-8.1beta5
postgresql-8.0.1/ChangeLogs/ChangeLog-8.1beta5-to-8.1beta6
postgresql-8.0.1/ChangeLogs/ChangeLog-8.1beta6-8.1RC1
postgresql-8.0.1/ChangeLogs/ChangeLog-8.1rc4-8.1
postgresql-8.0.1/ChangeLogs/ChangeLog-8.1.1-8.0.1
postgresql-8.0.1/COPYRIGHT
postgresql-8.0.1/config/
postgresql-8.0.1/config/ac_func_accept_argtypes.m4
[...........]
postgresql-8.0.1/src/utils/strdup.c
postgresql-8.0.1/src/win32.mak
postgresql-8.0.1/configure
postgresql-8.0.1/configure.in
postgresql-8.0.1/register.txt
```

ganti *owner* dan *group* dari direktori postgresql-8.0.1 yang baru terbentuk menjadi postgres.

```
#  ls -l
```

```
total 7944
drwxrwxrwx 7 1005  96   4096      May 24  2001  postgresql-8.0.1
-r--r--r--  1  root    root   8117016 Aug 15  09:09 postgresql-8.0.1.tar.gz

#  chown -R postgres.postgres  postgresql-8.0.1
#  ls -l
total 7944
drwxrwxrwx  7  postgres postgres   4096     May  24  2001  postgresql-8.0.1
r--r--r--    1  root    root   8117016  Aug  15   09:09 postgresql- 8.0.1.tar.gz
```

Dari direktori **postgresql-8.0.1**, login sebagai postgres *super user* dan lakukan konfigurasi.

```
#  cd postgresql-8.0.1
#  su postgres
```

Kemudian baru lakukan konfigurasi.

```
$  ./configure
creating cache ./config.cache
checking host system type... i686-pc-linux-gnu
checking which template to use... linux
checking whether to build with locale support... no
checking whether to build with recode support... no
checking whether to build with multibyte character support... no
checking whether to build with Unicode conversion support... no
checking for default port number... 5432
checking for default soft limit on number of connections... 32
checking for gcc... gcc
checking whether the C compiler (gcc  ) works... yes
checking whether the C compiler (gcc  ) is a cross-compiler... no
checking whether we are using GNU C... yes
checking whether gcc accepts -g... yes
linking ./src/backend/port/dynloader/linux.c to src/backend/port/dynloader.c
linking ./src/backend/port/dynloader/linux.h to src/include/dynloader.h
linking ./src/include/port/linux.h to src/include/os.h
linking ./src/makefiles/Makefile.linux to src/Makefile.port
linking ./src/backend/port/tas/dummy.s to src/backend/port/tas.s
```

Lakukan proses kompilasi dengan perintah **gmake**.

```
$  gmake
gmake -C doc all
gmake[1]: Entering directory `/usr/local/src/postgresql-8.0.1/doc'
gmake[1]: Nothing to be done for `all'.
gmake[1]: Leaving directory `/usr/local/src/postgresql-8.0.1/doc'
gmake -C src all
gmake[1]: Entering directory `/usr/local/src/postgresql-8.0.1/src'
gmake -C backend all
[........]
gmake[4]: Leaving directory `/usr/local/src/postgresql-8.0.1/src/pl/plpgsql/src'
gmake[3]: Leaving directory `/usr/local/src/postgresql-8.0.1/src/pl/plpgsql'
gmake[2]: Leaving directory `/usr/local/src/postgresql-8.0.1/src/pl'
gmake[1]: Leaving directory `/usr/local/src/postgresql-8.0.1/src'
All of PostgreSQL successfully made. Ready to install.
```

Kemudian, buat *test regression*, langkah ini bersifat opsional tetapi disarankan untuk melakukannya.

```
$  gmake check
gmake -C doc all
gmake[1]: Entering directory /usr/local/src/postgresql-7.1.3/doc'
gmake[1]: Nothing to be done for all'.
gmake[1]: Leaving directory /usr/local/src/postgresql-7.1.3/doc'
[...]
```

Setelah proses konfigurasi dan kompilasi dilakukan, lanjutkan ke tahap proses installasi, berikut perintahnya.

```
$ su -c "gmake install"
Password: < masukkan password root >
gmake -C doc install
gmake[1]: Entering directory /usr/local/src/postgresql-7.1.3/doc'
mkdir /usr/local/pgsql
mkdir /usr/local/pgsql/man
mkdir /usr/local/pgsql/doc
mkdir /usr/local/pgsql/doc/html
[...]

$  su -c "chown -R postgres.postgres  /usr/local/pgsql"
Password :  < masukkan Password root >

$ su -c "gmake -C src/interfaces/perl5  install"
password :  < masukkan Password root >
gmake:Enteringdirectory/usr/local/src/postgresql-1.3
/src/interfaces/perl5'
perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for Pg
gmake -f Makefile clean
[...]

$ su -c "gmake -C  src/interfaces/python install"
Password:  < masukkan Password root >
gmake: Entering directory
/usr/local/src/postgresql-7.1.3/src/interfaces/python'
sed -e 's,@libpq_srcdir@,../../../src/interfaces/libpq,g' \
-e 's,@libpq_builddir@,../../../src/interfaces/libpq,g' \
-e 's%@EXTRA_LIBS@% -lz -lcrypt -lresolv -lnsl -ldl -lm -lbsd   -lreadline     -ltermcap
%g' \
-e 's%@INCLUDES@%-I../../../src/include%g' \
[......................]
```

file headers juga dapat diinstal pada kesempatan ini, file ini sangat penting sebab instalasi default hanya akan menginstall file headers untuk pengembangan aplikasi client. Jadi, file headers dibutuhkan untuk pengembangan aplikasi dengan bahasa C yang menggunakan library libpq. Berikut perintahnya.

```
$  su -c "gmake install-all-headers"
Password: < masukkan Password root >
gmake -C src install-all-headers
gmake[1]: Entering directory /usr/local/src/postgresql-7.1.3/src'
gmake -C include install-all-headers
[.....................]
```

Login lagi sebagai **root** kemudian lakukan langkah berikutnya dengan menambahkan baris di bawah ini ke dalam file **/etc/profile**

```
$ su
Password: < masukkan Password root >
[root@localhost postgresql-8.0.1]#   vi  /etc/profile
```

Tambahkan file berikut ini.

```
PATH=$PATH:/usr/local/pgsql/bin
MANPATH=$MANPATH:/usr/local/pgsql/man
export PATH MANPATH
```

Pada default, instalasi bagian library akan terlihat dalam direktori */usr/local/pgsql/lib* (ini mungkin berbeda, bergantung apakah pada saat konfigurasi kita memilih opsi --prefix) salah satunya menetapkan LD_LIBRARY_PATH pada lingkungan variabel ke **/usr/local/pgsql/lib**. Berikut langkah-langkahnya.

```
$  LD_LIBRARY_PATH=/usr/local/pgsql/lib
$  export LD_LIBRARY_PATH
```

Membuat Database Cluster

Setelah proses kompilasi dilakukan, postgres masih belum bisa dijalankan, dikarenakan belum terbentuknya direktori yang berfungsi sebagai tempat diletakkannya file-file konfigurasi. Berikut perintah untuk membuat database culster postgres.

```
$  /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
```

Opsi -D pada perintah di atas menunjukkan lokasi di mana file-file konfigurasi akan disimpan. Lokasi ini dapat juga ditetapkan dengan PGDATA, jika anda menetapkan PGDATA, opsi –D tidak diperlukan lagi. Jika ingin menggunakan direktori yang berbeda untuk penanganan data file ini, pastikan user anda dapat menulis ke direktori tersebut. Berikut ini Output dari *initdb*:

```
$ /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data

This database system will be initialized with username "postgres."
This user will own all the data files and must also own the server process.

Creating directory /usr/local/pgsql/data
Creating directory /usr/local/pgsql/data/base
Creating directory /usr/local/pgsql/data/global
Creating directory /usr/local/pgsql/data/pg_xlog
Creating template1 database in /usr/local/pgsql/data/base/1
DEBUG:  database system was shut down at 2001-08-24 16:36:35 PDT
DEBUG:  CheckPoint record at (0, 8)
DEBUG:  Redo record at (0, 8); Undo record at (0, 8); Shutdown TRUE
DEBUG:  NextTransactionId: 514; NextOid: 16384
DEBUG:  database system is in production state
Creating global relations in /usr/local/pgsql/data/global
DEBUG:  database system was shut down at 2001-08-24 16:36:38 PDT
DEBUG:  CheckPoint record at (0, 108)
DEBUG:  Redo record at (0, 108); Undo record at (0, 0); Shutdown TRUE
DEBUG:  NextTransactionId: 514; NextOid: 17199
DEBUG:  database system is in production state
Initializing pg_shadow.
Enabling unlimited row width for system tables.
Creating system views.
Loading pg_description.
Setting lastsysoid.
Vacuuming database.
Copying template1 to template0.
Success. You can now start the database server using:

/usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data
or
/usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data -l logfile start
```

Setelah proses di atas selesai, lanjutkan dengan menjalankan *postmaster* pada *foreground* :

```
$  /usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data
DEBUG:  database system was shut down at 2001-10-12 23:11:00 PST
DEBUG:  CheckPoint record at (0, 1522064)
DEBUG:  Redo record at (0, 1522064); Undo record at (0, 0); Shutdown       TRUE
DEBUG:  NextTransactionId: 615; NextOid: 18720
```

```
DEBUG:  database system is in production state
```

Sampai disini kita telah berhasil menjalankan database postgres, kita juga dapat menjalankan *postmaster* pada *background*. Gunakan *pg_ctl* untuk menjalankan service *postmaster*, berikut perintahnya

```
$  /usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data -l /tmp/pgsql.log start
postmaster successfully started
```

Catatan: Aplikasi *pg_ctl* dapat digunakan untuk **start** dan **stop** *service* PostgreSQL , maksud dari **($)** adalah sebagai **user biasa** sedangkan **(#)** sebagai **root**.

Konfigurasi Skrip SysV

Paket tarball yang telah diekstrak disertakan juga dengan skrip postgresql, terletak dalam direktori **/usr/local/src/postgresql-8.0.1/.** Untuk itu, kita perlu menyalin skrip tersebut dari *contrib/start-scripts* ke direktori *init.d*, berikut perintahnya.

```
$  cd /usr/local/src/postgresql-8.0.1/
$  su -c "cp contrib/start-scripts/linux /etc/rc.d/init.d/postgresql"
```

Agar skrip itu dapat dieksekusi atau dijalankan, kita perlu mengubah ijin akses filenya, berikut perintahnya.

```
$ su  -c "chmod a+x  /etc/rc.d/init.d/postgresql"
```

setelah itu, jalankan service postgres.

```
$ service postgresql start
Starting PostgreSQL: ok
$ service postgresql stop
Stopping PostgreSQL: ok
```

Service postgres tersebut dapat juga dijalankan menggunakan perintah yang umumnya digunakan untuk menjalankan service-service yang lainnya. Login sebagai root dan jalankan service postgres.

```
#  /etc/rc.d/init.d/postgresql start
    Checking postgresql installation:            [  OK  ]
    Starting postgresql service:                 [  OK  ]

#  /etc/rc.d/init.d/postgresql stop
    Stopping postgresql service:                 [  OK  ]
```

Sampai disini, kita telah berhasil menginstal PostgreSQLdari dari paket tarball dengan cara mengkompilasinya.

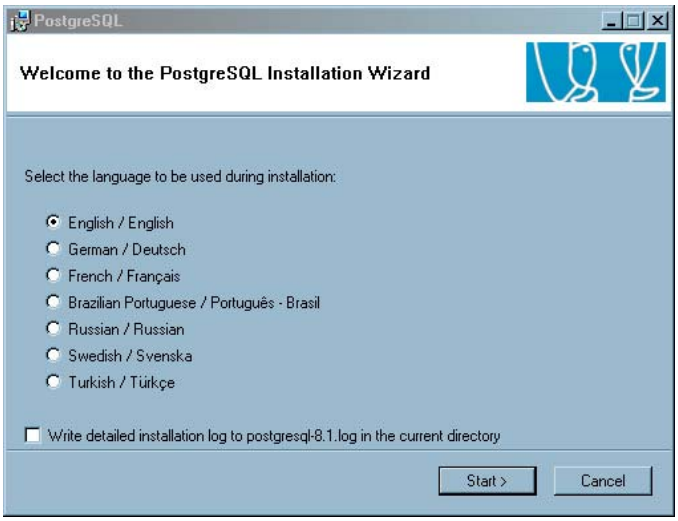
Instalasi di Windows XP

Cara instalasi di atas dilakukan pada sistem operasi linuk, pada bagian ini akan ditunjukan instalasi postgres pada sistem operasi windows. File installer sistem operasi windows dapat di download di web utama postgresql http://www.postgresql.org. File installer posgres untuk windows terdiri dari dua file yaitu

- postgresql-8.1.msi dan
- postgresql-8.1-int.msi

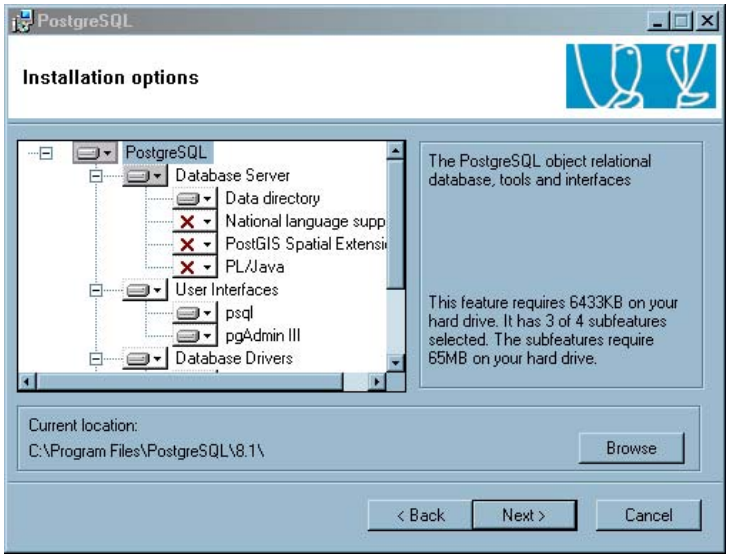
Setelah file di download kemudian ekstrak dan lakukan instalasi dengan klik dua kali pada

file installer tersebut. Setelah dijalankan maka file installer tersebut akan menampilkan jendela instalasi aplikasi postgresql. Jendela pertama akan menampilkan bahasa yang digunakan pada proses instalasi selanjutnya.



Gambar Proses Instalasi PostgreSQL pad Windows (1)

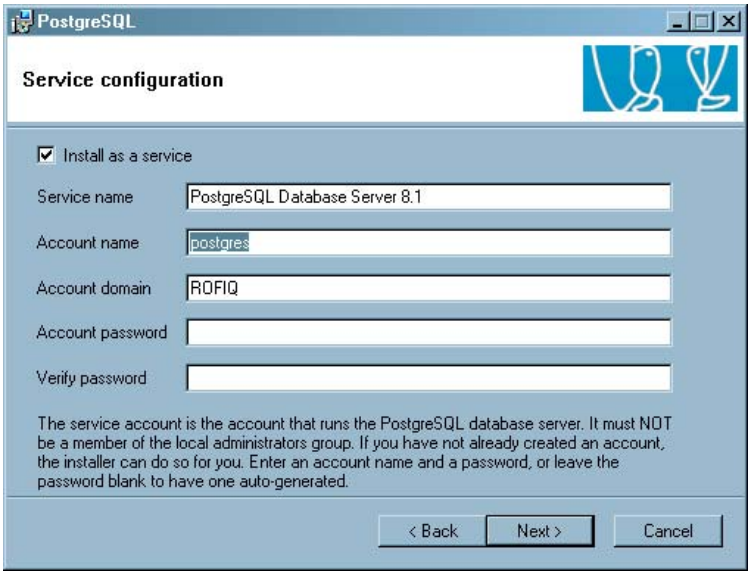
Setelah dipilih kemudian klik Start untuk memulai proses instalasi, pada bagian installation options kita akan diminta untuk paket yang akan diinstall. Secara default installer telah menentukan pilihan paket instalasi sistem baru, yang terdiri dari Server, User Interface dan Database Driver.



Gambar Proses Instalasi PostgreSQL pad WIndows(2)

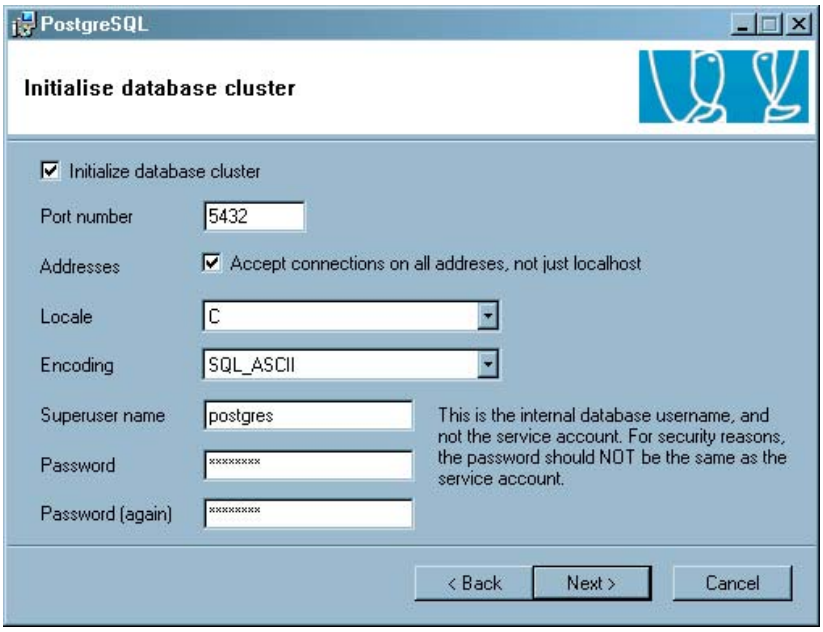
Langkah yang cukup penting dalam proses instalasi postgresql adalah penentuan

konfigurasi service. Pada bagian ini harus ditentukan nama user yang akan menjalankan service postgresql, dan user tersebut bukan merupakan anggota grup Administrator. Apabila nama user belum ada, maka sistem akan secara otomatis membuatkan sesuai dengan nama yang dimasukan



Gambar Proses Instalasi PostgreSQL pad WIndows(3)

Bagian yang sangat penting lainnya dalam proses instalasi adalah, konfigurasi database cluster. Bagian ini penting karena pada bagian inilah akan ditentukan nomor port service postgresql, nama super user sebagai administrator server database serta konfigurasi host yang diinjinkan untuk mengakses server yang di install. Apabila Addsesses di aktifkan, maka server postgresql akan dapat menerima koneksi selain dari localhost.



Gambar Proses Instalasi PostgreSQL pad Windows (4)

KONEKTIFITAS

BAB 2 – Konektifitas

Deskripsi

Bab konekitifitas ini akan membahas tentang tool-tool dari sisi client PostgreSQL yang digunakan untuk melakukan administrasi server database serta database didalamnya. Pembahasan pada bab ini mencakup pgAccess, pdAdmin dan psql yang merupakan aplikasi bawaan PostgreSQL.

Obyektif

Tujuan dari bab ini adalah agar peserta dapat mengenal dan menggunakan berbagai tool administratif database PostgreSQL.

Outline

- Psql
- Pgaccess
- pgAdmin
- phpPgAdmin

PostgreSQLdapat menjalin konektifitas antara program-program client dengan server postgres itu sendiri. Untuk itu, dibutuhkan beberapa informasi tentang, username, password, nama database, nama server, dan nomor port (5432 merupakan default port dari database postgres). PostgreSQLmenggunakan model komunikasi client/server, server postgres terus menerus berjalan dan menunggu permintaan dari client, sehingga nantinya permintaan tersebut akan diproses dan memberikan hasilnya ke client.

Sebagai sebuah server database, maka postgresql juga akan membutuhkan alat bantu yang digunakan untuk melakukan administrasi database termasuk membuat dan memanipulasi data. Beberapa alat bantu yang bisa digunakan antara lain adalah;

Psql

PostgreSQLmenyertakan program *client* yang disebut psql. Untuk membuat dan memanipulasi sebuah *database* dapat dilakukan dari psql. Untuk memebuat database, terlebih dulu harus login sebagai postgres superuser. Pada postgres telah tersedia juga contoh *database* yang diberi nama *template1*, dari *template1* inilah awal di mana kita akan membuat *database* baru. Setiap perintah SQL diakhiri dengan **titik koma (;)**, untuk keluar dari psql ketikkan perintah **\q**,

Untuk memulai database postgres, dari *prompt* user linux ketikkan *psql -u template1* kemudian Enter. setelah itu, kita diminta memasukkan username. Oleh karena kita belum memiliki username, maka diharuskan login sebagai postgres superuser. Jadi, pada username ketikkan *postgres* sedangkan pada passwordnya dibiarkan kosong. Sebagai **catatan** user Linux tidak sama dengan user postgresql.

```
#  su postgres
$                          - promt tersebut menunjukan user postgres bukan root lagi.
```

Dari prompt di atas ketikkan *psql template1* dan Enter, otomatis akan masuk ke dalam database template1. Dari tempat ini kita dapat membuat database dan username.

```
$  psql template1
Welcome to psql 8.1.1, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help with psql commands
       \g or terminate with semicolon to execute query
       \q to quit

mydb=#
```

Hal seperti di atas dapat juga dilakukan dari prompt user linux dengan mengetikkan psql -u template1. Hasilnya seperti berikut ini.

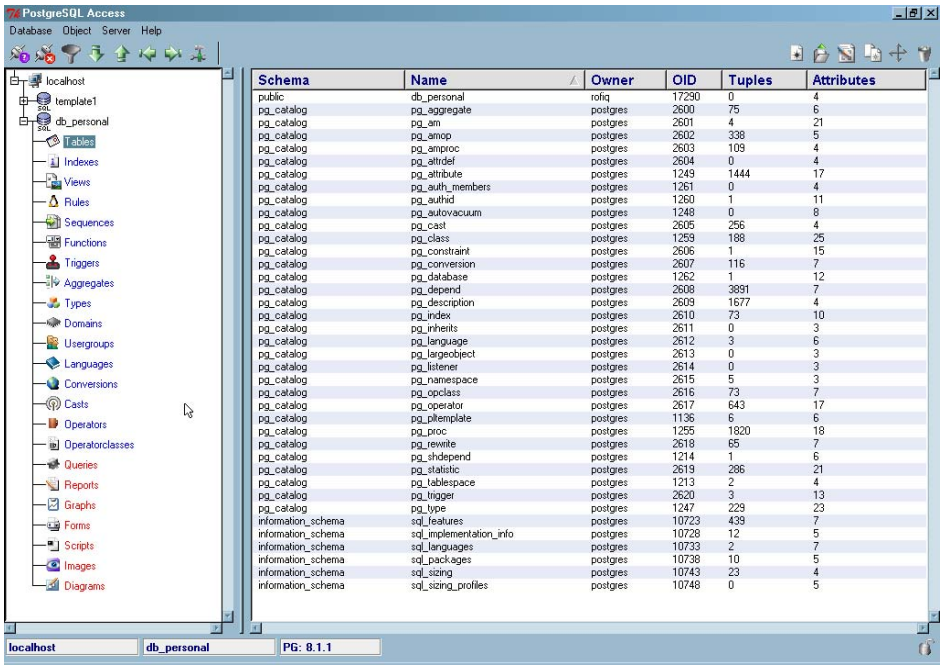
```
$ psql -u template1
psql: Warning: The -u option is deprecated. Use -U.
Username: postgres
Password:
Welcome to psql 8.1.1, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
           \h for help with SQL commands
           \? for help on internal slash commands
           \g or terminate with semicolon to execute query
       \q to quit
template1=#
```

Pada bagian ini kita diminta memasukkan username, sedangkan bagian yang pertama tadi langsung masuk pada prompt databasenya tanpa harus mengisi usernamenya.

Pgaccess

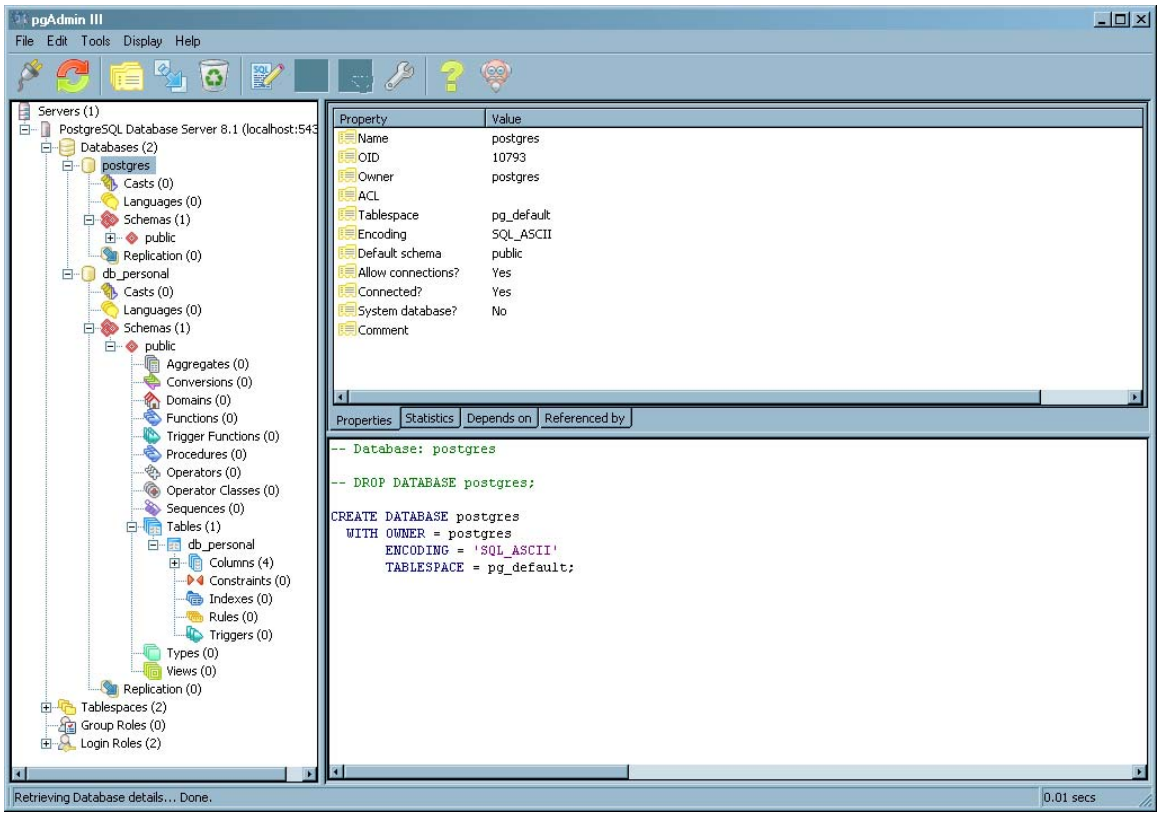
PgAccess merupakan tool database manager berbasis GUI TCL/TK. PgAccess tersedia dalam versi Windows dan Linux. PgAccess dpat di download secara gratis di www.pgaccess.com. Program pgAccess ini tidak besar hanya sekitar 3 MB, meskipun begitu pgAccess memempunyai tool yang cukup lengkap, seperti membuat database, tabel, form, query, atau fungsi.



Gambar pgAccess

pgAdmin

pgAdmin merupakan aplikasi atau interface database postgresql yang dapat digunakan untuk melakukan desain dan manajemen secara komperehensif, selain itu pgAdmin juga tersedia dalam versi Windows dan Linux. pgAdmin menggunakan lisensi Artistic License yang tetap dapat digunakan dan di sebar luaskan secara gratis. Versi terakhir pgAdmin pada saat modul ini ditulis adalah versi III yang dikembangkan dengan bahasa C++ dan menggunakan wxWidgets untuk mendukung cross platform. Koneksi ke postgresql dibuat dengan menggunakan native libpg library. pgAdmin juga dapat dilengkapi dengan pgAgent untuk mengatur penjadwalan proses dan Slony-I Support untuk mendukung proses replikasi master-slave.



Gambar pgAdmin III

Pada halaman utama pgAdmin III, akan ditampilkan struktur database dan detail setiap object yang ada di dalamnya, sehingga hampir semua pengelolaan databse dapat dilakukan dari pgAdmin secara komperehensif. Beberapa tool dalam pgAdmin yang dapat digunakan antara lain adalah;

Control server, digunakan untuk melihat status server database, menjalankan dan menghentikan service server database.

Export Tool, digunakan untuk melakukan eksport data dari Query Tool.

Edit Grid, digunakan untuk menampilkan dan mengubah data dalam tabel yang dipilih.

Maintenance, digunakan untuk melakukan perawatan database, seperti menjalankan task, statistik, clean up data dan melakukan indexing.

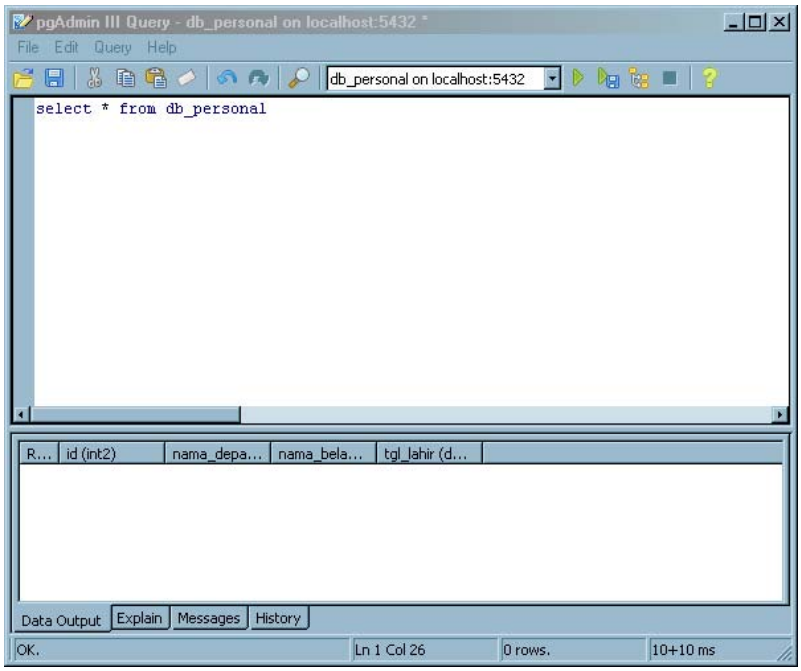
Backup, digunakan untuk melakukan backup database.

Restore, digunakan untuk mengembaliakn hasil dari data backup.

Grant Wizard, digunakan untuk memberikan privileges user atau grup user terhadap obyek tertentu.

Server status, untuk menampilkan informasi status server termasuk jumlah user yang sedang terhubung dan log server.

Options, digunakan untuk mengkonfigurasi pgAdmin.



Gambar pgAccess Query

phpPgAdmin

Kalau untuk melakukan administrasi database MySQL kita mengenal phpMyAdmin, maka postgresql juga tersedia interface web yaitu phpPgAdmin. Agar web server apache yang digunakan bisa mendukung postgresql, maka apache perlu di kompilasi ulang dengan mengaktifkan opsi postgresql.

Opsi yang digunakan adalah --with-pgsql configure

Setelah dilakukan instalasi kemudian phpPgAdmin dapat diakses melalui web browser http://localhost/phpPgAdmin/

PERINTAH DASAR DATABASE

Bab 3 – Perinta Dasar Database

Deskripsi

Bab perintah dasar database ini akan banyak membahas dasar-dasat perintah dalam database, yang meliputi pemahaman membuat database, menghapus database, session user dan query buffer. Bab ini juga membahas penggunaan fasilitas help dalam psql yang dapat dimanfaatkan apabila mengalami kesulitan dalam mengelola database.

Obyektif

Tujuan dari bab ini adalah agar peserta dapat memahami dan mampu untuk menggunakan perintah dasar dalam database PostgreSQL dan memanfaatkan query buffer. Bab ini juga bertujuan agar siswa bisa menggunakan bantuan apabila mengalami kesulitan dalam melakukan operaso-operasi database.

Outline

- Membuat Database
- Menghapus Database
- Membuat User
- Session User
- Query Buffer

Menggunakan Help PostgreSQL

Pada bagian ini akan dipelajari bagaimana terkoneksi ke sebuah server database sehingga kita dapat mengenal dengan benar karakteristik sebuah database, dalam hal ini postgres. Database merupakan kumpulan dari berbagai data yang saling berhubungan. Data tersebut meliputi objek-objek database seperti tabel, view, trigger, fungsi, operator dan berbagai sintak yang lainnya.

Dalam modul ini perintah-perintah database akan diberikan melalui prompt psql dan interface pgAdmin Query. Apabila perintah dilakukan melalui prompt psql, maka akan ditampilkan juga prompt psql seperti berikut;

```
template1=#  CREATE DATABASE  db_personal;
    CREATE DATABASE
template1=#
```

Sedangkan jika menggunakan pgAdmin Query maka prompt tidak akan ditampilkan, sehingga dalam contoh akan langsung ditampilkan query lengkapnya.

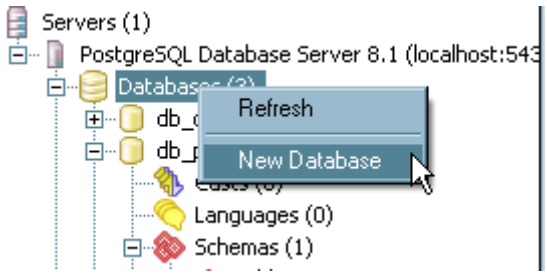
```
CREATE DATABASE  db_personal;
```

Membuat Database

Setelah berhasil masuk dan terkoneksi atau tersambung dengan *database template1*, langkah pertama buatlah sebuah *database* baru agar nantinya dipakai untuk melakukan pekerjaan selanjutnya dan bukan pada *database template1*. Sangat tidak dianjurkan bekerja pada *database template1*, sebab *template1* merupakan database system dan merupakan template bagi seluruh *database* yang baru dibuat. Ikuti perintah di bawah ini untuk membuat database baru.

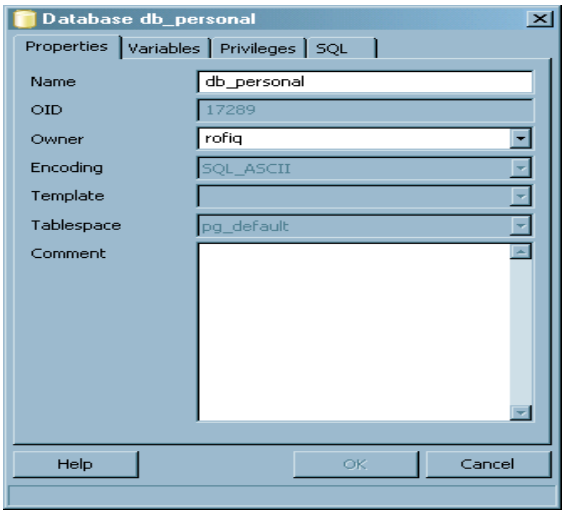
```
template1=#  CREATE DATABASE  db_personal;
CREATE DATABASE
template1=#
```

Untuk membuat database kita juga dalam menggunakan pgAdmin, yaitu dengan cara klik kanan pada Obyek Databases kemudian pilih New Database.



Gambar Membuat Database pada pgAccess III

Pada jendela yang muncul kemudian masukan parameter yang diperlukan seperti nama database, pemilik (owner) serta uer/grup dan privileges.



Gambar Memasukan Atribut Database pada pgAccess III

Kemudian untuk pembahasan selanjutnya kita gunakan database yang baru dibuat tadi.

Menghapus Database

Database yang telah terbentuk dapat dihapus, jika database tersebut telah diisi dengan berbagai macam tabel dan entri datanya. Dalam proses penghapusan sebuah database, tidak perlu menghapus tabel beserta isinya satu demi satu. Namun, kita langsung saja menghapus databasenya, secara otomatis semua tabel beserta data yang terdapat di dalam database tersebut akan ikut terhapus. Untuk menghapus database gunakan perintah DROP DATABASE nama_database. Lihat contoh di bawah ini.

Catatan:

Untuk menghapus database, user anda harus sebagai postgres superuser, dan untuk melihat daftar tabel database yang telah terbentuk ketikkan perintah backslash-l (\l) pada prompt psql. Untuk lebih jelasnya ikuti contoh di bawah ini.

```
$  psql db_personal postgres
Welcome to psql 8.1.1, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

db_personal=#
```

Terlebih dulu lihatlah daftar databasenya untuk memastikan database mana yang akan dihapus. Untuk melihat daftar database bisa digunakan opsi l (list).

```
db_personal=#  \l
        List of databases
 Database      |  Owner      | Encoding
---------------+-------------+-----------
 db_personal             | postgres    | SQL_ASCII
```

```
 coba          | postgres    | SQL_ASCII
 satu             | postgres    | SQL_ASCII
mydb    |postgres  |SQL_ ASCII
mydb1   |postgres  |SQL_ ASCII
 template0      | postgres    | SQL_ASCII
 template1      | postgres    | SQL_ASCII
(7 rows)
```

Setelah itu, hapuslah database yang diinginkan, pada contoh ini kita akan menghapus database mydb1.

```
db_personal=#  DROP DATABASE mydb1;
DROP DATABASE
```

untuk memastikan apakah database tersebut berhasil dihapus atau tidak, lihatlah daftar tabel databasenya.

```
db_personal=#  \l
        List of databases
 Database    |  Owner       | Encoding
 -------------+-------------+----------
 db_personal          | postgres     | SQL_ASCII
 latihan         | postgres    | SQL_ASCII
 mydb   |postgres  |SQL_ ASCII
 satu           | postgres     | SQL_ASCII
 template0   | postgres     | SQL_ASCII
 template1   | postgres     | SQL_ASCII
(6 rows)
```

Membuat User

Dalam PostgreSQL secara default telah terdapat user postgres yang juga sekaliigus menjadi administrator dan mempunyai level tertingggi dalam PostgreSQL. Dalam operasional database biasanya user potgres tidak digunakan dengan alasan keamanan. Membuat user pertama kali harus dilakukan oleh user postgres.

Dari database *template1* buatlah sebuah *user* baru, tapi sebelumnya kita lihat dulu *help* dari *create user* dengan mengetikkan **\h create user** pada prompt template1.

```
template1=#  \h create user
Command:     CREATE USER
Description: Creates a new database user
Syntax:
CREATE USER username
  [ WITH
  [ SYSID uid ]
  [ PASSWORD 'password' ] ]
  [ CREATEDB   | NOCREATEDB ] [ CREATEUSER |                 NOCREATEUSER ]
  [ IN GROUP     groupname [, ...] ]
  [ VALID UNTIL  'abstime' ]
```

Kemudian, dilanjutkan dengan membuat user baru:

```
template1=#  CREATE USER  rofiq createdb;
CREATE USER
template1=#
```

Agar user tersebut memiliki izin untuk membuat sebuah database baru, pada waktu pembuatan *user* kita perlu menambahkan opsi **createdb** di belakang nama usernya.

Selanjutnya, kita akan berpindah dari database template1 ke database coba. Loginlah dengan *username* yang baru dibuat tadi yaitu rofiq.

```
template1=#  \c db_personal rofiq
You are now connected to database db_personal as user rofiq.
```

Session User

Pengontrolan session digunakan untuk mengetahui keberadaan posisi *user* dan waktunya pada postgres. Gunakan queri *select current_user;* pada *prompt database* untuk melihat posisi *database* pada *user* siapa. Berikut contohnya.

```
$ psql -u db_personal
psql: Warning: The -u option is deprecated. Use -U.
Username: rofiq
Password: <ENTER>
Welcome to psql 8.1.1, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit
db_personal=>
```

Kemudian, ketikkan perintah dibawah ini untuk mengetahui posisi database terletak pada user apa.

```
db_personal=> select  current_user;
 current_user
--------------
    rofiq
  (1 row)
```

Dari output di atas terlihat dengan jelas bahwa kita berada pada user *rofiq.*

Berikut ini beberapa cara pengontrolan, misalkan jika ingin mengetahui waktu beserta tanggal pada hari ini, gunakan perintah *select current_timestamp;* tanggal beserta jam akan ditampilkan secara bersamaan.

```
db_personal=> select current_timestamp;
       timestamp
-------------------------------
2005-12-16 14:07:16.099+07
(1 row)
```

Berikut query untuk menampilkan tanggal.

```
db_personal=> select current_date;
       date
------------
2005-12-16
(1 row)
```

Query untuk menampilkan waktu.

```
db_personal=> select current_time;
   time
----------
 14:07:18
(1 row)
```

Query Buffer

Menulis atau mengetik pada *query buffer* mirip dengan menulis perintah pada sebuah prompt di dalam sebuah sistem operasi. Dalam psql, titik koma (;) dipakai untuk mengakhiri atau mengeksekusi sebuah perintah atau cukup dengan menuliskan *backslash*-g (\g). Berikut sebuah contoh *multiline query*, Misalkan, kita akan mencoba perintah *SELECT 4+6;* contoh ini dapat ditulis dalam satu baris, query tersebut dapat juga pilah-pilah menjadi beberapa baris. Berikut perintahnya.

```
db_personal=> SELECT
db_personal-> 2+3
db_personal-> ;
 ?column?
----------
         5
(1 row)
```

baris pertama (**=>)** adalah prompt penggantian atau pemilihan, baris kedua (**->)** menunjukkan bahwa proses masih berlanjut sedangkan baris ketiga menandakan akhir dari proses tersebut sehingga setelah dieksekusi proses itu akan dibawah oleh *psql* ke *server database*, kemudian memberikan output dari proses tadi.

Kita juga dapat mencoba beberapa *query* yang melibatkan proses aritmatik yang diawali dengan kata *SELECT* dan diakhiri dengan *titik koma* (;) atau diakhiri dengan *backslash*-g (\g). Contohnya *SELECT 5*9+5;* hasilnya adalah 50, Untuk penjumlahan simbolnya (+), pengurangan (-), pembagian (/) dan untuk perkalian (*).

```
db_personal=> SELECT
db_personal-> 5*9+5
db_personal-> \p
SELECT
5*9+5
db_personal-> \g
 ?column?
----------
      50
(1 row)
```

Untuk menampilkan query buffer, ketikkan perintah *backslash*-p (\p) dan untuk menghapus *query buffer* gunakan perintah *backslash*-r (\r). Perintah \p dapat menangani *query* yang sangat panjang atau banyak. Maksudnya, dalam penulisan *query* yang panjang, kita dapat melihat query yang ditulis tadi. Jika ada kesalahan, kita dapat mengubahnya kemudian mengeksekusinya. Berikut contohnya.

```
db_personal=> SELECT
db_personal-> 2+2*2*4
db_personal-> *2+4-1+10/
db_personal-> 2*2
db_personal-> +
db_personal-> 2
db_personal-> *
db_personal-> 5
db_personal-> -
db_personal-> 5
db_personal-> *2+4-1
db_personal-> \p
SELECT
2+2*2*4
```

```
*2+4-1+10/
2*2
+
2
*
5
-
5
*2+4-1
db_personal-> ;
 ?column?
----------
      50
(1 row)
```

Menggunakan Help PostgreSQL

Pada database postgres tersedia *help* untuk berbagai sintak atau perintah dalam menjalankan *database* tersebut. Untuk itu, ketika login dan kemudian masuk pada prompt psql, akan tertera beberapa sintaks perintah yang dapat digunakan sebagai acuan dalam penggunaan database postgres.

```
$ psql db_personal rofiq
Welcome to psql 8.1.1, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit
db_personal=>
```

Catatan: db_personal merupakan nama databasenya dan **rofiq** adalah nama usernamenya.

Untuk memperoleh help dari perintah SQL pada postgres cukup ketikkan perintah **\h** lalu tekan Enter, akan terlihat perintah-perintah yang terdapat dalam SQL. Namun, jika kita ingin lebih spesifik lagi, misalnya ingin melihat sintaks dari CREATE DATABASE, gunakan perintah ini *\h CREATE DATABASE*.

```
db_personal=> \h CREATE DATABASE
Command:     CREATE DATABASE
Description: Creates a new database
Syntax:
CREATE DATABASE name
       [ WITH [ LOCATION = 'dbpath' ]
         [ TEMPLATE = template ]
  [ ENCODING = encoding ] ]
```

MANAJEMEN TABEL

Bab 4 – Manajemen Tabel

Deskripsi

Bab Manajemen Tabel ini akan banyak membahas tentang pengelolaan tabel. Pembahasan meliputi membuat dan menghapus tabel serta manipulasi tabel, yang juga dibahas pada bab ini adalah privileges tabel serta inheritance.

Obyektif

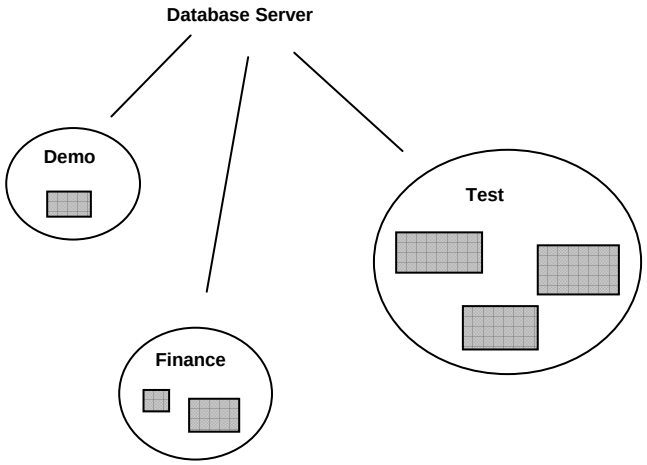
Tabel yang merupakan dasar dalam pembentukan database harus bisa dipahami secara mendalam oleh peserta, dengan memahami bab ini maka peserta diharapkan mampu melakukan desain, membuat dan memanipulasi tabel-tabel disesuaikan dengan kebutuhan.

Outline

- Database Relasional
- Tabel
- Membuat Tabel
- Manipulasi Tabel
- Menghapus Tabel
- Nilai Default
- Temporary Tabel
- GRANT dan REVOKE
- INHERITANCE

Database Relasional

Database berfungsi sebagai tempat penyimpanan data, dapat juga digunakan untuk menyimpan aliran data tempat kita dapat mengakses atau mendapatkan kembali data tersebut. Saat ini sebagian besar database system merupakan *relational database*. Dalam prakteknya maksud dari semua data yang tersimpan dalam database diatur dalam sebuah struktur yang sama. Dalam database dapat dibuat begitu banyak tabel, semua tabel tersebut merupakan dasar dari *Relational Database management System* (RDBMS). Tabel tersebut menampung data yang tersimpan ke dalam database, Gambar di bawah ini menunjukkan sebuah database server dengan tiga jalan pengaksesan, yaitu *demo, finance,* dan *test*.



Gambar Contoh Database

Tabel

Sebuah tabel dalam database relasional bisa dikatakan seperti sebuah tabel dalam file excel atau kertas kerja, dimana tabel terdiri dari baris (row) dam kolom (column). Jumlah dan nama kolom dalam database adalah statis (bukan variable), artinya jumlah dan namanya harus didefinisikan terlebih dulu di awal. Sedangkan baris merupakan sebuah variable yang dapat dihapus dan diisi kapanpun, sehingga jumlah nya akan selalu berubah sesuai dengan jumlahj data di dalamnya. Pada saat tabel dibaca SQL tidak menjamin urutan data dalam sebuah tabel ascending atau descending, urutan data hanya bisa dijamin dengan memberikan parameter secara eksplisit pada saat melakukan query.

Setiap kolom dalam tabel mempunyai tipe data, tipe data digunakan untuk membatasi jenis data yang bisa dimasukan, sehingga akan mempermudah dalam menggunakannya dan melakukan pengelolaan selanjutnya. Misalnya kolom yang mempunyai tipe data numerik tidak akan bisa dimasukan data string di dalamnya.

Sangat disarankan pada saat akan membuat tabel, sebaiknya membuat sebuah konvensi khusus dalam memberikan nama kolom dan tipe datanya.

Membuat Tabel

Tabel merupakan bagian dari *database* yang berfungsi sebagai tempat atau wadah berbagai macam data disimpan, setiap tabel terdiri atas *field* (kolom) dan *record* (baris). Dalam pembuatan sebuah tabel mempunyai beberapa ketentuan, antara lain.

- Harus memiliki Primary Key, artinya dalam pembuatan tabel haruslah terdapat sekelompok *field* yang menyebabkan setiap *record* dalam tabel tersebut tidak sama.
- Pada pendeklarasian *primary key* tidak boleh *null* (kosong), jadi kita harus mendeklarasikan sebagai *not null*. Namun, secara default PostgreSQL menganggapnya sebagai *nullable* (boleh kosong). Jika pada waktu deklarasi, kita tidak menyebutkan NULL atau NOT NULL.

Sekarang kita coba membuat sebuah tabel yang diberi nama **tbl_personal**. Tabel ini memiliki enam record, yaitu int_id, txt_NamaDepan, txt_NamaAkhir, dt_TglLahir, bol_IsNikah, txt_Pekerjaan. Setiap kolom berisikan tipe yang sama dan akan ditampilkan pada baris yang berbeda. Berikut adalah tipe dari struktur yang membuat database relasional.

Query yang digunakan adalah :

```
CREATE TABLE tbl_personal
(
  int_id            int4 NOT NULL,
  "txt_NamaDepan"   varchar(10),
  "txt_NamaAkhir"   varchar(10),
  "dt_TglLahir"     date,
  "bol_IsNikah"     bool,
  "txt_Pekerjaan"   varchar(20)
)
```

Untuk melihat kembali struktur tabel yang telah dibuat tadi, ketikkan perintah “\d” seperti di bawah ini dan lihat hasil outputnya.

```
db_personal=# \d tbl_personal
            Table "public.tbl_personal"
    Column     |          Type          | Modifiers
---------------+------------------------+-----------
 int_id        | integer                | not null
 txt_NamaDepan | character varying(10) |
 txt_NamaAkhir | character varying(10) |
 dt_TglLahir   | date                   |
 bol_IsNikah   | boolean                |
 txt_Pekerjaan | character varying(20) |
```

Manipulasi Tabel

Dalam membuat tabel sebaiknya direncanakan field-fieldnya dan tipe datanya untuk menghindari terjadinya perubahan setalah tabel berisi data. Akan tetapi meskipun begitu bukan berarti tabel tidak bisa diubah dan harus dihapus apabila akan di ubah. PostgreSQL dan standar database lainya menyediakan utilitas ALTER TABLE untuk mengubah struktur tabel yang sudah dibuat sebelumnya. Selain untuk melakukan perubahan pada tabel ALTER dengan opsi yang berbeda juga digunakan untuk mengubah view atau fungsi yang

sudah dibuat. Perlu diingat bahwa alter bukan untuk melakukan perubahan data pada tabel melainkan strukturnya.
Sebagai contoh kita akan buat sebuah tabel tbl_employee dengan struktur yang mirip dengan tabel personal, tabel ini akan diubah beberapa field di dalamnya.

```
db_personal=> CREATE TABLE tbl_employee
db_personal-> (
db_personal(>   int_id int4 NOT NULL,
db_personal(>   "txtNamaDepan" varchar(10),
db_personal(>   "txtNamaAkhir" varchar(10),
db_personal(>   "dtTglLahir" date,
db_personal(>   "bolIsNikah" bool,
db_personal(>   "txtPekerjaan" varchar(20)
db_personal(> ) ;
CREATE TABLE
```

Sebelum di hapus fiedlnya akan dilihat dulu strukturnya agar bisa di bandingkan dengan setelah dihapus. Untuk melihat struktur bisa digunakan **\d nama_tabel** pada prompt psql.

```
db_personal=> \d tbl_employee;
 int_id       | integer               | not null
 txtNamaDepan | character varying(10) |
 txtNamaAkhir | character varying(10) |
 dtTglLahir   | date                  |
 bolIsNikah   | boolean               |
 txtPekerjaan | character varying(20) |
```

Terlihat bahwa strukturnya sama dengan yang dibuat, selanjutnya akan dihapus satu field yaitu txtPekerjaan. Perintah untuk menghapus field adalah :

ALTER TABLE “nama_tabel” DROPM COLUMN “nama_field/column”;

```
db_personal=> ALTER TABLE tbl_employee DROP COLUMN "txtPekerjaan" ;
```

Selanjunya kita lihat kembali struktur tabel tbl_employee,

```
db_personal=> \d tbl_employee;
 int_id       | integer               | not null
 txtNamaDepan | character varying(10) |
 txtNamaAkhir | character varying(10) |
 dtTglLahir   | date                  |
 bolIsNikah   | boolean               |
 txtPekerjaan | character varying(20) |
```

Terlihat bahwa field txtPekerjaan sudah tidak ada dalam tabel tbl_employee. Kemudian akan kita tambahkan lagi field/kolom txtPekerjaan pada tabel tersebut, perintah yang digunakan untuk menambahkan kolom adalah;

ALTER TABLE “nama_table” ADD COLUMN “nama_field” “tipe_data”

```
db_personal=> ALTER TABLE tbl_employee ADD COLUMN "txtPekerjaan" varchar(20);
ALTER TABLE
```

Kemudian bisa dilihat lagi struktur terakhir tabel tbl_employee.

```
db_personal=> \d tbl_employee;
 int_id       | integer               | not null
 txtNamaDepan | character varying(10) |
 txtNamaAkhir | character varying(10) |
```

```
 dtTglLahir   | date                  |
 bolIsNikah   | boolean               |
 txtPekerjaan | character varying(20) |
```

Tampak pada contoh bahwa struktur tabel telah berubah menjadi seperti semula.

Contoh mengubah nama tabel tbl_employee menjadi tbl_pegawai

```
db_personal=> ALTER TABLE tbl_employee RENAME TO tbl_pegawai;
ALTER
```

Contoh mengganti kolom atribut *nama_depan* menjadi *nama*:

```
db_personal=> ALTER TABLE tbl_employee
db_personal-> RENAME COLUMN “txt_Pekerjaan”
db_personal-> TO Profesi;
ALTER
```

Pada db_personal lihat lagi atribut dari tabel teman, setelah melakukan perintah di atas secara otomatis id defaultnya akan terhapus:

```
db_personal=> \d teman
            Table "teman"
 Attribute        |     Type                    | Modifier
----------------+---------------+----------
 nama             | character(20)             |
 marga            | character(25)             |
 pekerjaan        | character(25)             |
 kota             | character(30)             |
 state            | character(3)              |
 umur             | integer                   |
 id                      | integer                   |
```

Contoh berikut adalah menambahkan primary key pada tabel tbl_employee

```
db_personal=> ALTER TABLE tbl_empployee ADD PRIMARY KEY (int_ID);
NOTICE:  ALTER TABLE / ADD PRIMARY KEY will create implicit index "tbl_empployee _
pkey" for table "tbl_empployee"
ALTER TABLE
```

Menghapus Tabel

Menghapus sebuah tabel berarti juga akan menghapus semua data di dalamnya. Untuk itu diperlukan kehati-hatian dalam melakukan penghapusan sebuah tabel. Sebelum sebuah tabel dihapus, daftar seluruh tabel yang terdapat dalam database yang sedang aktif sebaiknya ditampilkan terlebih dulu. Dengan melihat seluruh tabel maka akan membantu meyakinkan bahwa tabel yang akan dihapus namanya benar. Untuk melihat daftar tabel bisa digunakan perintah **\z** pada promp psql.

```
db_personal=> \z
 public | pga_diagrams | table |
 public | pga_forms    | table |
 public | pga_graphs   | table |
 public | pga_images   | table |
 public | pga_layout   | table |
 public | pga_queries  | table |
 public | pga_reports  | table |
 public | pga_scripts  | table |
 public | tbl_account  | table |
```

```
 public | tbl_employee | table |
 public | tbl_hitung   | table |
 public | tbl_personal | table |
```

Tampak pada daftar tabel diatas beberapa tabel bawaan PostgreSQL dan buatan user. Untuk menghapus tabel digunakan perintah DROP TABLE yang diikuti nama tabel yang akan dihapus. Misalnya akan menghapus tabel tbl_employee maka digunakan perintah seperti pada contoh berikut;

```
db_personal=> DROP TABLE tbl_employee;
DROP TABLE
```

Setelah itu kemudian bisa dilihat lagi apakah tabel tersebut masih ada dengan menggunakan \z.

```
db_personal=> \z
 public | pga_diagrams | table |
 public | pga_forms    | table |
 public | pga_graphs   | table |
 public | pga_images   | table |
 public | pga_layout   | table |
 public | pga_queries  | table |
 public | pga_reports  | table |
 public | pga_scripts  | table |
 public | sahabat      | table |
 public | tbl_account  | table |
 public | tbl_hitung   | table |
 public | tbl_personal | table |
```

Nilai Default

Salah satu fasilitas yang diberikan PostgreSQL yaitu mengontrol dan menggunakan *DEFAULT.* Ketika membuat sebuah tabel, nilai *DEFAULT* dapat digunakan pada setiap tipe kolom. Nilai Default tersebut akan digunakan kapan saja jika nilai kolom tidak dimasukkan pada saat melakukan INSERT data. Misalnya kita dapat menggunakan default *timestamp* untuk kolom waktu pembuatan, jadi pada kolom tersebut tidak perlu diisi karena secara otomatis akan terisikan oleh fungsi timestamp. Berikut contohnya.

```
CREATE TABLE tbl_account (
nama  CHAR(25),
neraca NUMERIC(16,2) DEFAULT 0,
aktive CHAR(1) DEFAULT 'Y',
Waktu_pembuatan  TIMESTAMP DEFAULT CURRENT_TIMESTAMP );
```

setelah itu masukkan data pada atribut *nama*, sedangkan untuk attribut neraca, aktif dan waktu pembuatan biarkan kosong sebab otomatis akan terisi oleh nilai defaultnya:

```
INSERT INTO tbl_account (nama) VALUES ('Pembangunan Daerah');
```

kemudian lihat isi dari tabel *account*:

```
db_personal=> select * from tbl_account;
           nama            | neraca | aktive |     waktu_pembuatan
---------------------------+--------+--------+-------------------------
 Pembangunan Daerah        |   0.00 | Y      | 2005-12-16 17:29:23.448
(1 row)
```

Jika diperhatikan bahwa field selain nama secara otomatis akan terisi sesuai dengan isian

defaultnya.

Temporary Tabel

Temporary tabel sifatnya hanya sementara atau berumur pendek artinya akan aktif hanya ketika kita sedang berada atau login ke *database* namun ketika kita *logout* atau keluar dari *psql database* maka secara otomatis *temporary* tabel akan terhapus. Pada contoh ini kita akan mengilustrasikan tentang konsep ini. Pertama membuat sebuah *temporary* tabel setelah itu keluar dari *psql*, kemudian login lagi ke *psql* ketika anda login lagi maka secara otomatis *temporary* tabel tersebut sudah tidak aktif lagi.

```
$ psql db_personal shinta
Welcome to psql 8.1.1, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit
db_personal=>

db_personal=> CREATE TEMPORARY TABLE cobatemp(col INTEGER);
CREATE

db_personal=> SELECT * FROM cobatemp;
 col
------------
(0 rows)

db_personal=> \q

$ psql db_personal shinta
Welcome to psql 8.1.1, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit
db_personal=>
db_personal=> SELECT * FROM cobatemp;
ERROR:  Relation 'cobatemp' does not exist
```

Temporary tabel hanya akan tampak jika setelah login ke dalam *database* dan kita langsung membuatnya pada *session psql* (pada *database* tersebut). Perlu diingat bahwa *temporary* tabel ini hanya akan tampak pada tempat di mana kita membuatnya, artinya jika kita membuatnya pada *database db_personal* dengan ***owner*-nya *shinta*** maka hanya pada *user* inilah dia akan tampak dan tidak akan tampak pada *user* lain. Beberapa *user* pada PostgreSQLdapat membuat *temporary* tabel dengan nama yang sama, dan setiap *user* hanya bisa melihat tabelnya masing-masing dari tabel.

User 1	User 2
CREATE TEMPORARY TABLE *cobatemp*(*col*	CREATE TEMPORARY TABLE *cobatemp*(*col*

User 1	User 2
INTEGER)	INTEGER)
INSERT INTO *cobatemp* VALUES (1)	INSERT INTO *cobatemp* VALUES (2)
SELECT *col* FROM *cobatemp* returns 1	SELECT *col* FROM *cobatemp* returns 2

GRANT dan REVOKE

Ketika kita membuat sebuah tabel maka secara default hanya user dimana tempat kita membuat tabel tersebut dan *user* postgres yang dapat mengaksesnya. Artinya jika kita membuat tabel pada *user* ***shinta*** maka hanya *user* tersebutlah yang dapat mengakses tabel itu. Namun jika kita ingin agar tabel yang telah dibuat pada *user* kita dapat diakses oleh user tertentu maupun semua user yang berada pada PostgreSQL, maka semua itu dapat dilakukan dengan perintah GRANT. Jika ingin mencabut lagi akses *user* lain terhadap tabel tersebut gunakan perintah REVOKE. Dengan perintah GRANT kita dapat memberi izin SELECT, UPDATE, INSERT , DELETE, RULE dan ALL pada *user* lain untuk mengakses database kita.

Gunakanlah perintah backslash-z (\z) pada *psql* untuk mengetahui hak akses *(privileges)* pada objek atau tabel yang telah ada. Misalkan kita ingin melihat informasi *privileges* pada **tabel tbl_personal** maka gunakan perintah \z seperti di bawah ini:

```
db_personal=# \z  tbl_personal
         Access privileges for database "db_personal"
 Schema |     Name     | Type  |       Access privileges
--------+--------------+-------+-------------------------------
 public | tbl_personal | table | {rofiq=arwdRxt/rofiq,=r/rofiq}
(1 row)
```

Keterangan :

```
              =xxxx -- privileges diberikan ke PUBLIC
          uname=xxxx -- privileges diberikan ke user
   group gname=xxxx -- privileges diberikan ke group

                 r -- SELECT ("read")
                 w -- UPDATE ("write")
                 a -- INSERT ("append")
                 d -- DELETE
                 R -- RULE
                 x -- REFERENCES
                 t -- TRIGGER
                 X -- EXECUTE
                 U -- USAGE
                 C -- CREATE
                 T -- TEMPORARY
            arwdRxt -- ALL PRIVILEGES (untuk tabel)
                 * -- opsi grant untuk preceding privilege (menggantikan string)

              /yyyy -- user yang memberikan privileges
```

Berikut sebuah contoh lain penggunaan GRANT dan REVOKE.

Misalkan rofiq membuat sebuah tabel tbl_hitung, kemudian akan menghapus hak akses shinta untuk mengakses tabel tbl_hitung. Dalam hal ini harus dilakukan proses REVOKE (penghapusan privileges), dan proses penghapusan privileges ini hanya bisa dilakukan apabila user mempunyai hak akses GRANT/REVOKE atau superuser seperti user postgres.

Berikut adalah contoh perintah untuk menghapus hak akses tbl_hitung terhadap user shinta.

```
db_personal=# REVOKE ALL ON TABLE tbl_hitung from shinta;
REVOKE
```

Kemudian kita lihat lagi atribut tabel tbl_hitung.

```
db_personal=# \z tbl_hitung
                Access privileges for database "db_personal"
 Schema |        Name        | Type  |          Access privileges
 public | tbl_hitung         | table | {postgres=arwdRxt/postgres}
 (13 rows)
```

Tampak pada contoh diatas bahwa semua hak akses dimiliki oleh user postgres. Dan untuk membuktikannya maka terlebih dulu login dengan user shinta ke database db_personal dan kemudian mencoba mengakses tabel tersebut.

```
$psql db_personal shinta
Password for user shinta:
Welcome to psql 8.1.1, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help with psql commands
       \g or terminate with semicolon to execute query
       \q to quit

db_personal=> select * from tbl_hitung;
ERROR:  permission denied for relation tbl_hitung
```

Terlihat bahwa akses user shinta terhadap tabel tbl_hitung di tolak.

Untuk mengembalikan hak akses user shinta maka digunakan GRANT. Contoh berikut akan menunjukan penggunaan GRANT pada tabel yang sama.

```
db_personal=# GRANT ALL ON TABLE tbl_hitung TO shinta;
GRANT
```

Setelah di berikan GRANT maka di cek lagi atribut tabel tbl_hitung dan tampak bahwa user shinta telah mempunyai hak akses terhadap tabel tersebut. Untuk melakukan pengecekan lebih lanjut bisa digunakan query pada tabel tersebut dari user shinta.

```
db_personal=# \z tbl_hitung
                  Access privileges for database "db_personal"
 Schema |    Name    | Type  |                  Access privileges

--------+------------+-------+-----------------------------------------------------
---
 public | tbl_hitung | table | {postgres=arwdRxt/postgres,shinta=arwdRxt/postgre
s}
(1 row)
```

GRANT dan REVOKE bisa digunakan terhadap atribut tertentu seperti SELECT, DELETE, UPDATE dan lain-lain. Perintah yang digunakan juga sama, hanya saja hak akses yang tidak lagi ALL tetapi disesuaikan dengan privileges yang akan diberikan. Berikut adalah contoh-contoh pemberikan hak aksesnya,

Menghapus hak akses UPDATE

```
db_personal=# REVOKE UPDATE ON TABLE tbl_hitung from shinta;
REVOKE
```

Menghapus hak akses DELETE

```
db_personal=# REVOKE DELETE ON TABLE tbl_hitung from shinta;
REVOKE
```

Menghapus hak akses SELECT

```
db_personal=# REVOKE SELECT ON TABLE tbl_hitung from shinta;
```

Mememberi hak akses SELECT

```
REVOKE
db_personal=# GRANT SELECT ON TABLE tbl_hitung TO shinta;
GRANT
```

INHERITANCE

INHERITANCE diperguanakan jika kita ingin membuat sebuah tabel baru yang berhubungan dengan tabel yang ada, dengan kata lain *turunan* dari tabel pertama. Misalkan kita membuat sebuah tabel baru bernama *kota* yang memiliki beberapa atribut di antaranya *nama, populasi, dan jarak*. Kemudian buat lagi sebuah tabel baru dengan nama *kabupaten* yang memilki hanya satu atribut, padahal kita menginginkan atributnya sama dengan tabel kota cuman ditambah *state* namun itu tidak menjadi masalah bagi *database* PostgreSQL. Karena hubungan atribut yang kita butuhkan terkait dengan tabel *kota* maka kita cukup menggunakan perintah INHERITANCE terhadap tabel *kota* pada pembuatan tabel kedua kita, berikut contohnya:

```
 db_personal=> CREATE TABLE kota (
 db_personal(>    nama             text,
 db_personal(>    population      float,
 db_personal(>    jarak              int
 db_personal(> );
CREATE
```

Menampilkan struktur atribut tabelnya

```
 db_personal=> \d kota
              Table "kota"
   Attribute       |  Type        | Modifier
--------------------+--------------------- ---+------------
   nama     |  text        |
   populasi |  double precision    |
   jarak    |  integer             |
```

```
 db_personal=> SELECT * FROM kota;
 nama | populasi  |  jarak
--------+------------+--------
(0 rows)
```

Buat sebuah tabel lagi yang hanya memiliki satu attribut

```
 db_personal=> CREATE TABLE kabupaten (
 db_personal(>     state                       char(2))
 db_personal->    INHERITS (kota);
CREATE


 db_personal=> \d kabupaten
              Table "kabupaten"
   Attribute       |  Type        | Modifier
---------------------+--------------------- ----+----------
   nama     |  text        |
   populasi|  double precision    |
   jarak    |  integer             |
   state    | character(2) |

 db_personal=> SELECT * FROM kabupaten;
 nama | populasi | jarak | state
--------+-----------+------+--------
(0 rows)
```

Masukkan beberapa data pada tabel *kota*

```
 db_personal=> INSERT INTO kota VALUES (
 db_personal(> 'Ambon', '219002', '28987');
INSERT 44387 1

 db_personal=> INSERT INTO kota VALUES (
 db_personal(> 'Yogyakarta', '20876', '8987');
INSERT 44388 1

 db_personal=> -- tampilkan tabel tersebut

 db_personal=> SELECT * FROM kota;
    nama         | populasi       | jarak
-----------------+----------------+-----------
 Ambon          |   219002      | 28987
 Yogyakarta    |   20876        | 8987
(2 rows)
```

Masukkan sebuah data pada tabel kabupaten

```
 db_personal=> INSERT INTO kabupaten VALUES (
 db_personal(> 'Nangru Aceh', '123', '8900', 'NA');
INSERT 44419 1
```

Semudian SELECT tabel kabupaten

```
 db_personal=> SELECT * FROM kabupaten*;
    nama          | populasi   | jarak   | state
------------------+------------+--------+---------
 Nangru Aceh   |  123        |  8900  | NA
(1 row)
```

sekarang coba lihat lagi tampilan dari tabel *kota* dengan menambahkan tanda bintang (*) di belakang *query* berikut ini (di belakang nama kota):

```
 db_personal=> SELECT * FROM kota* ;
    nama          | populasi       |  jarak
------------------+----------------+--------
 Ambon            |   219002       |  28987
 Yogyakarta      |   20876        |  8987
Nangru Aceh     |   123          |  8900
(3 rows)
```

Isi dari tabel kota bukan dua *row* lagi melainkan tiga *row*, ini menandakan bahwa tabel *row* mempunya turunan tabel lain yaitu tabel kabupaten. Dalam hal ini tabel *kota* adalah induk tabel yang mempunyai satu keturunan yaitu tabel kabupaten, sehingga jika kita menampilkan tabel induknya yaitu tabel *kota* maka secara otomatis semua turunannya akan ikut ditampilkan namun yang ditampilkan sesuai dengan atribut yang terdapat pada tabel *kota* sedangkan atribut yang tidak terdapat dalam tabel *kota* tidak ditampilkan.

```
 db_personal=> CREATE TABLE desa (wisata  text)
 db_personal-> INHERITS
 db_personal-> (kabupaten);
CREATE
 db_personal=> INSERT INTO desa VALUES  (
 db_personal=>                    ' Nusaniwe',
 db_personal=>                    ' 120 ',
 db_personal=>                    ' 12 ',
 db_personal=>                    ' NS ',
 db_personal=>                    ' Namalatu ',
 db_personal=> );
INSERT 44420 1

 db_personal=> SELECT * FROM desa;
   nama       | populasi   |  jarak | state  |  wisata
------- -+------ -+-----+--- -+---------
 Nusaniwe  |    120        |   12    |  NS    | Namalatu
(1 row)
```

Sekarang tampilkan tabel kota dengan semua turunannya

```
 db_personal=> SELECT * FROM kota* ;
    nama          | populasi       |  jarak
--------- -+------ ---+--------
 Ambon            |   219002      |  28987
 Yogyakarta    |   20876        |  8987
Nangru Aceh  |   123            |  8900
 Nusaniwe       |  120             |   12
(4 rows)
```

MANAJEMEN TABEL (Lanjut)

Bab 5 – Manajemen Tabel (lanjut)

Deskripsi

Bab ini merupakan kelanjutan dari bab sebelumnya yang akan membahas lebih mendalam tentang pengelolaan tabel.

Obyektif

Pada bab ini diharapkan peserta akan dapat memahami dan melakukan pengelolaan tabel secara lebih mendalam.

Outline

- Query Multi Tabel
- Alias Tabel
- Object Identification Numbers (OID)
- Serial dan Big Serial
- BLOBS

Pada bagian ini akan membahas bagaimana menyimpan data menggunakan *multiple* atau melakukan penyimpanan banyak tabel (*multitable*) dan *query* keduanya merupakan dasar dari *relational database*. Untuk pembahasan pada bab ini akan dimulai dari perintah pengujian tabel dan kolom yang mana sangat penting dalam *query* multi-tabel.

Hubungan AntarTabel (Reference)

Tabel merupakan bagian dari *database* yang dapat menyimpan berbagai macam data, tabel memiliki *field* atau kolom dan *record* atau baris. Untuk menghubungkan tabel ada beberapa syarat yang harus terpenuhi seperti, antara tabel yang mau dihubungkan harus memiliki sebuah *field* dengantipe data yang sama. Jadi setidaknya salah satu tabel harus memiliki *field* yang merupakan kunci (*primary key*) dari tabel yang lain, misalkan kita membuat dua buah tabel yang mana tabel pertama diberi nama *barang* berisikan dua buah atribut yaitu *id_barang* dan *nama_barang, id_barang* sebagai *primary key*. Tabel kedua berisikan tiga buah atribut yaitu *id_barang* (id-nya harus sesuai dengan id pada tabel pertama), *jenis_barang* dan *kegunaanya*. *id_barang* dan *kegunaanya* sebagai *primary key*. k*egunaan* dipilih sebagai *primary key* dikarenakan fungsi dari barang yang terdapat dalam tabel tersebut tidak sama, berikut contohnya

```
db_personal=> CREATE TABLE barang (
db_personal(> id INTEGER NOT NULL,
db_personal(> nama_barang VARCHAR(15),
db_personal(> PRIMARY KEY (id)
db_personal(> );
NOTICE:  CREATE TABLE/PRIMARY KEY will create implicit index 'barang_pkey' for table
'barang'
CREATE
```

kemudian masukkan beberapa data pada tabel *barang*

```
db_personal=> INSERT INTO barang
db_personal-> VALUES (2001, 'Komputer');
INSERT 53023 1

db_personal=> INSERT INTO barang
db_personal-> VALUES (2002, 'Buku Postgresql');
INSERT 53024 1

db_personal=> INSERT INTO barang
db_personal-> VALUES (2003, 'Mesin Cetak');
INSERT 53025 1

db_personal=> SELECT * FROM barang;
  id     |   nama_barang
----+-----------------
 2001 | Komputer
 2002 | Buku Postgresql
 2003 | Mesin Cetak
(3 rows)
```

Buatlah tabel kedua dengan nama *jenis_barang* yang memiliki dua *primary key id_barang* dan *kegunaan*, pada *field id_barang* diberi kata *REFERENCE barang* yang berfungsi untuk memastikan bahwa nilai *field* tersebut dipastikan terdapat pada *field primary key* milik tabel *barang,* id_barang pada tabel *jenis_barang* disebut FOREIGN KEY sebab mengandung *field id* yang sebenarnya merupakan PRIMARY KEY dari tabel *barang.* Jadi ketika kita memasukkan data pada tabel ini maka *id_barang* yang dimasukkan harus sama dengan *id*

pada tabel *barang,* jika tidak akan muncul pesan ERROR

```
db_personal=> CREATE TABLE jenis_barang (
db_personal(> id_barang INTEGER NOT NULL
db_personal(>   REFERENCES barang,
db_personal(> nama VARCHAR(10),
db_personal(> kegunaan VARCHAR(25) NOT NULL,
db_personal(> PRIMARY KEY (id_barang, kegunaan)
db_personal(> );
NOTICE:  CREATE TABLE/PRIMARY KEY will create implicit index 'jenis_barang_pkey' for
table 'jenis_barang'
NOTICE:  CREATE TABLE will create implicit trigger(s) for FOREIGN KEY check(s)
CREATE
```

Masukkan beberapa data pada tabel ini

```
db_personal=> INSERT INTO jenis_barang
db_personal-> VALUES (2001, 'Hardisk', 'Menyimpan Data');
INSERT 53047 1

db_personal=> INSERT INTO jenis_barang
db_personal-> VALUES (2001, 'Sound Card', 'Untuk Audio');
INSERT 53048 1

db_personal=> INSERT INTO jenis_barang
db_personal-> VALUES (2003, 'Mesin Foto ', 'Untuk Foto Copy');
INSERT 53049 1

db_personal=> SELECT * FROM jenis_barang;
 id_barang |    nama          |    kegunaan
-----------+----------+---------------
      2001 | Hardisk          | Menyimpan Data
      2001 | Sound Card  | Untuk Audio
      2003 | Mesin Foto   | Untuk Foto Copy
(3 rows)
```

Pesan *error* akan muncul ketika *id* yang dimasukkan berbeda dari *id* pada tabel *barang*.

```
db_personal=> INSERT INTO jenis_barang
db_personal-> VALUES (2005, 'Xerox', 'Foto Copy') ;
ERROR:  <unnamed> referential integrity violation - key referenced from jenis_barang
not found in barang
```

Setelah kedua tabel di atas terhubung atau saling berhubungan maka data pada tabel *barang* (tabel Induk) tidak dapat dihapus, sebab nilai *Primary Key*-nya sedang digunakan oleh tabel lain. Namun untuk kasus ini PostgreSQL telah menyiapkan ospi pelengkap REFERENCE di mana jika sebuah data dihapus maka data yang sama nomor ID-nya dengan tabel yang lain akan ikut terhapus juga. Sebagai contoh, hapuslah data pada salah satu tabel *barang*.

```
db_personal=> DELETE FROM barang
db_personal-> WHERE id = 2001 ;
ERROR:  <unnamed> referential integrity violation - key in barang still
referenced from jenis_barang
```

jadi ketika kita membuat tabel yang saling berhubungan sertakanlah opsi pelengkap REFERENCE agar nantinya data pada tabel tersebut dapat dihapus.

```
db_personal=> CREATE TABLE jenis_barang (
```

```
db_personal(> id_barang INTEGER NOT NULL
db_personal(>   REFERENCES barang ON DELETE CASCADE,
db_personal(> nama VARCHAR(10),
db_personal(> kegunaan VARCHAR(25) NOT NULL,
db_personal(> PRIMARY KEY (id_barang, kegunaan)
db_personal(> );
```

bahkan perubahan *id* pada tabel barang juga dapat mengubah nilai *id_barang* pada tabel *jenis_barang*, berikut contoh *query*-nya

```
db_personal=> CREATE TABLE jenis_barang (
db_personal(> id_barang INTEGER NOT NULL
db_personal(>   REFERENCES barang
db_personal(>       ON DELETE CASCADE
db_personal(>       ON UPDATE DISCADE
db_personal(> nama VARCHAR(10),
db_personal(> kegunaan VARCHAR(25) NOT NULL,
db_personal(> PRIMARY KEY (id_barang, kegunaan)
db_personal(> );
```

Query Multi Tabel

Ketika kita menghubungkan dua tabel, satu data dalam tabel pertama hanya terhubung dengan satu data pada tabel kedua, sehingga membuat hubungan *one-to-one*. Tetapi bagaimana jika customer melakukan lebih dari satu order, sehingga pada tabel satu data pada tabel customer akan dihubungkan dengan banyak data pada salesorder. Hubungn satu ke banyak seperti itu dikenal denga istilah *one-to-many*.

Sekarang, misalnya tidak ada *order* yang dibuat oleh seorang *customer*, maka data customer akan tetap ada meskipun tidak memliki data turunan di salesorder. Sehingga dalam proses query tidak akan ada nilai yang diembalikan. Kita dapat mengatakan atau memanggil situasi seperti ini sebagai penggabungan *one-to-none*. Dalam melakukan hubungan antar tabel harus terdapat satu buah kolom khusus yang berisi ID yang akan digunakan sebagai referensi dalam melakukan hubungan keduanya.

Dalam contoh kali ini akan dibuat sebuah tabel tbl_personal_child, tabel ini akan berisi data nama anak dari masing-masing nama yang terdapat dalam tabel tbl_personal. Sebelumnya akan dilihat lagi data pada tabel tbl_personal,

```
db_personal=> select * from tbl_personal ORDER By int_ID;
      1 | rofiq         | yuliardi      | 1999-07-14  | t           | Programmer
      2 | andhie        | lala          | 1960-08-08  | f           | Programmer
      3 | ade           | fajar         | 1980-11-01  | f           | Programmer
      4 | panuju        | sasongko      | 1970-09-12  | f           | System Ana
lyst
      5 | dudy          | rudianto      | 1973-12-11  | f           | Konsultan
      6 | ana           | hidayati      | 1983-10-01  | f           | Konsultan
```

Berdasarkan tabel tbl_personal akan dibuat sebuah tabel dengan satu buah kolom/field yang dijadikan referensi yaitu int_ID.

```
CREATE TABLE tbl_personal_child
(
  "int_ID" int4 NOT NULL,
  "txt_NamaDepan" varchar(10),
  "txt_NamaAkhir" varchar(10),
  "int_Umur" int4 DEFAULT 1
```

```
)

db_personal=> INSERT INTO tbl_personal_child VALUES (1, 'jhoni', 'yiliardi');
INSERT 0 1
db_personal=> INSERT INTO tbl_personal_child VALUES (1, 'ananda', 'yiliardi');
INSERT 0 1
db_personal=> INSERT INTO tbl_personal_child VALUES (2, 'sumanto', '');
INSERT 0 1
db_personal=> INSERT INTO tbl_personal_child VALUES (3, 'suharto', 'muhammad');
INSERT 0 1
```

Setelah proses input selesai, kemudian bisa dilihat hubungan kedua tabel dengan menggunakan WHERE yang membandingkan int_ID pada kedua tabel.

```
db_personal=> select tbl_personal.int_id, tbl_personal."txt_NamaDepan",
            tbl_Personal_child."txt_NamaDepan" from tbl_personal,tbl_personal_
           child Where tbl_personal."int_id" = tbl_personal_child."int_ID"
           ORDER By tbl_personal.int_ID;

1 | rofiq         | ananda
1 | rofiq         | jhoni
2 | andhie        | sumanto
3 | ade           | suharto
```

Dari kedua tabel tersebut terlihat bahwa rofiq mempunyai dua anak yang tersimpan dalam tabel tbl_personal_child, mendapatkan nama anak-anak tersebut dilakukan dengan membandingkan kode int_ID pada tabel child dengan int_ID pada tabel utama. Apabila sama maka hasilnya akan ditampilkan yang berarti menunjukan hubungan keduanya dalam query yang telah dibuat.

Alias Tabel

Ketika melakukan query pada dua tabel atau lebih maka nama tabel sebaiknya disertakan didepan nama kolom untuk menghindari terjadinya kebingungan pada server. Apalagi jika terdapat nama field yang sama pada kedua tabel tersebut, maka nama tabel wajib dituliskan sebagai prefiks nama kolom. Masalah yang kemudian muncul adalah ketikan nama tabel cukup panjang, dengan mengetikan nama tabel sebagai prefiks maka query menjadi jauh lebih panjang dan waktu mengetikannya juga lebih lama. Masalah tersebut dapat diatasi dengan menggunakan alias pada tabel yang bersangkutan. Sehingga nama alias tersebut yang akan dijadikan sebagai prefiks untuk mengganti nama tabel yang sesungguhnya.

Sebagai contoh akan digunakan query sebelumnya dan mengganti nama tabel tbl_personal dengan alias p, dan tbl_personal_child dengan alias c.

```
db_personal=> select p.int_id, p."txt_NamaDepan",
           c."txt_NamaDepan" from tbl_personal p,tbl_personal_child c Where
           p."int_id" = c."int_ID" ORDER By p.int_ID;

1 | rofiq         | ananda
1 | rofiq         | jhoni
2 | andhie        | sumanto
3 | ade           | suharto
```

Object Identification Numbers (OID)

Setiap *row* di dalam PostgreSQLdiberi sebuah kekhususan, biasanya pemanggilan nomor *object identification number* (OID), meskipun tidak kelihatan tetapi OID tetap ada. Counter OID biasanya berupa nomor khusus setiap dalam *row* walaupun *database* boleh dibuat dan dihapus, *counter* akan bertambah secara terus menerus. Ini digunakan oleh semua database, jadi nomor identifikasinya selalu khusus (*unique*) sehingga dalam setiap database tidak akan pernah memiliki OID yang sama.

Object identification number ini dapat dilihat pada saat selesai menambahkan sebuah data dengan INSERT, jika setelah penambahan data dieksekusi maka *object identification number* terdapat pada outputnya. berikut contohnya.

```
db_personal=> INSERT INTO tbl_personal_child VALUES (1, 'ananda', 'yiliardi');
INSERT 54245

db_personal=> INSERT INTO tbl_personal_child VALUES (2, 'sumanto', '');
INSERT 54246 1
```

Biasanya sebuah *row object identification number* dimunculkan atau ditampilkan hanya oleh *query* INSERT, namun OID dapat juga ditampilkan atau dimunculkan dengan *non-query*, jadi OID dapat dilihat dengan menggunakan SELECT. Setiap query tabel pada PostgreSQL pemanggilan kolom OID-nya tak diperlihatkan, seperti pemangilannya menggunakan *query* berikut ini *SELECT * FROM nama_tabel*. Agar OID dapat dilihat atau ditampilkan maka *query*-nya harus spesifik, dengan menetikan OID sebagai perameter dalam query.

*SELECT oid, * FROM nama_tabel*.

Untuk lebih jelasnya lihat contoh di bawah ini.

```
db_personal=> select oid, * from tbl_personal ORDER By int_ID;
50011  | 1 | rofiq         | yuliardi      | 1999-07-14  | t           | Programmer
50012  | 2 | andhie        | lala          | 1960-08-08  | f           | Programmer
50013  | 3 | ade           | fajar         | 1980-11-01  | f           | Programmer
50014  | 4 | panuju        | sasongko      | 1970-09-12  | f           | System Ana
lyst
50015  | 5 | dudy          | rudianto      | 1973-12-11  | f           | Konsultan
50016  | 6 | ana           | hidayati      | 1983-10-01  | f           | Konsultan
```

Serial dan Big Serial

Tipe data serial dan bigserial sebenarnya bukan merupakan tipe data, tetapi lebih pada sebuah notasi yang digunakan untuk memberikan nomor identifikasi yang terus bertambah dalam sebuah kolom. Tipe data serial ini bisa dikatakan seperti AUTO_INCREMENT dalam beberapa database lain. Jika kita menentukan atau menetapkan suatu kolom dengan tipe SERIAL maka dengan otomatis sebuah urutan akan dibuat dan DEFAULT urutan akan ditentukan ke kolom tersebut. Lihat contoh di bawah ini, pada baris pertama menunjukkan bahwa sebuah *sequence* telah dibuat untuk kolom tipe SERIAL.

```
 db_personal=> CREATE TABLE tes_serial (
 db_personal(>              customer_id SERIAL,
 db_personal(>              nama CHAR(25)
 db_personal(> );
NOTICE:  CREATE TABLE will create implicit sequence 'tes_serial_customer_id_seq' for
SERIAL column 'tes_serial.customer_id'
```

```
    NOTICE:  CREATE TABLE/UNIQUE will create implicit index 'tes_serial_customer_id_key'
    for table 'tes_serial'
    CREATE
     db_personal=>
     db_personal=> \d tes_serial
                    Table "tes_serial"
      Attribute        |          Type           |             Modifier
    -----------+------------+-------------------
     customer_id   |    integer            |  not null default
nextval('"tes_serial_customer_id_seq"'::text)
     nama              |    character(25) |  Index: tes_serial_customer_id_key
     db_personal=> INSERT INTO tes_serial (nama) VALUES ('Stenli van Harlen');
    INSERT 61530 1
     db_personal=>
     db_personal=> SELECT * FROM tes_serial;
     customer_id   |          nama
    -----------+-----------------
              1            |    Stenli van Harlen
    (1 row)
```

BLOBS

PostgreSQL tidak dapat menyimpan nilai yang lebih dari beberapa ribu byte menggunakan tipe data standar. Namun, semua ini tidak menjadi masalah karena pada PostgreSQL terdapat *large objects* (objek yang besar atau luas) yang dapat juga dipanggil *binary large object* atau BLOBs, BLOBs inilah yang biasanya menyimpan nilai yang sangat besar sekalipun dan binari data. Pengisian file yang besar ke dalam database postgres menggunakan *lo_import()*, dan untuk mendapatkannya kembali dari database gunakan *lo_export(),* berikut contohnya.

catatan: Untuk melakukan proses lo_import, lo_export, dan lo_unlink (BLOBs), user-nya harus sebagai postgres superuser.

Langkah pertama, loginlah sebagai postgres super user.

```
    $ psql db_personal postgres
    Welcome to psql 8.1.1, the PostgreSQL interactive terminal.

    Type:  \copyright for distribution terms
           \h for help with SQL commands
           \? for help on internal slash commands
           \g or terminate with semicolon to execute query
           \q to quit
    db_personal=#
```

Kemudian, buatlah sebuah tabel baru sebagai tempat untuk menyimpan file-file yang ingin kita import ke dalam database *coba*, berikut query-nya.

```
    db_personal=# CREATE TABLE hasil (
    db_personal(#              nama  CHAR(60),
    db_personal(#              image OID
    db_personal(#  );
    CREATE
```

Setelah itu, imporlah file yang diinginkan, misalnya kita ingin mengimpor sebuah file gambar dari direktori **/usr/images.** Berikut query-nya.

```
    db_personal=# INSERT INTO hasil
    db_personal-# VALUES ('Nida', lo_import('/usr/images/fott11.jpg'));
    INSERT 35488 1
```

File yang diimpor masuk ke dalam database postgres tidak terbatas ukurannya. Jadi, file sebesar apa pun dapat kita impor masuk ke dalam database postgres. Dari query di atas, kita mencoba mengimpor file gambar yang diambil dari direktori */usr/images,* sedangkan *fott11.jpg* adalah nama filenya. Setelah proses impornya berhasil, akan tersimpan di dalam database postgres, seperti contoh dibawah ini.

```
db_personal=# SELECT * FROM hasil;
    nama   | image
-----------+-------
 Nida       | 35753
(1 rows)
```

Untuk mendapatkan kembali atau melihat isi file tersebut kita harus mengekspornya terlebih dahulu ke salah satu direktori, query untuk mengekspornya seperti terlihat dibawah ini.

```
db_personal=# SELECT lo_export(hasil.image, '/tmp/outimage.jpg')
db_personal-# FROM hasil
db_personal-# WHERE nama = 'Nida';
 lo_export
---------------
         1
(1 row)
```

File *hasil.images* merupakan OID *(Object Identification Number)* dan *tmp* adalah nama direktori yang akan dituju sebagai tempat penyimpanan selanjutnya. *Outimage.jpg* merupakan nama file baru dari file *image* tersebut. Perhatikan *permission* (ijin akses) direktori yang kita tuju. Jika kita ingin mengekspor file tersebut ke direktori */home/nama_user*, permisi dari user tersebut perlu diubah terlebih dahulu baru kemudian bisa diekspor. Setelah proses *lo_export* berhasil, file *images* tersebut langsung dapat dilihat pada direktori yang dituju tadi, yaitu *tmp* dengan nama filenya *outimage.jpg*.

Setelah kita berhasil mendapatkan kembali atau mengekspor file *image* tadi, file *large object* bisa di hapus menggunakan perintah *lo_unlink*, secara otomatis file tersebut tidak dapat diekspor lagi. Berikut contoh query-nya.

```
db_personal=# SELECT lo_unlink(hasil.image) FROM hasil;
 lo_unlink
--------------
         1
(1 row)
```

Berikut ini contoh oengguaan BLOB lainnya.

```
db_personal=# CREATE TABLE gambar  (
db_personal(#              nama        CHAR(10),
db_personal(#              jenis_foto  CHAR(15),
db_personal(#              image OID
db_personal(#  );
CREATE

db_personal=# INSERT INTO gambar VALUES
db_personal-# ('Nida', 'Close-Up', lo_import('/mnt/win_d/nidha/fot2.jpg'));
INSERT 35844 1

db_personal=# INSERT INTO gambar VALUES
db_personal-# ('Rini', 'Close-Up', lo_import('/mnt/win_d/nidha/asti.jpg'));
INSERT 35883 1

db_personal=# INSERT INTO gambar VALUES
```

```
db_personal-# ('andy', 'Close-Up', lo_import('/mnt/win_d/nidha/fott1.jpg'));
INSERT 35895 1

db_personal=# SELECT * FROM gambar;
    nama  |   jenis_foto | size
------------+--------------+-------
 Nida            | Close-Up    | 35825
 Rini            | Close-Up    | 35845
 andy            | Close-Up    | 35884
(3 rows)

db_personal=# SELECT lo_export(gambar.size, '/home/andhie/nida1.jpg')
db_personal-# FROM gambar WHERE nama = 'Nida';
 lo_export
---------------
         1
(1 row)

db_personal=# SELECT lo_export(gambar.size, '/home/andhie/Jeng.jpg')
db_personal-# FROM gambar WHERE nama = 'Rini';
 lo_export
--------------
         1
(1 row)

db_personal=# SELECT lo_export(gambar.size, '/home/andhie/andi.jpg')
db_personal-# FROM gambar WHERE nama = 'andy';
 lo_export
--------------
         1
(1 row)
```

Backslash dan NULL

Jika sebuah data yang di INSERT ke dalam tabel mengggunakan karakter yang terdapat pada DELIMITERS, akan berpotensial mengalami kekacauan. COPY menghindari atau membatalkan apapun kekacauan oleh tanda khusus *delimiter* yang ditimbulkan dalam data *user*.

String Backslash	Keterangan
\TAB	Jika menggunakan default delimiter tab
\ |	*pipe* jika menggunakan *pipe* sebagai delimiter
\ N	NULL jika menggunakan default NULL output
\ b	Backspace
\ f	pemberian form
\ n	baris baru (newline)
\ r	hasil berurutan
\ t	tab

String Backslash	Keterangan
\ v \ ###	tab vertikal representasi karakter nomor oktal ###
\\	Backslash

Didahului dengan sebuah *backslash* (\), berikut contohnya:

```
 db_personal=> CREATE TABLE tes2 (
 db_personal(>                    huruf TEXT,
 db_personal(>                    nama  VARCHAR(20)
 db_personal(> );
CREATE

 db_personal=> INSERT INTO tes2
 db_personal-> VALUES ('abc|def', 'abjad');
INSERT 70723 1

 db_personal=> INSERT INTO tes2
 db_personal-> VALUES ('1234|56', NULL);
INSERT 70724 1

 db_personal=> SELECT * FROM tes2;
  huruf     |  nama
-------+-------
 abc|def  |  abjad
 1234|56 |
(2 rows)

 db_personal=> COPY tes2 TO stdout USING DELIMITERS '|';
abc\|def|abjad
1234\|56|\N
```

PERINTAH DASAR SQL

Bab 6 – Perintah Dasar SQL

Deskripsi

Bab ini akan membahas perintah-perintah dasar dalam standar SQL. Perintah dasar yang akan dibahas meliputi query untuk menampilkan, menambah, menghapus dan memanipulasi data dalam tabel.

Obyektif

Pada bab ini diharapkan peserta dapat memahami dan menggunakan perintah dasar SQl yang berhubungan dengan manipulasi data pada tabel.

Outline

- Memasukkan Data
- Menampilkan Data
- Memilih Data
- Menghapus Data
- Modifikasi Data
- Mengurutkan Data

Memasukkan Data

Sejauh ini kita telah membuat sebuah tabel, namun tabel tersebut belum berisikan data. Sebuah data dapat dimasukkan ke dalam tabel menggunakan perintah INSERT. Dalam pembuatan tabel dengan perintah CREATE TABLE telah ditentukan format spesifik sebuah kolom/field, format tersebut akan menantukan tipe data yang bisa dimasukan ke dalamnya. Misalnya sebuah kolom dengan tipe boolean hanya bisa diisi dengan data TRUE atau FALSE, sedangkan untuk data int4 hanya bisa diisi dengan angka.

Dalam pengisian tabel harus memperhatikan karakter stringnya, ketika kita memasukkan sebuah karakter dalam kolom bertipe varchar atau string lainya, maka data harus diawali dan diakhiri dengan tanda petik tunggal ('). Sedangkan untuk yang bertipe khusus seperti bit, integer, boolean, dan lain-lain – biasanya berupa angka atau data yang sudah ditentukan nilainya – maka tanda petik tidak diperlukan. Sedangkan untuk spasi dan huruf kapital hanyalah opsional, berikut kita akan mencoba memasukkan data ke dalam tabel tbl_personal.

```
INSERT INTO tbl_personal
VALUES (1, 'rofiq', 'yuliardi',
           '1999-07-14',False,
           ‘Programmer')
```

Berikut adalah penjelasan dari perintah diatase;

- Kolom pertama (1) - karena tipe datanya adalah integer maka tanda petik tidak digunakan.
- Kolom kedua (‚rofiq’), ketiga (‚yuliardi’), dan terakhir (‚programmer’) – karena tipe data untuk nama_depan adalah varchar maka tanda petik diperlukan, apabila tidak maka query akan dianggap salah.
- Kolom keempat (1999-07-14) – tipe datanya adalah date dan tipe date dianggap sebagai sebuah string sehingga harus menggunakan tanda petik.
- Kolom kelima (False), - tipe data pada kolom ini adalah boolean yang hanya mengijinkan isian true atau false sehingga tidak perlu tanda petik.

Data selanjutnya bisa ditambahkan dan dimasukkan ke dalam tabel tbl_personal dengan perintah yang sama, contohnya seperti berikut ini.

```
INSERT INTO tbl_personal
VALUES (2, 'andhie', 'lala',
           '1960-08-08',False,
'Programmer');

INSERT INTO tbl_personal
VALUES (3, 'ade', 'fajar',
           '1980-11-01',False,
           'Programmer');

INSERT INTO tbl_personal
VALUES (4, 'panuju', 'sasongko',
           '1970-09-12',False,
           'System Analyst');
INSERT INTO tbl_personal
VALUES (5, 'dudy', 'rudianto',
```

```
'1973-12-11',False,
'Konsultan');
```

Menampilkan Data

Dalam database postgresql, perintah untuk menampilkan sebuah tabel adalah menggunakan SELECT. Select merupakan sintak dalam SQL. Setelah kita melakukan penambahan data dalam tabel, cobalah tampilkan keseluruhan kolom dari tabel yang telah dibuat tadi. Kita juga dapat menampilkan sebagian dari atribut tabel. Query-nya adalah *SELECT txt_NamaDepan,txt_NamaAkhir, dt_TglLahir FROM tbl_personal;* artinya attribut dari tabel yang ditampilkan hanya berdasarkan field yang dipilih. Berikut contohnya.

```
select "txt_NamaDepan","txt_NamaAkhir","dt_TglLahir" From tbl_personal;

 txt_NamaDepan | txt_NamaAkhir | dt_TglLahir
---------------+---------------+-------------
 rofiq         | yuliardi      | 1999-07-14
 andhie        | lala          | 1960-08-08
 ade           | fajar         | 1980-11-01
 panuju        | sasongko      | 1970-09-12
 dudy          | rudianto      | 1973-12-11
(5 rows)
```

Kemudian, jika ingin menampilkan seluruh isi tabel, gunakan perintah *SELECT * FROM tbl_personal;*. Tanda asterik (*) pada perintah ini berarti semua kolom beserta data yang berada dalam sebuah tabel database akan ditampilkan.

```
SELECT * FROM tbl_personal;

db_personal=> select * from tbl_personal;
 int_id | txt_NamaDepan | txt_NamaAkhir | dt_TglLahir | bol_IsNikah | txt_Pekerj
aan
--------+---------------+---------------+-------------+-------------+-----------
-----
      1 | rofiq         | yuliardi      | 1999-07-14  | f           | Programmer
      2 | andhie        | lala          | 1960-08-08  | f           | Programmer
      3 | ade           | fajar         | 1980-11-01  | f           | Programmer
      4 | panuju        | sasongko      | 1970-09-12  | f           | System Ana
lyst
      5 | dudy          | rudianto      | 1973-12-11  | f           | Konsultan
(5 rows)
```

Memilih Data

Output seluruh data yang terdapat pada kolom dan baris dari sebuah tabel dapat dikontrol dengan perintah SELECT. Pada sesi sebelumnya, kita telah mempelajari cara menampilkan sebagian data dan tampilannya merupakan sebagian dari kolom pada tabel tersebut. Pada bagian ini kita akan mencoba untuk menampilkan data secara spesifik menggunakan perintah WHERE. Perintah WHERE berfungsi untuk menampilkan sekaligus mencari sebuah data secara spesifik dan outputnya berupa sebagian baris dari keseluruhan tabel. Misalnya, kita ingin mencari data *kota* dari tabel tersebut, untuk lebih spesifik langsung dapat menyebutkan nama kotanya. Berikut ini contoh query-nya *SELECT * FROM tbl_personal WHERE int_ID = 1';*

```
SELECT * FROM tbl_personal WHERE int_ID = '1';

db_personal=> select * from tbl_personal where int_ID = 1;
```

```
 int_id | txt_NamaDepan | txt_NamaAkhir | dt_TglLahir | bol_IsNikah | txt_Pekerj
aan
--------+---------------+---------------+-------------+-------------+-----------
----
      1 | rofiq         | yuliardi      | 1999-07-14  | f           | Programmer
(1 row)
```

WHERE juga dapat menangani permintaan untuk menampilkan data yang lebih kompleks lagi. Misalkan kita ingin menampilkan atau mencari data umur yang kurang dari (<) atau sama dengan (=) 20 tahun atau mencari yang tanggal lahirnya lebih kecul dari 1980-12-02, berikut contohnya.

```
select * from tbl_personal where "dt_TglLahir" < 1980-12-02;
db_personal=> select * from tbl_personal where "dt_TglLahir" < 1980-12-02;
 int_id | txt_NamaDepan | txt_NamaAkhir | dt_TglLahir | bol_IsNikah | txt_Pekerj
aan
--------+---------------+---------------+-------------+-------------+-----------
----
      2 | andhie        | lala          | 1960-08-08  | f           | Programmer
(1 row)
```

Query untuk menampilkan *nama depan* dan *nama belakang* yang berumur lahirnya setelah 1980-12-02.

```
db_personal=> select "txt_NamaDepan","txt_NamaAkhir" from tbl_personal where "dt
_TglLahir" > 1980-12-02;
 txt_NamaDepan | txt_NamaAkhir
---------------+---------------
 rofiq         | yuliardi
 ade           | fajar
 panuju        | sasongko
 dudy          | rudianto
(4 rows)
```

Menampilkan semua isi tabel yang nama depannya yuliardi.

```
db_personal=> select "txt_NamaDepan","txt_NamaAkhir" from tbl_personal where "tx
t_NamaDepan" = 'rofiq';
 txt_NamaDepan | txt_NamaAkhir
---------------+---------------
 rofiq         | yuliardi
(1 row)
```

Menghapus Data

Pada database postgres, DELETE digunakan untuk menghapus data pada sebuah tabel. Misalkan, perintah *DELETE FROM tbl_personal*, artinya semua baris dari tabel tbl_personal akan dihapus. Perintah DELETE juga dapat diberik kondisi seperti perintah select, misalnya query *DELETE FROM tbl_personal WHERE "int_ID" = 10*, maka hanya baris dan kolom yang memiliki ID = 5 yang akan dihapus. Untuk mencoba sebaiknya ditambahkan dulu satu nama lagi pada tabel *tbl_personal* dan kemudian menghapusnya.

```
       INSERT INTO tbl_personal VALUES (10, 'ana', 'hidayati', '1983-10-
01',False,'Konsultan');
    (6 rows)
```

Gunakan perintah select untuk memastikan bahwa data baru sudah dimasukan,

```
int_id | txt_NamaDepan | txt_NamaAkhir | dt_TglLahir | bol_IsNikah | txt_Pekerj
aan
--------+---------------+---------------+-------------+-------------+-----------
----
     10 | ana           | hidayati      | 1983-10-01  | f           | Konsultan
(1 row)
```

Kemudian, hapus lagi data yang baru ditambahkan tadi.

```
db_personal=> select * from tbl_personal where int_ID = 10;

db_personal=> delete from tbl_personal where int_ID = 10;
DELETE 1
```

Gunakan kembali perintah select untuk memastikan bahwa data baru sudah dihapus, proses penghapusan berhasil jika hasilnya adlah 0 rows.

```
db_personal=> select * from tbl_personal where int_ID = 10;
 int_id | txt_NamaDepan | txt_NamaAkhir | dt_TglLahir | bol_IsNikah | txt_Pekerj
aan
--------+---------------+---------------+-------------+-------------+-----------
----
(0 rows)
```

Modifikasi Data

Semua data yang telah dibuat dan terdapat di dalam database dapat diganti atau dimodifikasi sesuai dengan keinginan. Kita dapat menggunakan *DELETE* untuk menghapus sebuah baris sehingga secara otomatis data yang terdapat di dalam baris tersebut ikut terhapus. Kemudian, gunakan INSERT untuk menambah baris baru namun itu sangat tidak efisien. Agar lebih efisien, gunakan *UPDATE* untuk memodifikasi baris tersebut, karena data yang ingin dimodifikasi dapat langsung diganti tanpa harus menghapus data lain yang berada dalam satu baris dengan data tersebut.

Salah satu contoh dari tabel *tbl_personal* adalah ketika *Rofiq* yang semula belum menikah kini telah menikah sehingga data tabel *tbl_personal* perlu di modifikasi atau diganti. Sebagai catatan, setiap baris yang diganti atau dimodifikasi, baris tersebut akan bergeser dan perpindah ke posisi paling bawah dari tabel tersebut, ikuti contoh ini.

```
db_personal=> update tbl_personal set "bol_IsNikah" = True where "txt_NamaDepan"
= 'rofiq';
UPDATE 1
```

Setelah itu, cobalah tampilkan kembali data yang telah diubah,

```
db_personal=> select * from tbl_personal where "txt_NamaDepan" = 'rofiq';
 int_id | txt_NamaDepan | txt_NamaAkhir | dt_TglLahir | bol_IsNikah | txt_Pekerj
aan
--------+---------------+---------------+-------------+-------------+-----------
----
      1 | rofiq         | yuliardi      | 1999-07-14  | t           | Programmer
(1 row)
```

Mengurutkan Data

Pada tabel tbl_personal yang telah dibuat, dapat diurutkan langsung pada nama atribut dari tabel tersebut. Maksud dari mengurutkan data dengan ORDER BY adalah jika data sebuah tabel diurutkan dengan ORDER BY, data yang diurutkan tadi akan berubah kedudukannya dari atas kebawah menurut abjad. ini berlaku untuk data yang menggunakan huruf, namun untuk yang menggunakan angka, data akan diurutkan dari atas mulai yang angkanya paling kecil sampai yang terbesar. Ada juga perintah pengurutan menggunakan ORDER BY yang diakhiri dengan kata DESC, fungsinya kebalikan dari ORDER BY. Bedanya, jika memakai DESC, urutannya akan dimulai dari bawah ke atas. Secara default apabila tidak disebutkan ASC atau DESC maka pengurutan akan dilakukan secara ASC. lihat contoh berikut ini.

```
db_personal=> select * from tbl_personal ORDER BY "dt_TglLahir";
 int_id | txt_NamaDepan | txt_NamaAkhir | dt_TglLahir | bol_IsNikah | txt_Pekerj
aan
--------+---------------+---------------+-------------+-------------+-----------
-----
      2 | andhie        | lala          | 1960-08-08  | f           | Programmer
      4 | panuju        | sasongko      | 1970-09-12  | f           | System Ana
lyst
      5 | dudy          | rudianto      | 1973-12-11  | f           | Konsultan
      3 | ade           | fajar         | 1980-11-01  | f           | Programmer
      5 | ana           | hidayati      | 1983-10-01  | f           | Konsultan
      1 | rofiq         | yuliardi      | 1999-07-14  | t           | Programmer
(6 rows)
```

Berikut contoh pengurutan dengan ORDER BY yang diakhiri dengan DESC.

```
db_personal=> select * from tbl_personal ORDER BY "dt_TglLahir" DESC;
 int_id | txt_NamaDepan | txt_NamaAkhir | dt_TglLahir | bol_IsNikah | txt_Pekerj
aan
--------+---------------+---------------+-------------+-------------+-----------
-----
      1 | rofiq         | yuliardi      | 1999-07-14  | t           | Programmer
      5 | ana           | hidayati      | 1983-10-01  | f           | Konsultan
      3 | ade           | fajar         | 1980-11-01  | f           | Programmer
      5 | dudy          | rudianto      | 1973-12-11  | f           | Konsultan
      4 | panuju        | sasongko      | 1970-09-12  | f           | System Ana
lyst
      2 | andhie        | lala          | 1960-08-08  | f           | Programmer
(6 rows)
```

DAFTAR GAMBAR

Panduan
Administrasi Database PostgreSQL

Versi 1.4

BUKU 2

Sangkalan

Pada Intinya Perintah Dasar SQL Termasuk Administrasi Database PostgreSQL Adalah Identik, Tetapi Untuk GUI dan Perintah Lanjut Buku Pada Buku Ini Mungkin Tidak Kompatibel Lagi Dengan Versi Database Terbaru atau versi yang Anda Gunakan

Artikel dan e-book lainnya dapat di peroleh di www.yuliardi.com

Written and Published by

Rofiq Yuliardi
Web : www.yuliardi.com
Email : rofiq@yuliardi.com , rofiqy@gmail.com
YahooID : rofiqy2000
Phone : 0852-160-88127

DAFTAR ISI

MENGENAL OPERATOR DASAR

Bab 7 – Mengenal Operator Dasar

Deskripsi

Bab ini akan membahas operator dasar dalam standar SQL. Operator dasar yang akan dibahas meliputi operator untuk melakukan query yang lebih detail terhadap data dalam tabel.

Obyektif

Pada bab ini diharapkan peserta dapat memahami dan menggunakan operator dasar SQl yang berhubungan dengan manipulasi data pada tabel.

Outline

- AS
- AND dan OR
- BETWEEN
- LIKE
- CASE
- DISTINCT
- SET, SHOW dan RESET
- LIMIT
- UNION, EXCEPT, dan INTERSECT

Setiap data yang terdapat dalam SQL dapat disetting sesuai dengan kebutuhan, sebuah bahasa SQL tersusun dari bermacam-macam *key words* (kata kunci). Di antaranya Arithmatic dan Procedural yang mana dalam penggunaannya selalu diikuti dengan expresi. Berikut adalah beberapa operator dasar SQL.

AS

Label biasanya digunakan sebagai nama lain dari sebuah kolom yang dipilih, selain itu kita juga dapat mengontrol teks sebuah atribut yang digunakan untuk memanggil suatu kolom dengan menggunakan *AS*. AS biasa digunakan untuk menampilkan label kolom dengan nama lain sehingga yang akan muncul dalam hasil query bukan nama asli kolom, tetapi nama yang mungkin lebih sesuai dan mudah dimengerti. AS digunakan setelah nama kolom yang akan diganti yang kemudian diikuti dengan nama penggantinya.

```
db_personal=> select "txt_NamaDepan" AS "Nama Depan","txt_NamaAkhir" AS "Nama Be
lakang" from tbl_personal;
 Nama Depan | Nama Belakang
------------+---------------
 andhie     | lala
 ade        | fajar
 panuju     | sasongko
 dudy       | rudianto
 ana        | hidayati
 rofiq      | yuliardi
(6 rows)
```

AND dan OR

Pada bagian sebelumnya kita menggunakan anak kalimat atau sintaks WHERE hanya pada konteks yang sederhana. Berikut ini kita akan mencoba menggunakan *WHERE* untuk konteks yang lebih kompleks lagi, anak kalimat *where* yang kompleks akan bekerja dengan baik dengan menggunakan kata *AND* dan *OR*.
SQL menggunakan standar logika *boolean three-valued* seperti pada tabel berikut;

a	***b***	***a* AND *b***	***a* OR *b***
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

a	**NOT *a***
TRUE	FALSE
FALSE	TRUE
NULL	NULL

AND digunakan untuk menampilkan data dengan dua atau kondisi yang harus dipenuhi, apabila salah satu tidak terpenuhi maka pencarian tidak akan ada hasilnya. Contohnya adalah apabila akan mencari data yang nama depan ‘rofiq’ dan pekerjaanya ‘programmer’. Kedua kondisi tersebut harus dipenuhi, sehingga apabila dalam tabel terdapat nama rofiq tetapi pekerjaanya konsultan, maka hasil pencarian akan kosong.
Sedangkan OR digunakan untuk menampilkan data dengan dua atau lebih kondisi, tetapi

pencarian akan ada hasilnya meskipun salah satu kondisi saja yang terpenuhi. Contohnya adalah apabila akan mencari data yang nama depan 'rofiq' atau orang lain dengan pekerjaan 'programmer'. Kedua kondisi tersebut harus dipenuhi minimal salah satu, sehingga apabila dalam tabel terdapat tidak nama rofiq tetapi terdapat pekerjaan konsultan, maka hasil pencarian tetap diperoleh.

```
db_personal=>  select * from tbl_personal Where "txt_NamaDepan" = 'rofiq' and "t
xt_Pekerjaan" = 'Programmer';
 int_id | txt_NamaDepan | txt_NamaAkhir | dt_TglLahir | bol_IsNikah | txt_Pekerj
aan
--------+---------------+---------------+-------------+-------------+-----------
----
      1 | rofiq         | yuliardi      | 1999-07-14  | t           | Programmer
(1 row)
```

Contoh penggunaan OR.

```
db_personal=>  select * from tbl_personal Where "txt_NamaDepan" = 'rofiq' or "tx
t_Pekerjaan" = 'Konsultan';
 int_id | txt_NamaDepan | txt_NamaAkhir | dt_TglLahir | bol_IsNikah | txt_Pekerj
aan
--------+---------------+---------------+-------------+-------------+-----------
----
      5 | dudy          | rudianto      | 1973-12-11  | f           | Konsultan
      5 | ana           | hidayati      | 1983-10-01  | f           | Konsultan
      1 | rofiq         | yuliardi      | 1999-07-14  | t           | Programmer
(3 rows)
```

Dalam beberapa kasus sangat dimungkinkan untuk menggabungkan antar AND dan OR, contoh berikut menunjukan gabungan antara OR dan AND.

```
db_personal=>  select * from tbl_personal Where "txt_NamaDepan" = 'rofiq' and "t
xt_Pekerjaan" = 'Programmer' or "dt_TglLahir" < '1980-01-01';
 int_id | txt_NamaDepan | txt_NamaAkhir | dt_TglLahir | bol_IsNikah | txt_Pekerj
aan
--------+---------------+---------------+-------------+-------------+-----------
-----
      2 | andhie        | lala          | 1960-08-08  | f           | Programmer
      4 | panuju        | sasongko      | 1970-09-12  | f           | System Ana
lyst
      5 | dudy          | rudianto      | 1973-12-11  | f           | Konsultan
      1 | rofiq         | yuliardi      | 1999-07-14  | t           | Programmer
(4 rows)
```

BETWEEN

Between digunakan untuk menentukan lebar nilai yang akan di seleksi, penentuan lebar ini dilakukan dengan menentukan nilai terendah dan nilai tertinggi. Operator yang dapat digunakan dalam between adalah operator pembanding seperti pada tabel berikut;

Operator	Description
<	Kurang dari
>	Lebih dari
<=	Kurang dari atau sama dengan
>=	Lebih dari atau sama dengan
=	Sama dengan
<> *or* !=	*Tidak sama dengan*

Contoh logika operator between adalah sebagai berikut;

- Nilai a dalam formula "a **BETWEEN x AND y**" indentik dengan "**a >= x AND a <= y**"
- Sementara "**a NOT BETWEEN x AND y** " identik dengan "**a < x OR a > y**"

Contoh berikut ini memberikan batasan data personal yang akan ditampilkan adalah orang yang mempunyai tanggal lahir antara 1980-01-01 dan 1990-01-01.

```
db_personal=>  select * from tbl_personal Where "dt_TglLahir" Between '1980-01-0
1' AND '1990-01-01';
 int_id | txt_NamaDepan | txt_NamaAkhir | dt_TglLahir | bol_IsNikah | txt_Pekerj
aan
--------+---------------+---------------+-------------+-------------+-----------
----
      3 | ade           | fajar         | 1980-11-01  | f           | Programmer
      5 | ana           | hidayati      | 1983-10-01  | f           | Konsultan
(2 rows)
```

LIKE

LIKE digunakan untuk melakukan seleksi atau pemilihan tetapi tidak seperti halnya sama dengan (=) yang hanya akan menampilkan data yang benar-benar sesuai *(match)* dengan parameternya, LIKE akan menampilkan data yang mengandung string parameter yang dimasukan, meskipun hanya satu karakter saja yang sama. Singkatnya perbandingan LIKE digunakan ketika kita ingin mencari sebuah data yang hanya diwakili oleh salah satu atau lebuh hurufnya saja. Misalkan kita ingin mencari nama yang huruf awalnya *R*, maka kita harus menggunakan *LIKE.*

Simbol % dalam LIKE digunakan untuk merepresentasikan string kosong atau banyak string dalam parameter, sedangkan tanda garis bawah (_) merupakan represenrtasi satu karakter saja.

Untuk menghasilkan kebalikan dari operator LIKE ini maka dapat digunakan tambahan operator NOT di depan LIKE, sehingga menjadi NOT LIKE. Berikut adalah tabel contoh penggunaan LIKE.

Kasus	Operator
Mulai dengan D	LIKE ' D%'
Diakhiri dengan D	LIKE ' %D '
Huruf D pada posisi ke dua	LIKE ' _D% '
Mulai dengan D dan Berisikan e	LIKE ' D%e% '
Mulai dengan D, berisikan e, kemudian f	LIKE ' D%e%f% '
Mulai dari bukan D	NOT LIKE ' D% '

```
db_personal=>  select * from tbl_personal Where "txt_NamaDepan" Like 'an%';
 int_id | txt_NamaDepan | txt_NamaAkhir | dt_TglLahir | bol_IsNikah | txt_Pekerj
aan
--------+---------------+---------------+-------------+-------------+-----------
----
      2 | andhie        | lala          | 1960-08-08  | f           | Programmer
      5 | ana           | hidayati      | 1983-10-01  | f           | Konsultan
(2 rows)
```

```
db_personal=>  select * from tbl_personal Where "txt_NamaDepan" Like '%e';
 int_id | txt_NamaDepan | txt_NamaAkhir | dt_TglLahir | bol_IsNikah | txt_Pekerj
aan
--------+---------------+---------------+-------------+-------------+-----------
----
      2 | andhie        | lala          | 1960-08-08  | f           | Programmer
      3 | ade           | fajar         | 1980-11-01  | f           | Programmer
(2 rows)

db_personal=>  select * from tbl_personal Where "txt_NamaDepan" Like '%d%';
 int_id | txt_NamaDepan | txt_NamaAkhir | dt_TglLahir | bol_IsNikah | txt_Pekerj
aan
--------+---------------+---------------+-------------+-------------+-----------
----
      2 | andhie        | lala          | 1960-08-08  | f           | Programmer
      3 | ade           | fajar         | 1980-11-01  | f           | Programmer
      5 | dudy          | rudianto      | 1973-12-11  | f           | Konsultan
(3 rows)
```

Contoh penggunaan LIKE dengan garis bawah (_)

```
db_personal=> select * from tbl_personal where "txt_NamaDepan" LIKE 'a_';
 int_id | txt_NamaDepan | txt_NamaAkhir | dt_TglLahir | bol_IsNikah | txt_Pekerj
aan
--------+---------------+---------------+-------------+-------------+-----------
----
(0 rows)

db_personal=> select * from tbl_personal where "txt_NamaDepan" LIKE 'a__';
 int_id | txt_NamaDepan | txt_NamaAkhir | dt_TglLahir | bol_IsNikah | txt_Pekerj
aan
--------+---------------+---------------+-------------+-------------+-----------
----
      3 | ade           | fajar         | 1980-11-01  | f           | Programmer
      5 | ana           | hidayati      | 1983-10-01  | f           | Konsultan
(2 rows)
```

Contoh penggunaan NOT LIKE.

```
db_personal=>  select * from tbl_personal Where "txt_NamaDepan" NOT Like '%d%';
 int_id | txt_NamaDepan | txt_NamaAkhir | dt_TglLahir | bol_IsNikah | txt_Pekerj
aan
--------+---------------+---------------+-------------+-------------+-----------
-----
      4 | panuju        | sasongko      | 1970-09-12  | f           | System Ana
lyst
      5 | ana           | hidayati      | 1983-10-01  | f           | Konsultan
      1 | rofiq         | yuliardi      | 1999-07-14  | t           | Programmer
(3 rows)
```

CASE

Banyak bahasa pemrograman yang memiliki ketergantungan pada sebuah kondisi sebelumnya. Dengan kata lain jika sebuah kondisi *true* maka harus melakukan sesuatu sesuai dengan perintah pada kondisinya tersebut (true *then* do, *else* do *else*), bentuk struktur seperti ini digunakan untuk mengeksekusi dari *statement* dasar ke dalam beberapa kondisi. Meskipun SQL bukan merupakan sebuah prosedur bahasa perograman, namun dalam prosesnya dapat dengan bebas mengontrol data yang kembali dari *query*. Kata WHERE menggunakan perbandingan untuk mengontrol pemilihan data, sedangkan *CASE*

perbandingan dalam bentuk output kolom. Jadi intinya penggunaan *CASE* akan membentuk output tersendiri berupa sebuah kolom baru dengan data dari operasi yang di dalamnya.

CASE WHEN *condition* THEN *result*
    [WHEN ...]
    [ELSE *result*]
END

```
db_personal=> SELECT "txt_NamaDepan", "txt_NamaAkhir","dt_TglLahir", CASE WHEN "
dt_TglLahir" < '1980-01-01' THEN 'Dewasa' ELSE 'Remaja' END AS Umur from tbl_Per
sonal;
 txt_NamaDepan | txt_NamaAkhir | dt_TglLahir |  umur
---------------+---------------+-------------+--------
 andhie        | lala          | 1960-08-08  | Dewasa
 ade           | fajar         | 1980-11-01  | Remaja
 panuju        | sasongko      | 1970-09-12  | Dewasa
 dudy          | rudianto      | 1973-12-11  | Dewasa
 ana           | hidayati      | 1983-10-01  | Remaja
 rofiq         | yuliardi      | 1999-07-14  | Remaja
(6 rows)
```

Contoh berikut ini adalah penggunaan CASE dalam bentuk lain, yang mana tingkat kompleksitasnya lebih banyak:

```
db_personal=> SELECT "txt_NamaDepan", "txt_NamaAkhir","dt_TglLahir", CASE WHEN "
dt_TglLahir" < '1980-01-01' THEN 'Dewasa' ELSE 'Remaja' END AS Usia, CASE WHEN "
bol_IsNikah" ='f' THEN 'Belum' ELSE 'Sudah' END AS Status from tbl_Personal;
 txt_NamaDepan | txt_NamaAkhir | dt_TglLahir |  usia  | status
---------------+---------------+-------------+--------+--------
 andhie        | lala          | 1960-08-08  | Dewasa | Belum
 ade           | fajar         | 1980-11-01  | Remaja | Belum
 panuju        | sasongko      | 1970-09-12  | Dewasa | Belum
 dudy          | rudianto      | 1973-12-11  | Dewasa | Belum
 ana           | hidayati      | 1983-10-01  | Remaja | Belum
 rofiq         | yuliardi      | 1999-07-14  | Remaja | Sudah
(6 rows)
```

DISTINCT

Sebuah pengertian yang sederhana dari DISTINCT adalah untuk mencegah terjadinya duplikasi pada output sebuah tabel. DISTINCT sering kali diperlukan untuk mengembalikan hasil dari sebuah *query* dengan tidak terdapat duplikasi, artinya pada hasil outputnya tidak terjadi kesamaan data meskipun pada data sesungguhnya sangat mungkin banyak duplikasi.

```
db_personal=> select  "txt_Pekerjaan" from tbl_Personal;
 txt_Pekerjaan
----------------
 Programmer
 Programmer
 System Analyst
 Konsultan
 Konsultan
 Programmer
(6 rows)
```

Setelah di lakukan DISTINCT maka hanya akan dihasilkan tiga (3) row karena Programmer yang sebenarnya 3 row hanya akan ditampilkan sekali saja.

```
db_personal=> select DISTINCT "txt_Pekerjaan" from tbl_Personal;
```

```
 txt_Pekerjaan
----------------
 Konsultan
 Programmer
 System Analyst
(3 rows)
```

SET, SHOW dan RESET

Perintah SET digunakan untuk melakukan perubahan parameter PostgreSQL, pengantian dengan menggunakan perintah SET hanya berlaku untuk sekali *session* saja dimana perintah SET dilakukan. Misalnya perintah SET DATESTYLE bertugas untuk mengontrol penampilan sebuah data ketika terlihat dalam *psql*.

Function	Opsi SET
DATESTYLE	DATESTYLE TO ' I 'SO' | 'POSTGRES | 'SQL' | 'US' | 'NONEUROPEAN' | 'EUROPEAN' | 'GERMAN'
TIMEZONE	TIMEZONE TO *'value'*

Model	Opsi	Output untuk February 1, 1983
ISO		1983 - 02 - 01
POSTGRES	US or NonEUROPEAN	02-01-1983
POSTGRES	EUROPEAN	01-02-1983
SQL	US or NonEUROPEAN	02/01/1983
SQL	EUROPEAN	01/02/1983
GERMAN		01.02.1983

Pada masing-masing *server database* PostgreSQL sudah terdapat default *time zone*, sedangkan pada *psql client*, *time zone*-nya mungkin berbeda, untuk itu kita dapat mengesetnya dengan mengikuti parameter yang telah ditetapkan. Perintah SHOW dapat digunakan untuk melihat atau menampilkan parameter yang aktif. Sedangkan perintah RESET mengijinkan sebuah session parameter untuk dikembalikan pada posisi nilai defaultnya, berikut contohnya:

```
db_personal=> SHOW DATESTYLE;
NOTICE:  DateStyle is ISO with US (NonEuropean) conventions
SHOW VARIABLE

db_personal=> SET DATESTYLE TO 'SQL, EUROPEAN';
SET VARIABLE

db_personal=> SHOW DATESTYLE;
NOTICE:  DateStyle is SQL with European conventions
SHOW VARIABLE

db_personal=> RESET DATESTYLE;
RESET VARIABLE

db_personal=> SHOW DATESTYLE;
NOTICE:  DateStyle is ISO with US (NonEuropean) conventions
SHOW VARIABLE
```

```
db_personal=> show TIMEZONE;
   TimeZone
--------------
 Asia/Bangkok

db_personal=> show all;
              name               |    setting     |
            description
---------------------------------+----------------+----------------------------
-------------------------------------------------------------------
 add_missing_from                | off            | Automatically adds missing t
able references to FROM clauses.
 archive_command                 | unset          | WAL archiving command.
 australian_timezones            | off            | Interprets ACST, CST, EST, a
nd SAT as Australian time zones.
 authentication_timeout          | 60             | Sets the maximum time in sec
onds to complete client authentication.
```

LIMIT

LIMIT dan OFFSET digunakan untuk membatasi jumlah output dari query berdasarkan jumlah row bukan karena kondisi seperti WHERE. Sebagai contoh misalnya tabel *tbl_personal* memiliki id sebanyak sepuluh yaitu mulai dari 1 s/d 10 secara berturut, sekarang kita akan menggunakan perintah LIMIT dan OFFSET untuk menampilkan id tersebut secara spesifik sesuai keinginan. Bisa dikatakan LIMIT adalah untuk menentukan jumlah baris yang akan ditampilkan yang dihitung dari baris pertama, sedangkan OFFSET digunakan untuk menghilangkan row sesuai dengan jumlah yang diberikan pada OFFSET.
Contoh berikut adalah untuk menampilkan dua baris pertama dengan LIMIT,

```
db_personal=#  select * from tbl_personal ORDER BY int_ID LIMIT 2 ;
 int_id | txt_NamaDepan | txt_NamaAkhir | dt_TglLahir | bol_IsNikah | txt_Pekerj
aan
--------+---------------+---------------+-------------+-------------+-----------
----
      1 | rofiq         | yuliardi      | 1999-07-14  | t           | Programmer
      2 | andhie        | lala          | 1960-08-08  | f           | Programmer
(2 rows)
```

Contoh penggunaan OFFSET untuk menampilkan data setelah baris ke-3.

```
db_personal=# select * from tbl_personal ORDER BY int_ID OFFSET 3;
 int_id | txt_NamaDepan | txt_NamaAkhir | dt_TglLahir | bol_IsNikah | txt_Pekerj
aan
--------+---------------+---------------+-------------+-------------+-----------
-----
      4 | panuju        | sasongko      | 1970-09-12  | f           | System Ana
lyst
      5 | ana           | hidayati      | 1983-10-01  | f           | Konsultan
      5 | dudy          | rudianto      | 1973-12-11  | f           | Konsultan
(3 rows)
```

Contoh penggunaan gabungan LIMIT dan OFFSET, digunakan untuk menampilkan data dengan LIMIT 2 row setelah dilakukan OFFSET 3 row.

```
db_personal=# select * from tbl_personal ORDER BY int_ID LIMIT 2 OFFSET 3;
 int_id | txt_NamaDepan | txt_NamaAkhir | dt_TglLahir | bol_IsNikah | txt_Pekerj
aan
--------+---------------+---------------+-------------+-------------+-----------
-----
      4 | panuju        | sasongko      | 1970-09-12  | f           | System Ana
lyst
```

```
     5 | ana           | hidayati      | 1983-10-01  | f           | Konsultan
(2 rows)
```

UNION, EXCEPT, dan INTERSECT

Hasil dari dua buah query dapat dikombinasikan dengan menggunakan UNION, EXCEPT atau INTERSECT. UNION digunakan untuk menggabungkan hasil dua buah query menjadi satu kolom. Berikut salah satu contohnya, misalkan kita menginginkan output dari nama depan dan nama belakang menjadi satu kolom.

```
db_personal=> select "txt_NamaDepan" From tbl_Personal UNION Select "txt_NamaAkh
ir" from tbl_personal as Nama;
 txt_NamaDepan
---------------
 ade
 ana
 andhie
 dudy
 fajar
 hidayati
 lala
 panuju
 rofiq
 rudianto
 sasongko
 yuliardi
(12 rows)

db_personal=> select "txt_NamaDepan" From tbl_Personal UNION Select "txt_Pekerja
an" from tbl_personal as Nama;
 txt_NamaDepan
----------------
 Konsultan
 Programmer
 System Analyst
 ade
 ana
 andhie
 dudy
 panuju
 rofiq
(9 rows)
```

Contoh berikutnya misalkan kita menginginkan dua buah tabel menangani berbagai macam jenis penyakit, satu tabel menangani tentang *Penyakit Dalam* dan tabel yang satunya berisikan data tentang *Penyakit Menular*. Dua buah tabel tersebut dapat digunakan karena beberapa dari *record* informasinya memiliki kesamaan jenis penyakitnya. Berikut contohnya (contoh di bawah ini diasumsikan kita telah memiliki duah buah tabel yaitu *penyakit_dalam* dan *penyakit menular)* kemudian masukkan beberapa data penyakit seperti di bawah ini.

```
db_personal=> INSERT INTO Penyakit_dalam (nama) Values (
db_personal(> 'Jantung' );
INSERT 36068 1

db_personal=> INSERT INTO Penyakit_dalam (nama) Values (
db_personal(> 'paru-paru' );
INSERT 36069 1

db_personal=> Select * FROM Penyakit_dalam;
         nama
----------------------
```

```
 Jantung
 paru-paru
(2 rows)
```

masukkan data di bawah ini ke tabel *penyakit_menular.*

```
db_personal=> INSERT INTO Penyakit_menular (nama) Values (
db_personal(> 'TBC' );
INSERT 36070 1

db_personal=> INSERT INTO Penyakit_dalam (nama) Values (
db_personal(> 'Demamberdarah' );
INSERT 36071 1
db_personal=> INSERT INTO Penyakit_dalam (nama) Values (
db_personal(> 'Malaria' );
INSERT 36072 1

db_personal=> SELECT * FROM Penyakit_menular;
         nama
----------------------
 TBC
 Demamberdarah
 Malaria
(3 rows)
```

Sekarang jalankan kedua tabel tersebut dengan cara mengkombinasikan dua buah tabel penyakit tadi dengan perintah UNION, berikut contohnya:

```
db_personal=> SELECT nama
db_personal-> FROM Penyakit_dalam
db_personal-> UNION
db_personal-> SELECT nama
db_personal-> FROM Penyakit_menular;
         nama
----------------------
 Demamberdarah
 Jantung
 Malaria
 paru-paru
 TBC
(5 rows)
```

Defaultnyta UNION hanya akan menampilkan satu data saja sehingga tidak akan terjadi duplikat *row* dari hasil output query-nya. Sebagai contoh db_personal kita INSERT lagi sebuah data pada masing-masing tabel penyakit, nama penyakitnya adalah Migrand. Kemudian lakukan SELECT dengan menggunakan UNION, berikut contohnya

```
db_personal=> INSERT INTO Penyakit_dalam (nama) VALUES ('Migrand');
INSERT 36073 1

db_personal=> INSERT INTO Penyakit_menular (nama) VALUES ('Migrand');
INSERT 36074 1

db_personal=> SELECT nama
db_personal-> FROM Penyakit_dalam
db_personal-> UNION
db_personal-> SELECT nama
db_personal-> FROM Penyakit_menular;
         nama
----------------------
 Demamberdarah
 Jantung
 Malaria
 Migrand
 paru-paru
```

```
 TBC
(6 rows)
```

Untuk dapat melihat atau menampilkan semua duplikat yang ada pada tabel-tabel tersebut tadi maka gunakan perintah UNION ALL.

```
db_personal=> SELECT nama
db_personal-> FROM Penyakit_dalam
db_personal-> UNION ALL
db_personal-> SELECT nama
db_personal-> FROM Penyakit_menular;
         nama
----------------------
 Jantung
 paru-paru
 TBC
 Migrand
 Demamberdarah
 Malaria
 Migrand
(7 rows)
```

EXCEPT digunakan untuk menampilkan hanya query pertama saja, sedangkan hasil query kedua tidak akan ditampilkan.

```
db_personal=> SELECT nama
db_personal-> FROM Penyakit_dalam
db_personal-> EXCEPT
db_personal-> SELECT nama
db_personal-> FROM Penyakit_menular;
         nama
----------------------
 Jantung
 paru-paru
(2 rows)

db_personal=> SELECT nama
db_personal-> FROM Penyakit_menular
db_personal-> EXCEPT
db_personal-> SELECT nama
db_personal-> FROM Penyakit_dalam;
         nama
----------------------
 Demamberdarah
 Malaria
 TBC
(3 rows)
```

Pada output diatas jenis penyakit *Migrand* tidak ditampilkan, ini dikarenakan adanya duplikat *Migrand* atau nama penyakit yang sama. Namun dia akan muncul jika kita menggunakan perintah INTERSECT. Perintah ini hanya akan menampilkan seluruh isi dari data yang memiliki kesaman diantara hasil kedua query tersebut.

```
db_personal=> SELECT nama
db_personal-> FROM Penyakit_dalam
db_personal-> INTERSECT
db_personal-> SELECT nama
db_personal-> FROM Penyakit_menular;
         nama
----------------------
 Migrand
(1 row)
```

Dari rangkaian SELECT di atas dapat menimbulkan sebuah operasi yang bagus, seperti penggabungan sebuah kolom menjadi sebuah tabel pada SELECT pertama kemudian penggabungan kolom yang sama ke tabel lain pada SELECT kedua.
Berikut adalah contoh *query* dari penggabungan nama dari tiga buah tabel menjadi sebuah tabel dalam satu kolom:

```
db_personal=> SELECT nama
db_personal-> FROM Penyakit_dalam
db_personal-> UNION
db_personal-> SELECT nama
db_personal-> FROM Penyakit_menular
db_personal-> UNION
db_personal-> SELECT nama
db_personal-> FROM identitas;
         nama
----------------
 Demamberdarah
 Esna
 Iin
 Jantung
 Malaria
 Migrand
 Nidha
 paru-paru
 Restu
 Wati
 TBC
(11 rows)
```

Berikut penggabungan nama dari empat buah tabel menjadi satu tabel.

```
db_personal=> SELECT nama
db_personal-> FROM Penyakit_dalam
db_personal-> UNION
db_personal-> SELECT nama
db_personal-> FROM Penyakit_menular
db_personal-> UNION
db_personal-> SELECT nama
db_personal-> FROM identitas
db_personal-> UNION
db_personal-> SELECT marga
db_personal-> FROM identitas;
         nama
----------------------
 Adam
 Asti
 Demamberdarah
 Esna
 Iin
 Jantung
 Malaria
 Meilan
 Migrand
 Nidha
 paru-paru
 Priti
 Restu
 Wati
 TBC
 Utomo
(16 rows)
```

AGREGASI SQL

Bab 8 – Agregasi SQL

Deskripsi

Bab ini akan membahas agregasi SQL yang diperlukan untuk melakukan operasi perhitungan pada data.

Obyektif

Pada bab ini diharapkan peserta dapat memahami dan menggunakan agregasi SQl yang berhubungan dengan manipulasi data pada tabel.

Outline

- Aggregate
- GROUP BY
- HAVING

Fungsi aggregate atau disebut fungsi ringkasan digunakan untuk melakukan penghitungan menjadi sebuah nilai dari beberapa nilai input.

Aggregate

Aggregate dapat digabungkan dengan sebuah parameter seperti WHERE untuk menghasilkan suatu hasil yang lebih kompleks lagi. Nilai NULL tidak diproses oleh sebagian besar Aggregate, seperti MAX(), SUM(), dan AVG(). Adapun fungsi agregate yang disediakan oleh PostgreSQL dapat dilihat pada tabel berikut;

Aggregate	Keterangan
COUNT(*)	Menghitung jumlah row
SUM(nama_kolom)	Menghitung penjumlahan data
MAX(nama_kolom)	Menghasilkan nilai terbesar
MIN(nama_kolom)	Menghasilkan nilai terkecil
AVG(nama_kolom)	Menghasilkan nilai rata-rata

Berikut contoh dari beberapa *aggregates query*:

```
db_personal=> select * from tbl_Personal;
 int_id | txt_NamaDepan | txt_NamaAkhir | dt_TglLahir | bol_IsNikah | txt_Pekerj
aan
--------+---------------+---------------+-------------+-------------+-----------
-----
      2 | andhie        | lala          | 1960-08-08  | f           | Programmer
      3 | ade           | fajar         | 1980-11-01  | f           | Programmer
      4 | panuju        | sasongko      | 1970-09-12  | f           | System Ana
lyst
      5 | dudy          | rudianto      | 1973-12-11  | f           | Konsultan
      5 | ana           | hidayati      | 1983-10-01  | f           | Konsultan
      1 | rofiq         | yuliardi      | 1999-07-14  | t           | Programmer
(6 rows)

db_personal=> select count(*) from tbl_Personal;
 count
-------
     6
(1 row)

db_personal=> select max(int_ID) from tbl_Personal;
 max
-----
   5
(1 row)

db_personal=> select min(int_ID) from tbl_Personal;
 min
-----
   1
(1 row)

db_personal=> select avg(int_ID) from tbl_Personal;
        avg
--------------------
 3.3333333333333333
(1 row)
```

Seperti yang telah dibahas pada pembahasan *aggregate* di atas bahwa SQL *aggregate* dapat digunakan pada nilai NULL (bukan nol (0)) dan non-NULL, berikut contoh *query* dari keduanya.

```
db_personal=> CREATE TABLE test (col INTEGER);
CREATE

db_personal=> INSERT INTO test VALUES (NULL);
INSERT 36087 1

db_personal=> SELECT SUM(col) FROM test;
 sum
---------

(1 row)

db_personal=> SELECT MAX(col) FROM test;
 max
-----

(1 row)

db_personal=> SELECT COUNT(*) FROM test;
 count
-------------
     1
(1 row)

db_personal=> SELECT COUNT(col) FROM test;
 count
-----------
     0
(1 row)

db_personal=> INSERT INTO test VALUES (3);
INSERT 36088 1

db_personal=> SELECT AVG(col) FROM test;
     avg
--------
       3
(1 row)

db_personal=> SELECT COUNT(*) FROM test;
 count
----------
     2
(1 row)

db_personal=> SELECT COUNT(col) FROM test;
 count
-----------
     1
(1 row)
```

jika kita ingin mencari rata-rata (Average) secara spesifik lagi maka gunakan perintah *WHERE* agar dapat melihat hasilnya secara spesifik lagi. Berikut contoh penggunaan *avarage* dengan spesifik *query* pada tabel *nama_kolom*:

```
 db_personal=> SELECT AVG(colname) FROM tes
 db_personal-> WHERE colname >= 6;
     avg
----------------------
 8.1666666667
(1 row)

 db_personal=> SELECT AVG(colname) FROM tes
```

```
 db_personal-> WHERE colname <= 6;
     avg
-------------
     2.4
(1 row)
```

GROUP BY

Group By merupakan fungsi yang digunakan untuk melakukan pengelompokan dari perintah SELECT. Group by seringkali diperlukan untuk menjalankan *aggregate* menjadi sebuah kelompok dari hasil Query. Misalkan kita ingin melihat jumlah total dari *row* dalam sebuah tabel, ini dapat dilakukan dengan menggunakan GROUP BY. Misalnya untuk memperoleh atau mengetahui perhitungan dan jumlah dari banyaknya orang berprofesi sebagai programmer, Konsultan dan System Analyst. Berikut contoh *query*-nya

```
db_personal=> select "txt_Pekerjaan", Count(*) From tbl_Personal Group By "txt_P
ekerjaan";
 txt_Pekerjaan  | count
----------------+-------
 Konsultan      |     2
 System Analyst |     1
 Programmer     |     3
(3 rows)


db_personal=> select CASE WHEN "bol_IsNikah" ='f' THEN 'BELUM' ELSE 'SUDAH' END
 AS STATUS, Count(*) From tbl_Personal Group By "bol_IsNikah";
 status | count
--------+-------
 BELUM  |     5
 SUDAH  |     1
(2 rows)
```

Berikut adalah beberapa contoh *query* untuk menampilkan minimum dan maksimum tanggal lahir dari nama-nama yang terdapat dalam tabel tersebut berdasarkan pekerjaanya. Berikut ini contoh *query*-nya:

```
db_personal=> select "txt_Pekerjaan", min("dt_TglLahir"), max("dt_TglLahir") fro
m tbl_personal Group By "txt_Pekerjaan";
 Programmer     | 1960-08-08 | 1999-07-14
 System Analyst | 1970-09-12 | 1970-09-12
 Konsultan      | 1973-12-11 | 1983-10-01
```

GROUP BY dapat digunakan lebih dari satu kolom, ini dikarenakan GROUP BY dapat mengumpulkan semua nilai NULL dalam sebuah *group*, berikut contoh *query*-nya :

```
db_personal=> select "txt_Pekerjaan","bol_IsNikah", min("dt_TglLahir"), max("dt_
TglLahir") from tbl_personal Group By "txt_Pekerjaan", "bol_IsNikah";
 Konsultan      | f           | 1973-12-11 | 1983-10-01
 System Analyst | f           | 1970-09-12 | 1970-09-12
 Programmer     | t           | 1999-07-14 | 1999-07-14
 Programmer     | f           | 1960-08-08 | 1980-11-01
```

HAVING

Pemakaian HAVING terkait dengan GROUP BY, kegunaanya adalah untuk menentukan kondisi bagi GROUP BY, dimana kelompok yang memenuhi kondisi saja yang akan di hasilkan. Sebagai contoh misalnya kita ingin mengetahui semua profesi yang jumlah orangnya lebih dari 3. Dari hasil *query* di bawah ini dengan menggunakan HAVING maka

akan terlihat dengan jelas bahwa pekerjaan apa yang memiliki lebih dari tiga orang (artinya pekerjaan tersebut memiliki dua nama yang sama), berikut contohnya:

```
db_personal=> select "txt_Pekerjaan", Count(*) from tbl_personal GROUP BY "txt_P
ekerjaan" HAVING Count(*) > 2;
 Programmer    |     3


db_personal=> SELECT * FROM identitas;
```

Pada contoh di bawah ini hasilnya kosong dikarenakan tidak ada pekerjaan yang memiliki nama lebih dari tiga buah nama (HAVING COUNT(*) > 3):

```
db_personal=> select "txt_Pekerjaan", Count(*) from tbl_personal GROUP BY "txt_P
ekerjaan" HAVING Count(*) > 3;
 txt_Pekerjaan | count
------ -+-------
(0 rows)
```

VIEW dan RULE

Bab 9 – View dan Rule

Deskripsi

Bab ini akan membahas tentang View, view merupakan sebuah tabel bayangan yang berisi query baik dari satu tabel maupun berbagai tabel. Bab ini juga akan membahas tentang RULE yang spesifik digunakan untuk View.

Obyektif

Pada bab ini diharapkan peserta dapat memahami dan menggunakan VIEW serta RULE dalam PostgreSQL yang berhubungan dengan manipulasi data pada tabel.

Outline

- VIEW
- RULE

Pada umumnya sebagian besar SQL (Structured Query Lenguage) dipakai sebagai jalan komunikasi dengan database server, SQL itu sendiri didukung oleh hampir semua sistem database dan menjadi standar untuk mengakses database relasional.

VIEW

Views dapat juga disebut tabel bayangan tetapi buka *temporary table*, bukan juga merupakan sebuah tabel yang asli. Tetapi meskipun demikian di dalam PostgreSQL*view* dapat juga di SELECT, INSERT, UPDATE atau DELETE dengan menggunakan atau menerapkan *rule*. Satu lagi kelebihan yang dimiliki oleh view yaitu dapat menyimpan perintah *query*, dan dapat mewakili sebuah subset dari tabel asli dan memilih kolom atau *row* tertentu dari tabel biasa. View juga dapat memainkan perannya pada penggabungan tabel dikarenakan view dapat memberikan ijin salinannya, kita dapat menggunakan view untuk membatasi atau melarang pengaksessan atas sebuah tabel, jadi sebuah *user* hanya dapat melihat kolom atau *row* secara spesifik dari sebuah tabel. View dapat dibuat dengan menggunakan perintah CREATE VIEW, lihat contoh berikut ini.

```
db_personal=> CREATE VIEW vw_getJob AS
db_personal-> SELECT * from tbl_personal
db_personal-> Where "txt_Pekerjaan" = 'Programmer';
CREATE VIEW
```

View diakses dengan menggunakan query SELECT seperti tabel, dan bisa di beri parameter maupun tanpa parameter.

```
db_personal=> select * from vw_getJob;
 int_id | txt_NamaDepan | txt_NamaAkhir | dt_TglLahir | bol_IsNikah | txt_Pekerj
aan
--------+---------------+---------------+-------------+-------------+-----------
----
      2 | andhie        | lala          | 1960-08-08  | f           | Programmer
      3 | ade           | fajar         | 1980-11-01  | f           | Programmer
      1 | rofiq         | yuliardi      | 1999-07-14  | t           | Programmer
(3 rows)
```

Contoh berikut ini adalah untuk menampilkan hanya kolom tertentu dari View.

```
db_personal=> select "txt_NamaDepan" from vw_getJob;
 txt_NamaDepan
---------------
 andhie
 ade
 rofiq
(3 rows)
```

Untuk menghapus view gunakan DROP VIEW, view yang dihapus sama sekali tidak mempengaruhi tabel aslinya sebab view bukanlah tabel yang sebenarnya. Misalkan kita telah membuat sebuah view dari tabel tbl_personal dengan nama vw_getJob sehingga dapat dikatakan vw_getJob merupakan duplikat dari tabel identitas dengan parameter txt_Pekerjaan, karena dalam pembuatan view dapat juga ditentukan spesifikasi sebuah kolom.

RULE

Rule berfungsi untuk merubah atau meng-*update* sebuah tabel, hanya dengan melekatkan

rule tersebut pada suatu tabel maka sifat dari tabel tadi dapat diubah. VIEW dalam PostgreSQL sebenarnya telah memiliki *SELECT RULE*. Rule memiliki sebuah *keyword* yang berfungsi untuk mencegah terjadinya penambahan suatu data ke dalam sebuah tabel, *keyword* yang dimaksud adalah NOTHING, berikut contohnya:

```
db_personal=>

 db_personal=> CREATE TABLE tesrule (nama text) ;
CREATE
 db_personal=> SELECT * FROM tesrule;
 nama
------
(0 rows)

 db_personal=> CREATE RULE tes_insert AS  -- nama rule
 db_personal-> ON INSERT TO tesrule       -- Insert rule
 db_personal-> DO INSTEAD         -- type rule
 db_personal->    NOTHING ;               -- keyword
CREATE
```

Kemudian masukkan sebuah data ke dalam tabel *tesrule* kemudian tampilkan tabel-nya

```
 db_personal=> INSERT INTO tesrule VALUES ('andhie') ;
 db_personal=> SELECT * FROM tesrule;
 nama
------
(0 rows)
```

dengan menggunakan RULE kita dapat memanfaatkan berbagai macam default yang telah tersedia pada PostgreSQL, seperti timestamp dan current *user*. Berikut contoh yang mengimplementasikan dafault tersebut, misalkan kita membuat dua buah tabel barang berisikan atribut yang sama namun berbeda fungsinya. Nama tabel tersebut adalah *barang* dan *barang_log.* Fungsi dari tabel *barang* untuk menampung data yang masuk sedangkan tabel *barang_log* berfungsi untuk menampung data hasil UPDATE atau DELETE dari tabel *barang,* artinya jika sebuah data pada tabel *barang* di-*update* ataupun dihapus maka data tersebut tidak hilang melainkan secara otomatis akan tersimpan pada tabel *barang_log.*

Sebagai contoh buatlah dua buah tabel dengan atribut yang sama

```
 db_personal=> CREATE TABLE barang (
 db_personal(>            id_barang  INTEGER,
 db_personal(>          nama_barang  text,
 db_personal(>          kasir text DEFAULT CURRENT_USER,
 db_personal(>          tanggal timestamp DEFAULT CURRENT_TIMESTAMP);
CREATE

 db_personal=> CREATE TABLE barang_log (
 db_personal(>            id_barang  INTEGER,
 db_personal(>          nama_barang  text,
 db_personal(>          kasir text DEFAULT CURRENT_USER,
 db_personal(>          tanggal timestamp DEFAULT CURRENT_TIMESTAMP);
CREATE
```

Kemudian buat RULE update dan delete dari tabel barang

```
 db_personal=> CREATE RULE update_barang AS        -- UPDATE  rule
 db_personal-> ON UPDATE TO barang
 db_personal-> DO
 db_personal->    INSERT INTO barang_log (id_barang, nama_barang)
```

```
 db_personal->    VALUES (old.id_barang, old.nama_barang);
CREATE

 db_personal=> CREATE RULE delete_barang AS        -- DELETE rule
 db_personal-> ON DELETE TO barang
 db_personal-> DO
 db_personal->    INSERT INTO barang_log (id_barang, nama_barang)
 db_personal->    VALUES (old.id_barang, old.nama_barang);
CREATE
```

masukkan beberapa data pada tabel barang, khusus untuk attribut *kasir* dan *tanggal* tidak perlu diisi sebab akan terisi secara otomatis.

```
 db_personal=> INSERT INTO barang (id_barang, nama_barang)
 db_personal-> VALUES (270281, 'Kado ultah') ;
INSERT 44715 1

 db_personal=> INSERT INTO barang (id_barang, nama_barang)
 db_personal-> VALUES (812702, 'Komputer') ;
INSERT 44716 1

 db_personal=> INSERT INTO barang (id_barang, nama_barang)
 db_personal-> VALUES (832612, 'Buku Postgresql');
INSERT 44717 1

 db_personal=> SELECT * FROM barang;
 id_barang  |   nama_barang          |  kasir   |        tanggal
------ --+--------------+--- --+---------------------
    270281   |   Kado Ultah           | andhi  | 2002-03-08 12:06:03+07
    812702   |   Komputer     | andhi  | 2002-03-08 12:06:38+07
    832612   |   Buku Postgresql  | andhi  | 2002-03-08 12:07:27+07
(3 rows)

 db_personal=> SELECT * FROM barang_log;
 id_barang  |   nama_barang       |  kasir   |        tanggal
------ --+--------- ---+--- --+----------
(0 rows)
```

Sekarang cobalah update salah satu data pada tabel *barang*

```
 db_personal=> UPDATE barang
 db_personal-> SET nama_barang = 'Coklat Silverquein'
 db_personal-> WHERE id_barang = 270281 ;
UPDATE 1

 db_personal=> SELECT * FROM barang;
 id_barang  |  nama_barang    |  kasir   |        tanggal
------ --+--------------+--- --+---------------------
    812702   | Komputer       | andhi   | 2002-03-08 12:06:38+07
    832612   | Buku Postgresql    | andhi   | 2002-03-08 12:07:27+07
    270281   | Coklat Silverquein | andhi   | 2002-03-08 12:06:03+07
(3 rows)

 db_personal=> SELECT * FROM barang_log;
 id_barang  |  nama_barang|  kasir   |        tanggal
------ --+--------- ---+--- --+-------------------
    270281   | Kado Ultah |  andhi  | 2002-03-08 12:06:03+07
(1 rows)
```

Coba hapus salah satu data dari tabel *barang*

```
 db_personal=> DELETE FROM barang
 db_personal-> WHERE id_barang = 270281 ;
DELETE 1
```

```
 db_personal=> SELECT * FROM barang;
 id_barang  |   nama_barang         |  kasir   |        tanggal
------ --+---------- ---+--- --+---------------------
    812702   |   Komputer            | andhi  | 2002-03-08 12:06:38+07
    832612   |   Buku Postgresql | andhi  | 2002-03-08 12:07:27+07
(2 rows)

 db_personal=> SELECT * FROM barang_log;
 id_barang |   nama_barang   |  kasir   |        tanggal
--------+-------------+--- --+---------------------
    270281  | Kado Ultah      |  andhi  | 2002-03-08 12:06:03+07
    270281  | Coklat Silverquein | andhi   | 2002-03-08 12:06:03+07
(2 rows)
```

View bukan sebuah tabel yang asli namun dia hanya merupakan duplikat dari sebuah tabel, view dapat di-*select* layaknya sebuah tabel namun untuk meng-*update* ataupun menghapus data yang terdapat di dalamnya haruslah menggunakan RULE. Jadi dengan cara melekatkan *rule* pada tabel view tadi maka proses *update* dan *delete* dapat dilakukan sebgaimana mestinya, berikut contohnya.

```
 db_personal=> CREATE TABLE tes_view (
 db_personal(>             No  INTEGER,
 db_personal(>             Mata_Kuliah  text,
 db_personal(>             Kode_MK text);
CREATE
```

Buatlah view dari tabel tes_view dengan nama tes_v, setelah itu masukkan beberapa data pada tabel tes_view

```
 db_personal=> CREATE VIEW  tes_v AS
 db_personal-> SELECT * FROM tes_view;
CREATE

 db_personal=> INSERT INTO tes_view VALUES (
 db_personal(>  1, 'Database Postgresql', 'DP 001');
INSERT 52889 1

 db_personal=> INSERT INTO tes_view VALUES (
 db_personal(>  2, 'Star Database ', 'SD 002');
INSERT 52890 1

 db_personal=> SELECT * FROM tes_view;
 no  |     mata_kuliah    | kode_mk
---+-------------- --+--------
  1   | Database Postgresql       | DP 001
  2   | Star Database     | SD 002
(2 rows)

 db_personal=> SELECT * FROM tes_v;
 no  |     mata_kuliah           | kode_mk
---+--------------------+--------
  1   | Database Postgresql             | DP 001
  2   | Star Database            | SD 002
(2 rows)
```

Untuk memodifikasi sebuah view terlebih dahulu kita harus membuat RULE *insert*, *update* maupun *delete* pada view yang dimaksud

```
 db_personal=> CREATE TABLE vitamin (nama text);
CREATE
 db_personal=> CREATE VIEW vitamin_view AS
```

```
 db_personal-> SELECT * FROM vitamin ;
CREATE

 db_personal=> \z vitamin
Access permissions for database "satu"
   Relation        | Access permissions
-----------+-----------------
  vitamin           |
  vitamin_view  |
(2 rows)

 db_personal=> INSERT INTO vitamin VALUES (
 db_personal(> 'vitacimin');
INSERT 52935 1
 db_personal=> INSERT INTO vitamin VALUES (
 db_personal(> 'B 12');
INSERT 52936 1

 db_personal=> SELECT * FROM vitamin;
   nama
-----------
 vitacimin
 B 12
(2 rows)

 db_personal=> SELECT * FROM vitamin_view;
   nama
-----------
 vitacimin
 B 12
(2 rows)
```

sebuah pesan kegagalan akan muncul ketika menginsert data pada sebuah VIEW tanpa membuat RULE terlebih dahulu

```
 db_personal=> INSERT INTO vitamin_view VALUES ('vit B');
ERROR:  Cannot insert into a view without an appropriate rule
```

berikut ini *query* untuk membuat RULE *insert*, *update*, *delete* dari view *vitamin_view*

```
 db_personal=> CREATE RULE vitamin_view_insert AS       -- INSERT rule
 db_personal-> ON INSERT TO vitamin_view
 db_personal-> DO INSTEAD
 db_personal->    INSERT INTO vitamin
 db_personal->    VALUES (new.nama);
CREATE

 db_personal=> CREATE RULE vitamin_view_update AS         -- UPDATE rule
 db_personal-> ON UPDATE TO vitamin_view
 db_personal-> DO INSTEAD
 db_personal->    UPDATE vitamin
 db_personal->    SET nama = new.nama
 db_personal->    WHERE nama = old.nama;
CREATE

 db_personal=> CREATE RULE vitamin_view_delete AS         -- DELETE rule
 db_personal-> ON DELETE TO vitamin_view
 db_personal-> DO INSTEAD
 db_personal->    DELETE FROM vitamin
 db_personal->    WHERE nama = old.nama;
CREATE
```

Setelah pembuatan RULE selesai, maka view tersebut sekarang dapat dimodifikasi sesuai

dengan keinginan kita.

```
 db_personal=> INSERT INTO vitamin_view
 db_personal-> VALUES ('Fatigon');
INSERT 52943 1

 db_personal=> INSERT INTO vitamin_view
 db_personal-> VALUES ('Cratindaeng');
INSERT 52944 1

 db_personal=> SELECT * FROM vitamin_view;
    nama
-------------
 vitacimin
 B 12
 Fatigon
 Cratindaeng
(4 rows)

 db_personal=> UPDATE vitamin_view
 db_personal-> SET nama = 'FitUp'
 db_personal-> WHERE nama = 'Cratindaeng' ;
UPDATE 1

 db_personal=> SELECT * FROM vitamin_view;
   nama
-----------
 vitacimin
 B 12
 Fatigon
 FitUp
(4 rows)

 db_personal=> DELETE FROM vitamin_view
 db_personal-> WHERE nama = 'vitacimin' ;
DELETE 1

 db_personal=> SELECT * FROM vitamin_view;
  nama
---------
 B 12
 Fatigon
 FitUp
(3 rows)
```

INDEKS

Bab 10 – Indeks

Deskripsi

Bab ini akan membahas tentang Indeks dalam database PostgreSQL, bab ini juga akan membahas dalam penggunakan peimary key dan foreign key.

Obyektif

Pada bab ini diharapkan peserta dapat memahami dan menggunakan indeks dalam pengelolaan database PostgreSQL. Peserta juga dapat memahami konsep dasar primary key dan foreign key.

Outline

- Indeks Unik
- Kolom Unik
- CLUSTER
- Primary Key
- Foreign Key
- Integritas Referensial
- Check

Ketika mengakses sebuah tabel biasanya PostgreSQL akan membaca seluruh tabel baris per baris sampai selesai. Ketika jumlah row sangat banyak sedangkan hasil dari query hanya sedikit, maka hal tersebut sangat tidak efisien. Seperti halnya ketika kita membaca sebuah buku, dan ingin mencari kata atau istilah tertentu dalam buku maka biasanya akan di cari dengan membuka setiap halaman dari awal sampai akhir. Dengan adanya indeks pada buku maka kita cukup membuka indeks, sehingga dengan cepat bisa mengetahui posisi halaman dimana kata yang dicari berada.
Dalam database juga demikian karena dengan indeks dapat dengan cepat dan spesifik menemukan nilai dalam indeks, dan langsung mencocokkan dengan *row* yang dimaksud. Sebagai contoh, perhatikan *query* berikut ini SELECT * FROM customer WHERE col = 26. Tanpa indeks, PostgreSQL harus mengamati atau meninjau seluruh tabel untuk mencari di mana row *col* yang sama dengan 26. Jika menggunakan indeks maka PostgreSQLakan langsung menuju ke *row* yang sama dengan 26. Untuk sebuah tabel yang besar dan luas, PostgreSQLdapat mengecek setiap *row* dalam menit sedangkan jika menggunakan indeks untuk menemukan spesifik *row* waktu yang diperlukan hanya sedikit (dalam hitungan detik).
PostgreSQL menyimpan data dalam sistem operasi file, setiap tabel memiliki *file owner* dan *row* data yang satu tersimpan dalam file setelah yang lainnya. Index merupakan file terpisah yang disimpan oleh satu atau lebih kolom, berisikan *pointer* dalam file tabel yang memberi akses cepat ke spesifik *value* dalam tabel. PostgreSQL tidak bisa membuat indeks dengan otomatis, sehingga *user* dapat membuat indeks tersebut untuk seringkali digunakan kolom, biasanya dalam *clause* WHERE.
Berikut contoh *query* untuk membuat indeks.

```
db_personal=> CREATE INDEX teman_idx ON teman (umur);
CREATE
db_personal=>

db_personal=> -- lihatlah attribut tabel teman
db_personal=> \d teman
           Table "teman"
 Attribute |     Type            | Modifier
-----------+-----------+--------
 nama      | character(20)  |
 marga     | character(25)  |
 pekerjaan | character(25)  |
 kota              | character(30)  |
 state             | character(3)    |
 umur              | integer             |
 id        | integer             |
Index: teman_idx
```

Penamaan indeks dapat diberikan dengan bebas, akan tetapi sangat disarankan jika penamaanya mewakili nama yang di-indeks sehingga memudahkan perawatan. Selain itu indeks sebaiknya jangan digunakan pada tabel yang sangat jarang atau tidak pernah diakses.
Sekali sebuah indeks dibuat, maka tidak diperlukan laghi interfensi user karenas sistem akan secara otomatis melakukan update indeks ketika terjadi perubahan dalam tabel. Selain untuk perintah SELECT Indeks juga bermanfaat untuk UPDATE dan DELETE yang menggunakan kondisi pencarian.

Indeks Unik

Unique index biasa mirip dengan indeks kecuali itu tetapi lebih untuk mencegah duplikasi

nilai yang terdapat dalam tabel jadi dengan adanya *unique index* berarti pembaca tidak dapat meng-*insert* nilai yang sama dalam sebuah tabel. Contoh berikut menunjukkan bagaimana membuat satu tabel dan sebuah *unique index*, sebuah indeks dikatakan *unique* karena *unique keyword*. Jika data yang di-*insert* sama maka akan muncul pesan kesalahan atau *error*. Terkadang *unique index* dibuat hanya untuk mencegah terjadinya duplikasi nilai, bukan untuk alasan kinerja. Unique *index* multi kolom menjamin kombinasi dari sisa kolom *unique index*.

```
db_personal=> CREATE TABLE tes_duplikat (
db_personal(> nama VARCHAR(25)
db_personal(> );
CREATE
db_personal=>

db_personal=> CREATE UNIQUE INDEX tes_duplikat_indx ON tes_duplikat (nama);
CREATE
db_personal=>

db_personal=> INSERT INTO tes_duplikat VALUES ('Renaissance Indonesia');
INSERT 61551 1
db_personal=>

db_personal=> INSERT INTO tes_duplikat VALUES ('Renaissance Indonesia');
ERROR:  Cannot insert a duplicate key into unique index tes_duplikat_indx
```

Kolom Unik

Unique berfungsi untuk menjaga agar tidak terjadinya duplikasi nilai (kesamaan data) dalam sebuah kolom, ini dapat ditangani dengan membuat sebuah indeks pada kolom yang dimaksud. Unique ini sering digunakan dalam pembuatan primary key, karena dalam primary key tidak boleh terdapat dua atau lebih data yang salam dalam satu kolom. Berikut contohnya, kita akan membuat *unique index* dengan perintah CREATE TABLE *namatabel* (*namakolom* TYPE UNIQUE), Yang mana tabel tersebut merupakan *multiple null* yang dapat di INSERT ke dalam sebuah unique kolom. Jika sebuah *constraint unique* yang aktif lebih dari satu kolom, maka *unique* tersebut tidak dapat digunakan sebagai sebuah *constraint* kolom. Sebagai pengganti pembaca harus menggunakan baris baru untuk spesifikasi unique pada contraint kolom, ini dilakukan dengan membuat sebuah *tabel constraint* unique.

```
 db_personal=> CREATE TABLE tes_unique (
 db_personal(>                          id  INT UNIQUE,
 db_personal(>                          nama VARCHAR(25)
 db_personal(>                         );
NOTICE:  CREATE TABLE/UNIQUE will create implicit index 'tes_unique_id_key' for table
'tes_unique'
CREATE

 db_personal=> \d tes_unique
              Table "tes_unique"
 Attribute   |           Type                  | Modifier
--------+------------- ----+----------
 id              | integer                        |
 nama        | character varying(25)  |
Index: tes_unique_id_key

db_personal=> INSERT INTO tes_unique VALUES (26, 'Nidha');
INSERT 62037 1
db_personal=> INSERT INTO tes_unique VALUES (12, 'Esna');
INSERT 62038 1

db_personal=> INSERT INTO tes_unique VALUES (12, 'Esna');
ERROR:  Cannot insert a duplicate key into unique index tes_unique_id_key
```

```
 db_personal=> INSERT INTO tes_unique VALUES (12, 'Wati');
ERROR:  Cannot insert a duplicate key into unique index tes_unique_id_key

 db_personal=> INSERT INTO tes_unique VALUES (NULL, 'Wati');
INSERT 62041 1

 db_personal=> SELECT * FROM tes_unique;
 id   |  nama
---+-------
 26  |  Nidha
 12  |  Esna
      |  Wati
(3 rows)

 db_personal=> INSERT INTO tes_unique VALUES (' ', 'Dwi');
INSERT 62042 1

 db_personal=> SELECT * FROM tes_unique;
 id   |  nama
---+-------
 26  |  Nidha
 12  |  Esna
      |  Wati
  0   |  Dwi
(4 rows)

 db_personal=> INSERT INTO tes_unique VALUES (NULL, 'Alfikri');
INSERT 62043 1

 db_personal=> SELECT * FROM tes_unique;
 id   |   nama
---+---------
 26  |  Nidha
 12  |  Esna
      |  Wati
  0   |  Dwi
      |  Alfikri
(5 rows)
```

contoh berikut adalah membuat *unique* tabel constraint :

```
 db_personal=> CREATE TABLE tes_unique1 (
 db_personal(>                          id  INT,
 db_personal(>                          No_Mhs INT,
 db_personal(>                          Nama VARCHAR(25),
 db_personal(>                          UNIQUE (id, No_Mhs)
 db_personal(>                         );
NOTICE:  CREATE TABLE/UNIQUE will create implicit index 'tes_unique1_id_key' for table
'tes_unique1'
CREATE

 db_personal=> INSERT INTO tes_unique1 VALUES (2702, 98514045, 'Asti');
INSERT 62064 1
 db_personal=> INSERT INTO tes_unique1 VALUES (2702, 98514012, 'ALfikri');
INSERT 62065 1

 db_personal=> INSERT INTO tes_unique1 VALUES (2702, 98514045, 'ALfikri');
ERROR:  Cannot insert a duplicate key into unique index tes_unique1_id_key

 db_personal=> INSERT INTO tes_unique1 VALUES (7788, 98514045, 'ALfikri');
INSERT 62067 1
 db_personal=> SELECT * FROM tes_unique1;
  id     |  no_mhs   |  nama
----+--------+---------
 1220 | 98514073 | Andhie
```

```
 2702 | 98514045 | Asti
 2702 | 98514012 | ALfikri
 7788 | 98514045 | ALfikri
(4 rows)
```

CLUSTER

Perintah CLUSTER digunakan untuk melakukan perubahan pada tabel berdasarkan informasi pada Indeks, sehingga sebelum melakukan cluster terlebih dulu harus membuat indeks. Ketika sebuah tabel di cluster, maka secara fisik akan diurutkan kembali berdasarkan informasi dari indeks. Proses cluster hanya *one-time operation* artinya tidak bisa secara otomatis berulang seperti indeks, jadi meskipun terjadi perubahan maka tidak secara otomatis akan di cluster.

Sebagai contoh misalnya kita ingin memindahkan kolom nama dari tabel identitas ke sebuah tabel baru, kita tidak perlu terlebih dahulu membuat tabel baru tersebut karena perintah CLUSTER secara otomatis akan membuat tabel itu. Sehingga untuk memindahkan data yang berada dalam kolom tersebut cukup dengan menggunakan perintah SELECT, CLUSTER juga mendukung perintah ORDER BY dan WHERE untuk spesifikasinya, berikut implementasinya.

Sebagai catatan: membuat CLUSTER, haruslah pada tabel yang telah memiliki INDEX

```
db_personal=> CLUSTER identitas_indx ON identitas ;
CLUSTER

db_personal=> SELECT marga INTO TABLE tabel_new
db_personal-> FROM identitas
db_personal-> ORDER BY marga ;
SELECT

db_personal=> SELECT * FROM tabel_new ;
  marga
--------
 Adam
 Asti
 Asti
 Meilan
 Priti
 Utomo
 Wulanti
(7 rows)

db_personal=> SELECT nama,state,umur INTO TABLE tabel_baru
db_personal-> FROM identitas
db_personal-> ORDER BY umur  ;
SELECT
db_personal=> SELECT * FROM tabel_baru;
  nama     | state  | umur
---------- --+--- -+------
 Restu     | MG    |   13
 Iin               | NA     |   26
 Nidha     | DE     |   18
 Wati              | DE     |   21
 Wati              | YK     |   22
 Aci               | SB     |   22
 Esna              | YK     |   23
(7 rows)

db_personal=> SELECT nama,state,umur INTO TABLE tabel_satu
db_personal-> FROM identitas
db_personal-> WHERE umur < 23
db_personal-> ORDER BY umur ;
```

```
SELECT
db_personal=>
db_personal=> SELECT * FROM tabel_satu;
    nama    | state  | umur
------------+--- -+------
   Restu    | MG     |   13
   Nidha    | DE      |   18
   Wati            | DE     |   21
   Wati            | YK     |   22
   Aci             | SB     |   22
(5 rows)
```

Primary Key

Constraint PRIMARY KEY berfungsi untuk membuat sebuah *unique* kolom yang akan mengidentifikasi setiap *row*, pembuatannya merupakan gabungan dari UNIQUE dan *constraint* (batasan) NOT NULL. Tugas dari masing-masing tipe tersebut berbeda antara lain UNIQUE berfungsi untuk menjaga agar tidak terjadi sebuah duplikasi *value* (nilai yang sama) sedangkan NOT NULL bertugas untuk menjaga *value* NULL dalam sebuah kolom. Berikut contoh pembuatan sebuah kolom PRIMARY KEY. Sebagai catatan waktu pembuatan *primary key* secara otomatis sebuah indeks akan tebuat juga dan kolom tersebut ditetapkan sebagai NOT NULL, berikut contohnya:

```
 db_personal=> CREATE TABLE tes_primary (
 db_personal(>                          id INTEGER PRIMARY KEY,
 db_personal(>                          state CHAR(2)
 db_personal(>                          );
NOTICE:  CREATE TABLE/PRIMARY KEY will create implicit index 'tes_primary_pkey' for table
'tes_primary'
CREATE

 db_personal=> \d tes_primary
         Table "tes_primary"
 Attribute |     Type          | Modifier
-------+----------+----------
 id           | integer           | not null
 state      | character(2)  |
Index: tes_primary_pkey

 db_personal=> INSERT INTO tes_primary VALUES (1, 'DE');
INSERT 70230 1
 db_personal=> INSERT INTO tes_primary VALUES (2, 'YK');
INSERT 70230 1

 db_personal=> INSERT INTO tes_primary VALUES (2, 'YK');
ERROR:  Cannot insert a duplicate key into unique index tes_primary_pkey

 db_personal=> SELECT * FROM tes_primary;
 id   |  state
---+------
  1   |  DE
  2   |  YK
(2 rows)
```

jika kita ingin agar dalam satu tabel terdapat dua UNIQUE maka harus mendeklarasikan kolom *unique* tersebut pada baris tersendiri, artinya kita menggabungkan dua kolom menjadi satu form *primary key*. Sebagai catatan dalam sebuah tabel tidak boleh memiliki lebih dari satu PRIMARY KEY, *primary key* mempunyai arti khusus ketika menggunakan *foreign key*.

Berikut contoh mengkombinasikan dua kolom dalam satu from *primary key*:

```
 db_personal=> CREATE TABLE tes_primary1 (
 db_personal(>                          id INT,
 db_personal(>                          no_Mhs INT,
 db_personal(>                          nama VARCHAR(25),
 db_personal(>                          PRIMARY KEY (id, no_Mhs)
 db_personal(>                          );
NOTICE:  CREATE TABLE/PRIMARY KEY will create implicit index 'tes_primary1_pkey' for
table 'tes_primary1'
CREATE

 db_personal=> \d tes_primary1
             Table "tes_primary1"
  Attribute   | Type               |  Modifier
----- ---+-------------- --+--------
  id               | integer                          |  not null
  no_mhs   | integer                            |  not null
  nama       | character varying(25)  |
Index: tes_primary1_pkey

 db_personal=> INSERT INTO tes_primary1
 db_personal-> VALUES (2, 120, 'Rofiq');
INSERT 70251 1

 db_personal=> INSERT INTO tes_primary1
 db_personal-> VALUES (2, 121, 'Muzakir');
INSERT 70252 1

 db_personal=> INSERT INTO tes_primary1
 db_personal-> VALUES (3, 121, 'Noviar');
INSERT 70254 1

 db_personal=> INSERT INTO tes_primary1
 db_personal-> VALUES (2, 120, 'Ade');
ERROR:  Cannot insert a duplicate key into unique index tes_primary1_pkey

setelah itu amatilah dengan teliti isi dari tabel dibawah ini :
 db_personal=> SELECT * FROM tes_primary1;
 id | no_mhs |  nama
--+-------+--------
  2 |    120     | Rofiq
  2 |    121     | Muzakir
  3 |    121     | Noviar
(3 rows)
```

Foreign Key

Jika *primary key* membuat sebuah kolom UNIQUE dan NOT NULL, sedangkan FOREIGN KEY dari sisi yang lain yaitu membuat *constraint* (batasan) pada kolom di dalam tabel lain akan tetapi kolom tersebut harus berhubungan dengan kolom *primary key* pada tabel lain. Berikut kita akan membuat sebuah contoh tabel *relationship primary key* dan *foreign key*, *constraints* dari *foreign key* membuat REFERENCES yang dihubungkan ke *primary key* tabel yang lain.

```
 db_personal=> CREATE TABLE state (
 db_personal(>                     kode CHAR(2) PRIMARY KEY,
 db_personal(>                     NAMA VARCHAR(25)
 db_personal(>                    );
NOTICE:  CREATE TABLE/PRIMARY KEY will create implicit index 'state_pkey' for table
'state'
CREATE

 db_personal=> INSERT INTO state VALUES ('DE', 'Ambon');
```

```
INSERT 70269 1
 db_personal=> INSERT INTO state VALUES ('YK', 'Yogya');
INSERT 70270 1
 db_personal=> INSERT INTO state VALUES ('UP', 'SulSel');
INSERT 70271 1

 db_personal=> SELECT * FROM state;
 kode |  nama
----+--------
 DE    | Ambon
 YK    | Yogya
 UP    | SulSel
(3 rows)

 db_personal=> CREATE TABLE customer1 (
 db_personal(>                          id_customer  INTEGER,
 db_personal(>                          Nama            VARCHAR(25),
 db_personal(>                          Telp               CHAR(10),
 db_personal(>                          Alamat           VARCHAR(25),
 db_personal(>                          Kota               VARCHAR(20),
 db_personal(>                          state               CHAR(2) REFERENCES state,
 db_personal(>                          Kodepos        CHAR(6),
 db_personal(>                          Negara           VARCHAR(15)
 db_personal(> );
NOTICE:  CREATE TABLE will create implicit trigger(s) for FOREIGN KEY check(s)
CREATE
```

Berikut contohnya, misalkan kita membuat tabel *state* yang berisi kode dan nama kota. Jumlah huruf dari kode tersebut hanya berjumlah dua karakter, kemudian kita buat tabel *customer1* yang mana kolom *state* dari tabel tersebut merupakan *foreign key* yang berfungsi untuk menjaga agar tidak terjadi kesalahan dalam memasukkan data kode kota, jadi kode yang dimasukkan dalam kolom *state* pada tabel *customer1* harus sesuai dengan kode yang terdapat dalam *kolom state* pada tabel *state.* Berikut implementasinya

```
 db_personal=> INSERT INTO customer1 (state)
 db_personal-> VALUES ('DE');
INSERT 70295 1

 db_personal=> INSERT INTO customer1 (state)
 db_personal-> VALUES ('YK');
INSERT 70296 1

 db_personal=> INSERT INTO customer1 (state)
 db_personal-> VALUES ('UP');
INSERT 70297 1

 db_personal=> INSERT INTO customer1 (state)
 db_personal-> VALUES ('AB');
ERROR:  <unnamed> referential integrity violation - key referenced from customer1 not
found in state

 db_personal=> SELECT * FROM customer1;
 id_customer | nama | telp | alamat | kota | state | kodepos | negara
----------+----+- --+-- ---+----+----+-------+--------
                       |          |         |            |         |  DE   |
|
                       |          |         |            |         |  YK   |
|
                       |          |         |            |         |  UP   |
|
(3 rows)
```

Berikut kita akan membuat kelompok tabel menggunakan PRIMARY KEY dan FOREIGN KEY.

```
     db_personal=> CREATE TABLE customer (
     db_personal(>                        customer_id INTEGER PRIMARY KEY,
     db_personal(>                        nama            VARCHAR(25),
     db_personal(>                        telp               CHAR(10),
     db_personal(>                        alamat          VARCHAR(20),
     db_personal(>                        kota              VARCHAR(15),
     db_personal(>                        state             CHAR(2),
     db_personal(>                        kodepos       CHAR(6),
     db_personal(>                        negara         VARCHAR(15)
     db_personal(>                       );
    NOTICE:  CREATE TABLE/PRIMARY KEY will create implicit index 'customer_pkey' for table
'customer'
    CREATE
     db_personal=> CREATE TABLE employee (
     db_personal(>                        employee_id  INTEGER PRIMARY KEY,
     db_personal(>                        nama             VARCHAR(25),
     db_personal(>                        tanggal           DATE
     db_personal(>                       );
    NOTICE:  CREATE TABLE/PRIMARY KEY will create implicit index 'employee_pkey' for table
'employee'
    CREATE
     db_personal=> CREATE TABLE part (
     db_personal(>                      part_id      INTEGER PRIMARY KEY,
     db_personal(>                      nama        VARCHAR(25),
     db_personal(>                      berat         FLOAT
     db_personal(>                     );
    NOTICE:  CREATE TABLE/PRIMARY KEY will create implicit index 'part_pkey' for table 'part'
    CREATE
     db_personal=> CREATE TABLE salesorder (
     db_personal(>                          order_id           INTEGER,
     db_personal(>                          customer_id     INTEGER REFERENCES customer,
     db_personal(>                          employee_id    INTEGER REFERENCES employee,
     db_personal(>                          part_id             INTEGER REFERENCES part,
     db_personal(>                          tanggal_order   DATE,
     db_personal(>                          tanggal_ship    DATE,
     db_personal(>                          pembayaran    NUMERIC(8,2)
     db_personal(>                         );
    NOTICE:  CREATE TABLE will create implicit trigger(s) for FOREIGN KEY check(s)
    CREATE
```

Integritas Referensial

Apabila antara kedua tabel telah memiliki hubungan yaitu PRIMARY KEY dan FOREIGN KEY maka data pada kedua kolom yang saling berhubungan tersebut tidak dapat di-*update*. Dalam proses UPDATE dan DELETE dalam hubungan PK-FK, terdapat beberapa cara penanganan dalam SQL, yaitu;

NO ACTION proses UPDATE dan DELETE pada *primary key* dicegah oleh *foreign key* artinya tidak bisa melakukan kedua proses tersebut namun proses *delete* berlaku untuk *foreign key*.

CASCADE UPDATE dapat dilakukan pada kolom *primary key* maka secara otomatis kolom *foreign key* pada tabel lain (kolom yang berhubungan dengan *primary key*) akan ikut ter-*update* juga. (UPDATE tidak bisa dilakukan pada kolom *foreign key*).

SET NULL UPDATE dan DELETE pada *row primary key* karena *foreign key* telah di-SET NULL.

Berikut kita akan membuat sebuah tabel dimana salah satu kolomnya berhubungan dengan

kolom *primary key* pada tabel lain, pada kolom tersebut dideklarasikan beberapa *option update foreign key* dan penggunaan CASCADE dan NO ACTION.

```
     db_personal=> CREATE TABLE customer (
     db_personal(>                        customer_id  INTEGER,
     db_personal(>                        nama             VARCHAR(20),
     db_personal(>                        telp                CHAR(10),
     db_personal(>                        alamat           VARCHAR(25),
     db_personal(>                        kota               VARCHAR(15),
     db_personal(>                        state              CHAR(2) REFERENCES state
     db_personal(>                                                          ON UPDATE
CASCADE
     db_personal(>                                                          ON DELETE SET
NULL,
     db_personal(>                        KODEPOS    CHAR(5),
     db_personal(>                        NEGARA      VARCHAR(15)
     db_personal(> );
    NOTICE:  CREATE TABLE will create implicit trigger(s) for FOREIGN KEY check(s)
    CREATE

     db_personal=> CREATE TABLE db_personal_primary (
     db_personal(>                             id INTEGER PRIMARY KEY,
     db_personal(>                             nama VARCHAR(25)
     db_personal(>                            );
    NOTICE:  CREATE TABLE/PRIMARY KEY will create implicit index 'db_personal_primary_pkey'
for table 'db_personal_primary'
    CREATE

     db_personal=> CREATE TABLE db_personal_foreign (
     db_personal(>                             kode INTEGER REFERENCES db_personal_primary
     db_personal(>                                          ON UPDATE CASCADE
     db_personal(>                                          ON DELETE NO ACTION,
     db_personal(>                             nama VARCHAR(25)
     db_personal(> );
    NOTICE:  CREATE TABLE will create implicit trigger(s) for FOREIGN KEY check(s)
    CREATE

     db_personal=> INSERT INTO db_personal_primary VALUES (1, 'ELEX');
    INSERT 70465 1

     db_personal=> INSERT INTO db_personal_foreign VALUES (1, 'ELEX');
    INSERT 70467 1

     db_personal=> SELECT * FROM db_personal_primary;
     id   | nama
    ---+------
      1   | ELEX
    (1 row)

     db_personal=> SELECT * FROM db_personal_foreign;
     kode | nama
    ----+------
        1    | ELEX
    (1 row)
```

Kemudian lakukan proses UPDATE pada tabel *coba_primary* dan *coba_foreign*:

```
     db_personal=> UPDATE db_personal_foreign SET kode = '2';
    ERROR:  <unnamed> referential integrity violation - key referenced from
db_personal_foreign
    not found in db_personal_primary

     db_personal=> UPDATE db_personal_primary SET id = 2;
    UPDATE 1
```

Proses UPDATE pada tabel *coba_foreign* mengalami kegagalan (*error*), ini berarti proses *update* tersebut hanya dapat dilakukan pada tabel *coba_primary* (tabel *primary key*):

```
db_personal=> SELECT * FROM db_personal_primary;
id  | nama
- -+------
  2  | ELEX
(1 row)

db_personal=> SELECT * FROM db_personal_foreign;
kode | nama
----+------
    2   | ELEX
(1 row)
```

Perintah NO ACTION pada kolom *state* dalam tabel *coba_foreign* berfungsi untuk mencegah terjadinya proses penghapusan data. Pada tabel *coba_primary,* Sekarang coba kita jalankan perintah DELETE pada kedua tabel tersebut.

```
db_personal=> DELETE FROM db_personal_primary;
ERROR:  <unnamed> referential integrity violation - key in db_personal_primary still
referenced from db_personal_foreign

db_personal=> DELETE FROM db_personal_foreign;
DELETE 1

db_personal=> SELECT * FROM db_personal_primary;
id   | nama
---+------
  1   | ELEX
(1 row)

db_personal=> SELECT * FROM db_personal_foreign;
kode  | nama
-- --+------
(0 rows)
```

Check

Check berfungsi untuk melakukan pembatasan nilai masukan dalam sebuah kolom, sebagai contoh misalkan kita ingin agar kolom *gender* yang terdiri dari satu karakter hanya memiliki dua pilihan karakter yaitu **M** (*mail*) atau **F** (Fimail) ini dapat kita seting dengan menggunakan CHECK. Dengan menggunakan CHECK maka sebuah kolom hanya bisa diisi dengan data yang memenuhi kriteria dalam CHECK.

Berikut contohnya

```
db_personal=> CREATE TABLE mahasiswa (
db_personal(>                          nama  VARCHAR(10),
db_personal(>                          marga VARCHAR(15),
db_personal(>                          kota  VARCHAR(10),
db_personal(>       state CHAR(10)  CHECK (length(trim(state)) = 2),
db_personal(>       umur  INTEGER   CHECK (umur >= 0),
db_personal(>       gender CHAR(1)  CHECK (gender IN ('M', 'F')),
db_personal(>     tgl_bertemu DATE  CHECK (tgl_bertemu BETWEEN '1998-01-01'
db_personal(>                               AND CURRENT_DATE),
db_personal(>       CHECK (upper(trim(nama)) != 'ND' OR
db_personal(>              upper(trim(marga)) != 'ANDHIE')
db_personal(> );
CREATE

db_personal-> VALUES ('Nd', 'Andhie', 'Ambon', 'DE', 23, 'M', '1999-02-12');
ERROR:  ExecAppend: rejected due to CHECK constraint $5
```

```
 db_personal-> VALUES ('WATI', 'SHINTA', 'Yogya', 'YK', 21, 'F', '2001-03-10');
INSERT 70613 1

 db_personal=> INSERT INTO mahasiswa (nama, gender)
 db_personal-> VALUES ('Alfikri', 'L');
ERROR:  ExecAppend: rejected due to CHECK constraint                  mahasiswa_gender

 db_personal=> INSERT INTO mahasiswa (nama, gender)
 db_personal-> VALUES ('Alfikri', 'M');
INSERT 70616 1

 db_personal=> SELECT * FROM mahasiswa;
  nama   |   marga          | kota     |   state    | umur | gender | tgl_bertemu
------+-----------+-----+------+-- ---+--- --+---------
 WATI     | SHINTA | Yogya  |    YK     |   21    |  F        | 2001-03-10
 Alfikri    |                      |             |             |          |  M        |
(2 rows)
```

TRANSAKSI

Bab 11 – Transaksi

Deskripsi

Transaksi merupakan sebuah fungsi khusus dalam database, yang hanya dimiliki oleh database besar seperti PostgreSQL. Bab ini akan membahas tentang penggunaan transaksi dalam PostgreSQL untuk menjamin integritas sebuah transaksi.

Obyektif

Pada bab ini diharapkan peserta dapat memahami dan menggunakan operator transaksi dalam PostgreSQL serta melakukan Rollback sebuah transaksi.

Outline

- Transaksi
- Transaksi Multistatement
- ROLLBACK

LUW (logica units of work) atau yang dikenal dengan istilah transaksi adalah kumpulan atau sederetan operasi yang berkedudukan sebagai satu kesatuan proses. Misalnya dalam melakukan transaksi melalui ATM, proses tersebut mencakup pemasukan kartu ATM, pemasukan nomor PIN, penentuan jumlah uang, dan mengambil uang yang dikeluarkan oleh mesin ATM. Dalam transaksi ATM tersebut terdapat dua kemungkinan yang harus dipenuhi,

- Transaksi dianggap berhasil jika semua dari proses-proses tersebut berjalan dengan baik dan lancar,
- Transaksi dianggap gagal jika salah satu dari proses tersebut ada yang gagal.

Kedua kondisi tersebut harus dipenuhi salah satu karena jika tidak maka akan terjadi kekacauan, misalnya proses tiba-tiba berhenti pada saat memasukan nilai uang yang akan diambil. Kondisi tersebut akan menyebabkan nilai uang dalam database sudah dikurangi, padahal nasabah belum menerima uangnya. Dengan memilih salah satu kondisi diatas, maka proses akan dianggap berhasil atau gagal, dan jika gagal maka nasabah dianggap tidak melakukan apapun.

Dengan kondisi-kondisi tersebut PostgreSQL menyediakan transaksi yang juga dimiliki oleh database besar lainnya. Jadi tujuan utama dari transaksi adalah untuk menjamin bahwa proses yang seharusnya terjadi akan dilakukan sampai dengan selesai, dan tidak ada perubahan yang bersifat parsial jika transaksi belum diselesaikan dengan lengkap.

Contoh lain dadam database adalah ketika proses *update* sedang berjalan dan telah sampai pada *row* ke 100 kemudian kita menekan control-c atau keluar dari *database* tersebut dan kembali *login*, maka seluruh proses *update* tadi mengalami kegagalan. Dalam PostgreSQL digunakan BEGIN WORK untuk memulai sebuah transaki dan untuk mengakhirinya digunakan COMMIT. Dengan menggunakan COMMIT berarti telah terjadi persetujuan untuk melakukan perubahan pada database.

```
 db_personal=> CREATE TABLE tes_trans
 db_personal-> VALUES (id INTEGER);
CREATE

 db_personal=> INSERT INTO tes_trans VALUES (5);
INSERT 61873 1

 db_personal=> INSERT INTO tes_trans VALUES (4);
INSERT 61874 1

 db_personal=> SELECT * FROM tes_trans;
 id
----
  5
  4
(2 rows)

 db_personal=> BEGIN WORK;
BEGIN

 db_personal=> INSERT INTO tes_trans VALUES (3);
INSERT 61877 1

 db_personal=> COMMIT WORK;
COMMIT

 db_personal=> SELECT * FROM tes_trans;
 id
```

```
----
  5
  4
  3
(3 rows)
```

Transaksi Multistatement

Pada contoh diatas menggunakan INSERT dengan transaksi eksplisit, yang mana untuk memulai sebuah transaksi diawali dengan perintah BEGIN WORK dan untuk menjalankan perintah tersebut menggunakan perintah COMMIT WORK. Kita juga dapat melakukan sebanyak mungkin UPDATE, INSERT maupun DELETE dalam sebuah transaksi, sebagai contoh lihat di bawah ini yang mana terjadi dua kali INSERT dengan nilai yang berbeda menggunakan transaksi.

```
 db_personal=> SELECT * FROM tes_trans;
 id
----
  5
  4
(2 rows)

 db_personal=> BEGIN WORK;
BEGIN

 db_personal=> INSERT INTO tes_trans VALUES (3);
INSERT 61921 1

 db_personal=> INSERT INTO tes_trans VALUES (6);
INSERT 61922 1

 db_personal=> COMMIT WORK;
COMMIT

 db_personal=> SELECT * FROM tes_trans;
 id
----
  5
  4
  3
  6
(4 rows)
```

Proses ini sangat berguna untuk kegiatan transaksi pada sebuah bank. Sebagai contoh misalnya terdapat sebuah tabel nasabah dan transaksi yang mana keduanya saling berhubungan, jika seorang nasabah menyetor uang maka dia akan dicatat pada tabel nasabah sedangkan seluruh kegiatan masuk-keluarnya uang harus tercatat pada tabel transaksi. Jika proses insert berjalan dengan baik maka saldo pada tabel transaksi tetap konsisten, namun setelah tabel nasabah di-INSERT kemudian terjadi suatu kecelakaan seperti sistem *down* dan kemudian normal kembali maka nilai saldo pada tabel transaksi tidak bisa dipertanggungjawabkan. Semuanya itu akan teratasi jika digunakannya *transaction* pada proses tersebut, berikut contohnya:

```
 db_personal=> CREATE TABLE nasabah (
 db_personal(>                       id  INT NOT NULL PRIMARY KEY,
 db_personal(>                       nama VARCHAR(25) NOT NULL,
 db_personal(>                       saldo FLOAT NOT NULL
 db_personal(> );
NOTICE:  CREATE TABLE/PRIMARY KEY will create implicit index 'nasabah_pkey' for table
'nasabah'
```

```
CREATE

 db_personal=> CREATE TABLE transaksi (
 db_personal(>                        tanggal  DATETIME NOT NULL DEFAULT NOW(),
 db_personal(>                        id_nasabah  INT REFERENCES nasabah,
 db_personal(>                        jumlah  FLOAT NOT NULL
 db_personal(> );
NOTICE:  CREATE TABLE will create implicit trigger(s) for FOREIGN KEY check(s)
CREATE

 db_personal=> SELECT * FROM nasabah;
 id   | nama  | saldo
---+-----+------
(0 rows)

 db_personal=> SELECT * FROM transaksi;
 tanggal   |  id_nasabah    | jumlah
-------+-----------+-------
(0 rows)

 db_personal=> BEGIN WORK;
BEGIN

 db_personal=> INSERT INTO nasabah (id, nama, saldo)
 db_personal-> VALUES (2707, 'Wati', '100000');
INSERT 61915 1
 db_personal=> INSERT INTO nasabah (id, nama, saldo)
 db_personal-> VALUES (7802, 'Esna', '200000');
INSERT 61916 1
 db_personal=> INSERT INTO nasabah (id, nama, saldo)
 db_personal-> VALUES (8326, 'Nidha', '300000');
INSERT 61917 1
```

setiap melakukan sebuah transaksi baik itu uang masuk maupun uang keluar harus dicatat dalam tabel transaksi, berikut contohnya:

```
 db_personal=> INSERT INTO transaksi (id_nasabah, jumlah)
 db_personal-> VALUES (2707, '100000');
INSERT 61918 1
 db_personal=> INSERT INTO transaksi (id_nasabah, jumlah)
 db_personal-> VALUES (7802, '200000');
INSERT 61919 1
 db_personal=> INSERT INTO transaksi (id_nasabah, jumlah)
 db_personal-> VALUES (8326, '300000');
INSERT 61920 1
```

sebelum proses yang telah dilakukan dijalankan, adakan pengecekkan terlebih dahulu apakah telah sesuai atau tidak:

```
 db_personal=> SELECT * FROM nasabah;
  id      | nama   | saldo
-- --+------+--------
 2707  | Wati      | 100000
 7802  | Esna    | 200000
 8326  | Nidha   | 300000
(3 rows)

 db_personal=> SELECT * FROM transaksi;;
        tanggal                  | id_nasabah |  jumlah
--------------------+---------+--------
 2002-03-20 15:40:18+07 |       2707      | 100000
 2002-03-20 15:40:18+07 |       7802      | 200000
 2002-03-20 15:40:18+07 |       8326      | 300000
(3 rows)
```

kemudian jalankan proses tersebut dengan menggunakan perintah COMMIT WORK:

```
db_personal=> COMMIT WORK;
COMMIT
```

Berikut ini sebuah contoh tentang penggunaan UPDATE dalam sebuah transaksi, misalnya dalam tabel *nasabah* terdapat tiga *account* (rekening) dengan nomor *id* dan jumlah *saldo* yang dimiliki. Jika *Nidha* ingin mentransfer uang sejumlah Rp. 100.000 ke nomor rekening *Wati* maka langkah yang harus kita lakukan adalah meng-UPDATE account *Nidha* dan *Wati*. Langkah pertama yang dilakukan yaitu mengurangi jumlah saldo *Nidha* sebanyak Rp. 100.000 (karena mentransfer maka saldonya harus dikurangi) kemudian langkah berikutnya menambahkan jumlah saldo *Wati* sebanyak jumlah yang ditransfer oleh *Nidha* yaitu Rp 100.000 (dikarenakan *Wati* yang menerima transferan dari *Nidha* maka saldonya harus ditambah).

```
db_personal=> SELECT * FROM nasabah;
 id      | nama | saldo
--- -+-----+-------
 2707  | Wati    | 100000
 7802  | Esna  | 200000
 8326  | Nidha | 300000
(3 rows)

db_personal=> BEGIN WORK;
BEGIN

db_personal=> UPDATE nasabah SET saldo = saldo - 100000 WHERE id = 8326;
UPDATE 1

db_personal=> UPDATE nasabah SET saldo = saldo + 100000 WHERE id = 2707;
UPDATE 1
sebelum melakukan COMMIT WORK, ceklah terlebih dahulu proses update-nya dalam tabel
nasabah ? :
db_personal=> SELECT * FROM nasabah;
 id      | nama   | saldo
--- -+--- --+--------
 7802  | Esna   | 200000
 8326  | Nidha  | 200000
 2707  | Wati     | 200000
(3 rows)

db_personal=> COMMIT WORK;
COMMIT

db_personal=> SELECT * FROM nasabah;
 id      | nama   | saldo
--- -+--- --+--------
 7802  | Esna   | 200000
 8326  | Nidha  | 200000
 2707  | Wati     | 200000
(3 rows)
```

ROLLBACK

ROLLBACK digunakan untuk membatalkan semua proses dalam transaksi, sehingga segala perubahan yang telah dilakukan akan dibatalkan. ROLLBACK ini sangat bermanfaat untuk menghindari terjadinya kesalahan yang disimpan pada saat melakukan proses transaksi.

Misalkan setelah melakukan INSERT atau pun UPDATE pada tabel *nasabah* dengan menggunakan transaksi kemudian terjadi kesalahan maka untuk membatalkan proses transaksi tersebut dapat gunkan perintah ROLLBACK WORK. Misalnya kita ingin meng-*insert account* baru ke dalam tabel *nasabah* dengan jumlah saldo Rp. 500.000, namun terjadi kesalahan penulisan saldo maka langkah yang harus dilakukan adalah membatalkan transaksi tersebut dengan perintah ROOLBACK WORK. Dengan perintah ini maka secara otomatis kita akan keluar dari proses transaksi tanpa menyimpan data yang telah kita INSERT tadi.

```
 db_personal=> BEGIN WORK;
BEGIN

 db_personal=> INSERT INTO nasabah
 db_personal-> VALUES (3311, 'Restu', '550000');
INSERT 61923 1

 db_personal=> SELECT * FROM nasabah;
  id      | nama   | saldo
--- -+---- -+--------
 7802  | Esna   | 200000
 8326  | Nidha  | 200000
 2707  | Wati     | 200000
 3311  | Restu  | 550000
(4 rows)

 db_personal=> ROLLBACK WORK;
ROLLBACK

 db_personal=> SELECT * FROM nasabah;
  id      | nama   | saldo
--- -+--- --+--------
 7802  | Esna   | 200000
 8326  | Nidha  | 200000
 2707  | Wati     | 200000
(3 rows)

 db_personal=> BEGIN WORK;
BEGIN
 db_personal=> DELETE FROM nasabah
 db_personal-> WHERE nama = 'Restu';
DELETE 1
 db_personal=> ROLLBACK WORK;
ROLLBACK

 db_personal=> SELECT * FROM nasabah;
  id     | nama  | saldo
----+-----+--------
 7802 | Esna  | 200000
 8326 | Nidha | 200000
 2707 | Wati    | 200000
 3311 | Restu | 500000
(4 rows)
```

FUNGSI DAN SUBQUERY

Bab 12 – Fungsi dan Subquery

Deskripsi

Bab ini akan membahas tentang penggunaan fungsi dan subquery dalam database PostgreSQL. Bab ini juga akan membahas trigger dan beberapa subquery lainnya.

Obyektif

Pada bab ini diharapkan peserta dapat memahami, membuat dan menggunakan fungsi dalam database PostgreSQL.

Outline

- Fungsi SQL
- Fungsi PL/PGSQL
- Trigger
- Dukungan Fungsi
- Subquery Bervariable
- INSERT Data Menggunakan SELECT

Penggunaan fungsi pada banyak aplikasi dapat dilakukan sebab fungsi dapat dipasang di dalam sebuah *database server*. Dengan cara seperti ini setiap aplikasi yang membutuhkan fungsi tidak perlu lagi menyalinnya namun cukup dengan memanggil fungsi tersebut dari *database server*. Tidak seperti *client-side function*, *server-side function* dapat dipanggil dengan query. Fungsi yang terinstal di dalam *server* mudah dimodifikasi. Ketika sebuah fungsi pada *server* diganti maka semua klien dengan sendirinya akan menjalankan aplikasi menggunakan versi terbaru (fungsi yang baru). Fungsi sertaan (*built-in*) dalam PostgreSQL sangatlah banyak, pada prompt psql dapat dilihat fungsi sertaan tersebut dengan perintah *backsalsh-df* (\df). Selain memiliki fungsi *built-in* PostgreSQL juga menyediakan *built-in language* (bahasa sertaan) bernama SQL, dan bahasa ini dapat digunakan bagi yang ingin membuat fungsi sendiri.
Ada beberapa konsep yang menarik dari fungsi antara lain:

- Bahasa yang dipakai dapat didefenisikan sendiri dengan tersedianya parameter LANGUAGE, tanpa harus mengkompilasi ulang PostgreSQL.
- Kita dapat membuat dua buah fungsi dengan nama yang sama namun parameter masukkannya yang berbeda, baik tipe data maupun jumlahnya.

Fungsi SQL

Fungsi SQL adalah sebuah kumpulan query – biasanya query yang detail dan panjang - yang dibungkus menjadi satu dan disimpan dalam *database* dan kemudian apabila diperlukan hanya tinggal mengaksesnya tanpa mengetikan query detail. Untuk membuat fungsi gunakan perintah CREATE FUNCTION, dan menghapusnya dengan perintah DROP FUNCTION.
Ada beberapa faktor yang perlu diperhatikan dalam membuat fungsi antara lain:

- Nama Fungsi
- Nomor dari fungsi *argument*
- Tipe data dari setiap *argument*
- Tipe dari hasil fungsi
- Fungsi *action*
- Bahasa yang digunakan oleh fungsi *action.*

Berikut contoh sederhana pembuatan fungsi SQL untuk mengkonversi sebuah temperatur dari fahrenheit ke derajat celcius. Informasi pembuatannya adalah, nama fungsi *suhu*, *argument* yang diambil dari tipe *float*, tipe hasilnya *float*, fungsi *action*-nya SELECT ($1-32.0)*5.0/9.0; fungsi *language*-nya SQL.

```
db_personal=> CREATE FUNCTION suhu (float)
db_personal-> RETURNS float
db_personal-> AS 'SELECT ($1-32.0) * 5.0 / 9.0;'
db_personal-> LANGUAGE 'sql';
CREATE

db_personal=> SELECT suhu(140);
suhu
------
   60
(1 row)
```

Berikut kita akan membuat dua buah fungsi dengan nama yang sama, ini dapat dilakukan dalam PostgreSQL. Meskipun nama fungsinya sama tetapi parameter masukkannya harus

berbeda begitupun dengan jumlah dan tipe datanya:

```
 db_personal=> CREATE FUNCTION perkalian (FLOAT, FLOAT, FLOAT)
 db_personal-> RETURNS FLOAT
 db_personal-> AS 'SELECT ($1 + $2) * $3;'
 db_personal-> LANGUAGE 'sql';
CREATE

 db_personal=> SELECT perkalian (10,10,10);
 perkalian
-----------
       200
(1 row)

 db_personal=> SELECT perkalian (2.5,4.5,4);
 perkalian
-----------
        28
(1 row)

 db_personal=> CREATE FUNCTION perkalian (FLOAT, FLOAT,FLOAT, FLOAT)
 db_personal-> RETURNS FLOAT
 db_personal-> AS 'SELECT (($1 / $2) - $3) * $4;'
 db_personal-> LANGUAGE 'sql';
CREATE

 db_personal=> SELECT perkalian (20,2,5,2);
 perkalian
-----------
        10
(1 row)
```

berikut ini contoh fungsi SQL untuk menampilkan nama *kota* dari *state* pada tabel *identitas*:

```
db_personal=> SELECT * FROM identitas;
   nama    |  marga       |    kota            | state  | umur
-----------------------+--------------+-------------------+--------+----------
   Esna    | Asti  | Yogya       |  YK    |   23
   Iin             | Meilan       | Nangroe         |  NA     |   26
   Wati            | Asti  | Yogya       |  YK     |   22
   Wati            | Priti | Bantu       |  DE     |   21
   Restu   | Utomo | Mangga Dua  |  MG    |   13
   Nidha   | Adam  | Ambon       |  DE    |   18
(6 rows)
```

Tampilan di atas merupakan isi dari tabel *identitas,* sekarang kita langsung saja membuat fungsinya

```
db_personal=> CREATE FUNCTION nkota (text)
db_personal-> RETURNS text
db_personal-> AS ' SELECT CAST(kota AS TEXT)
db_personal'>          FROM identitas
db_personal'>         WHERE state = $1;'
db_personal-> LANGUAGE 'sql';
CREATE

db_personal=> SELECT nkota ('YK');
   nkota
------------
 Yogya
(1 row)

db_personal=> SELECT nkota ('DE');
   nkota
```

```
------------
 Ambon
(1 row)
```

Fungsi PL/PGSQL

PL/PGSQL merupakan bahasa yang lain untuk membuat sebuah fungsi, biasanya bahasa ini digunakan untuk menangani fungsi yang lebih kompleks. Pl/pgsql sudah terdapat dalam instalasi PostgreSQL, namun kita perlu mendaftarkannya pada masing-masing database yang akan menggunakannya. Untuk itu kita perlu menginstal pl/pgsql pada *database* yang diinginkan. Langkah yang dilakukan adalah;

- Masuklah terlebih dahulu sebagai *postgres superuser* dan dari prompt pgsql ketikkan perintah *createlang plpgsql nama_database*, berikut contoh urutan penginstalannya:

```
[root@localhost root]#  su postgres
[postgres@localhost root]$  cd
[postgres@localhost pgsql]$
```

Untuk melihat opsinya denga jelas dapat dilihat pada help-nya pada prompt dengan perintah *createlang -?*

```
$  createlang -?

createlang installs a procedural language into a PostgreSQL database.

Usage:
  createlang [options] [langname] dbname

Options:
  -h,   --host=HOSTNAME                Database server host
  -p,   --port=PORT                            Database server port
  -U,  --username=USERNAME         Username to connect as
  -W,  --password                              Prompt for password
  -d,   --dbname=DBNAME                Database to install language in
  -L,   --pglib=DIRECTORY                Find language interpreter file in
                    DIRECTORY
  -l,   --list                                                 Show a list of currently
installed                                       languages

If 'langname' is not specified, you will be prompted interactively.
A database name must be specified.
Report bugs to <pgsql-bugs@postgresql.org>.
```

Kemudian gunakan perintah *createlang plpgsql coba* untuk melakukan penginstalan, disini *coba* merupakan nama *database* yang dituju untuk menginstal *language plpgsql*. Artinya kita menginstal plpgsql pada *database coba.*

```
$  createlang  plpgsql db_personal
```

perintah di bawah ini untuk menampilkan language yang telah terinstal, hasil yang ditampilkan kolom *trusted* adalah "t" (true) artinya fungsi yang menggunakan *language plpgsql* dapat dijalankan.

```
$  createlang -l  db_personal
     Procedural languages
  Name    | Trusted?   |  Compiler
------------+--------------+-------------------
 plpgsql   |    t             |  PL/pgSQL
(1 row)
```

ada beberapa faktor yang harus diperhatikan pada bahasa ini antara lain :

- DECLARE mendeklarasikan variabel yang digunakan dalam fungsi
- SELECT INTO sebuah form spesial (khusus) dari SELECT yang memperbolehkan hasil query menempati ke dalam variabel.
- RETURN Exit dan merupakan hasil value dari fungsi

Berikut ini implementasi dari contoh penggunaan fungsi dengan bahasa PLPGSQL:

```
db_personal=> CREATE FUNCTION nama_kota (text)
db_personal-> RETURNS text
db_personal-> AS 'DECLARE hasil TEXT;
db_personal'>  BEGIN
db_personal'>              SELECT INTO hasil CAST(kota AS TEXT)
db_personal'>              FROM identitas
db_personal'>              WHERE state = $1;
db_personal'>               RETURN hasil;
db_personal'>  END;'
db_personal-> LANGUAGE 'plpgsql';
CREATE

db_personal=> SELECT nama_kota ('YK');
 nama_kota
-----------------
 Yogya
(1 row)

db_personal=> SELECT nama_kota ('MG');
 nama_kota
--------------------
 Mangga Dua
(1 row)
```

Di bawah ini merupakan contoh fungsi untuk karakter yang agak kompleks

```
 db_personal=> CREATE FUNCTION operasi_bil (FLOAT, FLOAT, FLOAT,           FLOAT)
 db_personal-> RETURNS FLOAT
 db_personal-> AS '
db_personal'>        BEGIN
db_personal'>                  RETURN ($1 * $2) - ($3 / $4);
db_personal '>        END;'
 db_personal-> LANGUAGE 'plpgsql';
CREATE

 db_personal=> SELECT operasi_bil (4,5,10,2);
 operasi_bil
-----------------
          15

 db_personal=> SELECT operasi_bil (4,5,10,2) AS hasilnya ;
 hasilnya
-------------
       15
```

Untuk menghapus sebuah fungsi harus menyertakan parameternya, dikarenakan komposisi parameter suatu fungsi merupakan bagian dari "ID" fungsi itu sendiri. Misalkan kita akan menghapus fungsi ***operasi_bil,*** pada fungsi ini terdapat empat parameter yaitu FLOAT sehingga untuk menghapusnya perlu disertakan semua parameter yang ada, berikut contohnya:

```
 db_personal=> DROP FUNCTION operasi_bil(FLOAT, FLOAT, FLOAT,        FLOAT);
DROP
```

Trigger

Trigger digunakan untuk menyisipkan sebuah fungsi pada saat suatu *record* di-INSERT, UPDATE dan DELETE. Trigger sangat ideal untuk mengecek atau memodifikasi sebuah data pada kolom sebelum dimasukkan ke dalam *database*, sehingga sebuah fungsi dapat dipanggil setiap saat secara otomatis ketika sebuah *row* akan dimodifikasi. Ciri khas dari fungsi yang diperuntukkan untuk *trigger* adalah menghasilkan output bertipe OPAQUE. Tipe *opaque* adalah sebuah tipe yang menginformasikan pada *database* bahwa fungsi tersebut tidak menghasilkan satu dari tipe data yang ditetapkan SQL dan tidak secara langsung dapat digunakan dalam statemen SQL. Language (bahasa) PL/PGSQL dapat digunakan untuk *trigger procedure*, fungsi untuk *trigger* ini memiliki beberapa variabel khusus yang terdeklarasi secara otomatis.
Variabel tersebut antara lain:

- NEW: Variabel yang berisi nilai baru suatu record pada saat INSERT atau UPDATE, bertipe RECORD.
- OLD: Variabel yang berisi nilai lama suatu record pada saat UPDATE atau DELETE, juga bertipe RECORD.

Berikut ini beberapa contoh penggunaan fungsi sebagai *trigger procedure*:
Contoh *trigger* berikut ini memastikan isi *field* atau kolom *nama* pada tabel *anggota* selalu huruf besar, langkah pertama buatlah fungsinya terlebih dahulu:

```
 db_personal=> CREATE FUNCTION tes_trigger()
 db_personal-> RETURNS opaque
 db_personal-> AS 'BEGIN
db_personal'>          NEW.nama := UPPER(NEW.nama);
db_personal'>          RETURN NEW;
db_personal'>     END;'
 db_personal-> LANGUAGE 'plpgsql';
CREATE
```

kemudian lanjutkan dengan pembuatan *trigger* yang berfungsi untuk memanggil fungsi secara otomatis ketika kita melakukan INSERT ataupun UPDATE pada tabel *anggota*.

```
 db_personal=> CREATE TRIGGER tes1_trigger
 db_personal-> BEFORE INSERT OR UPDATE
 db_personal-> ON anggota
 db_personal-> FOR EACH ROW
 db_personal-> EXECUTE PROCEDURE tes_trigger();
CREATE
```

cobalah INSERT beberapa data ke dalam tabel anggota:

```
 db_personal=> INSERT INTO anggota (id, nama)
 db_personal-> VALUES (26, 'andhie');
INSERT 70831 1

 db_personal=> INSERT INTO anggota
 db_personal-> VALUES (83, 'rWatia');
INSERT 70832 1
```

tampilkan isi dari tabel *anggota,* hasilnya seperti pada tabel di bawah ini. Jadi setiap data yang kita INSERT walaupun dalam penulisannya menggunakan huruf kecil namun secara otomatis *trigger* akan memanggil fungsi yang bertugas untuk mengganti setiap data yang masuk agar hasilnya nanti selalu menjadi huruf besar:

```
 db_personal=> SELECT * FROM anggota;
  id     |    nama
--------+-------------
  26    |  ANDHIE
  83    |  RWATIA
(2 rows)
```

berikut ini kita akan memcoba membuat sebuah fungsi sebagai *trigger procedure,* maksud dari *trigger procedure* di bawah ini adalah memberi ketentuan atau batasan ketika mamasukkan data dalam kolom pada tabel *namakota*. Contoh batasan yang penulis berikan antara lain nama kota yang akan di INSERT ke dalam kolom *nama* harus berupa karakter huruf dan tidak boleh kurang dari tiga karakter serta selalu menjaga agar huruf pertama dari nama yang dimasukkan berupa huruf besar, sedangkan nama *state* yang ingin dimasukkan ke dalam kolom *state* harus berjumlah dua karakter dan selalu menjaga agar nama *state* berupa huruf besar, berikut contohnya:
Sebelumnya kita telah memiliki tabel *namakota* yang berisi tiga buah data*,* jika belum punya buatlah tabel baru dengan dua buah atribut yaitu *nama* dan *kota* kemudian masukkan tiga buah data sehingga jika di SELECT hasilnya seperti tabel dibawah ini. Pada tabel inilah nantinya akan kita buat fungsi sebagai *trigger procedure* :

```
 db_personal=> SELECT * FROM namakota;
    nama    | state
---------------------+----------
 Yogyakarta        | YK
 Ambon     | DE
 Kaltim    | KT
(3 rows)
```

langkah berikutnya membuat fungsi yang ketentuan serta batasannya sesuai dengan apa yang penulis ungkapkan pada penjelasan di atas:

```
 db_personal=> CREATE FUNCTION trigger_modifikasi()
 db_personal-> RETURNS opaque
 db_personal-> AS 'BEGIN
db_personal'>          IF new.nama !~ ''^[A-Za-z]*$''
db_personal'>          THEN RAISE EXCEPTION ''nama kota harus berupa karakter huruf.'';
db_personal'>          END IF;
db_personal'>          IF new.state !~ ''^[A-Za-z][A-Za-z]$''
db_personal'>          THEN RAISE EXCEPTION ''nama state harus berjumlah dua karakter
huruf.'';
db_personal'>          END IF;
db_personal'>          IF length(trim(new.nama)) < 3
db_personal'>          THEN RAISE EXCEPTION ''nama kota harus lebih dari tiga karakter
huruf.'';
db_personal'>          END IF;
db_personal'>          new.nama = initcap(new.nama);
db_personal'>          new.state = upper(new.state);
db_personal'>          RETURN new;
db_personal'>      END;'
 db_personal-> LANGUAGE 'plpgsql';
CREATE
```

kemudian buatlah *trigger procedure*-nya, untuk informasi lebih jelas tentang pembuatan *trigger* lihat pada manual page *create trigger*.

```
 db_personal=> CREATE TRIGGER trigger_nkota
 db_personal-> BEFORE INSERT OR UPDATE
 db_personal-> ON namakota
 db_personal-> FOR EACH ROW
 db_personal-> EXECUTE PROCEDURE trigger_modifikasi();
```

```
CREATE
```

setelah itu hapuslah seluruh data yang berada pada tabel *namakota* dan tampilkan tabel tersebut:

```
 db_personal=> DELETE FROM namakota;
DELETE 3

 db_personal=> SELECT * FROM namakota;
 nama   |  state
----------+----------
(0 rows)
```

cobalah INSERT beberapa data yang tidak sesuai dengan ketentuan atau batasan yang telah ditetapkan pada fungsi sebagai *trigger procedure* :

```
 db_personal=> INSERT INTO namakota VALUES ('timur2', 'k');
ERROR:  nama kota harus berupa karakter huruf.

 db_personal=> INSERT INTO namakota VALUES ('timur', 'k');
ERROR:  nama state harus berjumlah dua karakter huruf.

 db_personal=> INSERT INTO namakota VALUES ('jk', 'jt');
ERROR:  nama kota harus lebih dari tiga karakter huruf.
```

Dari output di atas menunjukkan bahwa *trigger procedure* selalu menolak jika data yang dimasukkan tidak sesuai dengan kententuan yang telah ditetapkan di dalam *trigger procxedure* tersebut. Kemudian masukkan data sesuai dengan ketentuan yang ditetapkan:

```
 db_personal=> INSERT INTO namakota VALUES ('ambon', 'de');
INSERT 70842 1

 db_personal=> INSERT INTO namakota VALUES ('yogyakarta', 'yk');
INSERT 70843 1

 db_personal=> SELECT * FROM namakota;
 nama       |  state
----------------------+---------
 Ambon      |  DE
Yogyakarta  |  YK
(2 rows)
```

Untuk menghapus sebuah trigger juga menggunakan perintah *drop*, namun sedikit berbeda dengan cara menghapus fungsi, penghapusan *trigger* harus mengikutkan *nama_tabel.* Berikut contoh perintahnya DROP TRIGGER *nama_trigger* ON *namatabel*. Untuk lebih jelasnya dapat dilihat pada *manual page drop trigger*:

```
DROP TRIGGER trigger_nkota ON namakota;
DROP
```

Dukungan Fungsi

PostgreSQL memiliki fungsi yang dapat mengubah suatu nilai dalam suatu kolom atau baris menjadi huruf besar. Fungsi tersebut bernama *upper(nama_kolom)*, berfungsi memanggil fungsi *upper* dengan *nama_kolom* sebagaii argumen sehingga menghasilkan nilai pada kolom dalam huruf besar. Berikut contohnya.

```
db_personal=> SELECT nama  FROM identitas WHERE nama = 'Restu';
  nama
---------
```

```
 Restu
(1 row)

db_personal=> SELECT upper (nama) FROM identitas WHERE nama = 'Restu';
  upper
---------
 RESTU
(1 row)
```

Subquery Konstan

Subquery atau sub select merupakan bentuk query yang terdapat dalam query yang lain. Artinya dengan melakukan pada sebuah operasi pada subquery, maka query-query di dalamnya akan dieksekusi.
Untuk mempermudah perhatikan contoh berikut;
SELECT * FROM tbl_personal
WHERE “dt_TglLahir” =
(SELECT MAX(“dt_TglLahir”) FROM tbl_personal)
Perhatikan pada contoh diatas, bahwa where di bandingkan dengan hasil dari perintah select bukan dengan nilai konstan,
Sebagai contoh penulis menggunakan tabel *identitas*, misalnya kita ingin mencari siapa yang tidak memiliki kesamaan *state* dalam tabel *identitas* yang bermarga *asti*. Kita dapat mengetahui *state* tersebuat menggunakan string *constant* 'YK', jika dipindahkan ke *state* yang lain dapat dilakukan dengan mengganti *query*-nya. Untuk lebih jelas perhatikan contoh berikut, *query* pertama menggunakan *self-join* untuk membandingkan *state* Esna Asti. Sedangkan pada *query* terakhir menunjukkan penggunaan *subquery* untuk menghasilkan *state* 'YK',

```
db_personal=> SELECT * FROM identitas;
   nama    |  marga       |    kota            | state  | umur
-----------+--------+----- -----+-------+----
   Esna    | Asti  | Yogya       |  YK    |   23
   Iin             | Meilan       | Nangroe           |  NA    |   26
   Wati            | Asti  | Yogya       |  YK    |   22
   Wati            | Priti | Bantu       |  DE    |   21
   Restu   | Utomo | Mangga Dua  |  MG    |   13
   Nidha   | Adam  | Ambon       |  DE    |   18
(6 rows)

db_personal=> SELECT n1.nama, n1.marga, n1.state
db_personal-> FROM identitas n1, identitas n2
db_personal-> WHERE n1.state <> n2.state AND
db_personal->               n2.nama = 'Esna' AND
db_personal->               n2.marga = 'Asti'
db_personal-> ORDER BY nama, marga;
   nama    |  marga       | state
--- ---------+--------+-------
   Iin             |  Meilan      | NA
   Nidha   |  Adam | DE
   Restu   |  Utomo       | MG
   Wati            |  Priti       | DE
(4 rows)

db_personal=> SELECT n1.nama, n1.marga, n1.state
db_personal-> FROM identitas n1
db_personal-> WHERE n1.state <> (
db_personal(>                    SELECT n2.state
db_personal(>                    FROM   identitas n2
db_personal(>                    WHERE  n2.nama = 'Esna' AND
```

```
db_personal(>                    n2.marga = 'Asti'
db_personal(>                    )
db_personal-> ORDER BY nama, marga ;
   nama    |  marga       | state
-----------+--------+-------
   Iin             |  Meilan      | NA
   Nidha   |  Adam | DE
   Restu   |  Utomo       | MG
   Wati            |  Priti       | DE
(4 rows)
```

Subquery Bervariable

Pada proses penambahan konstan, *subquery* dapat bertindak sebagai penghubung. Sebagai contoh misalnya kita ingin mencari umur yang tertua pada tabel identitas maka untuk menyelesaikannya gunakan perintah HAVING dan tabel alias, lihat contoh di bawah ini.

```
db_personal=> SELECT n1.nama, n1.marga, n1.umur
db_personal-> FROM identitas n1, identitas n2
db_personal-> WHERE n1.state = n2.state
db_personal-> GROUP BY n2.state, n1.nama, n1.marga, n1.umur
db_personal-> HAVING n1.umur = max(n2.umur)
db_personal-> ORDER BY nama, marga ;
   nama    |  marga       | umur
-----------+--------+------
   Esna    |  Asti |   23
   Iin             |  Meilan      |   26
   Restu   |  Utomo       |   13
   Wati            |  Priti       |   21
(4 rows)

db_personal=> SELECT n1.nama, n1.marga, n1.umur
db_personal-> FROM identitas n1
db_personal-> WHERE umur = (
db_personal(>                SELECT MAX(n2.umur)
db_personal(>                FROM   identitas n2
db_personal(>                WHERE  n1.state = n2.state
db_personal(>               )
db_personal-> ORDER BY nama, marga ;
   nama    |  marga       | umur
-----------+--------+------
   Esna    |  Asti |   23
   Iin             |  Meilan      |   26
   Restu   |  Utomo       |   13
   Wati            |  Priti       |   21
(4 rows)


db_personal=> SELECT DISTINCT employee.nama
db_personal-> FROM employee, salesorder
db_personal-> WHERE employee.employee_id = salesorder.employee_id AND
db_personal->       salesorder.tanggal_order = '03/11/2002' ;
    nama
---------
     Asti
     Nidha
(2 rows)

db_personal=> SELECT nama
db_personal-> FROM employee
db_personal-> WHERE employee_id IN (
db_personal(>                       SELECT employee_id
db_personal(>                       FROM   salesorder
```

```
db_personal(>                       WHERE  tanggal_order = '03/11/2002'
db_personal(>                      );
 nama
---------
     Asti
     Nidha
(2 rows)
```

INSERT Data Menggunakan SELECT

Setiap INSERT berisikan daftar konstan sebuah nilai yang akan di masukkan ke dalam *row* pada tabel yang dituju. Pada PostgreSQL perintah INSERT digunakan untuk memasukkan suatu data kedalam tabel, namun selain *INSERT* kita dapat juga memanfaatkan perintah SELECT untuk menambah atau memasukkan data tersebut. Berikut contohnya, misalkan kita mempunyai dua buah tabel dengan nama *satu* dan *tiga*. Sekarang kita ingin menambah data yang diambil dari tabel *satu* dan ditambahkan pada tabel *tiga,* maka langkah yang kita lakukan tidak perlu meng-*insert* lagi satu per satu, namun cukup dengan menggunkan perintah *select* maka data tersebut secara otomatis seluruh data pada tabel *satu* akan ter-*insert* ke dalam tabel *tiga* , dengan catatan kedua tabel tersebut harus memiliki atribut yang sama.

```
 db_personal=> SELECT * FROM satu;
 nama  |   marga    | state
-------+--------+-------
 Wati    | Shinta  | YK
 Nidha | Adam       | DE
(2 rows)

 db_personal=> SELECT * FROM tiga;
 nama    | state
------+-------
(0 rows)

 db_personal=> INSERT INTO tiga
 db_personal-> SELECT nama, state
 db_personal-> FROM satu ;
INSERT 0 2

 db_personal=> SELECT * FROM tiga;
   nama    |  state
---------- --+-------
   Wati           |  YK
   Nidha   |  DE
(2 rows)
```

kita juga dapat menggabungkan kolom *nama* dan *marga* dari tabel *satu* menjadi satu kolom pada tabel *tiga,* menggunakan perintah *trim()* dan *operator* ||.

```
 db_personal=> SELECT * FROM tiga;
   nama    |  state
---------- --+-------
   Wati           |  YK
   Nidha   |  DE
(2 rows)

 db_personal=> INSERT INTO tiga (nama, state)
 db_personal-> SELECT trim(nama) || ' ' || marga, state
 db_personal-> FROM satu ;
```

```
INSERT 0 2

 db_personal=> SELECT * FROM tiga;
      nama       | state
----------------+-------
 Wati                    | YK
 Nidha                          | DE
 Wati Shinta             | YK
 Nidha Adam              | DE
(4 rows)
```

Membuat Tabel Menggunakan SELECT

Untuk menambah data ke dalam tabel biasanya kita gunakan perintah SELECT dan INSERT, namun pada *session* ini penulis ingin menunjukkan bahwa perintah SELECT bisa juga digunakan untuk membuat tabel baru dari tabel yang telah ada. SELECT...INTO memiliki arti mengkombinasikan perintah CREATE TABLE dan SELECT dalam sebuah statemen tunggal, sedangkan perintah AS dapat digunakan untuk memilih nama kolom dan kemudian dikontrol dalam tabel baru. Perintah SELECT....INTO *namatabel* dapat ditulis juga seperti CREATE TABLE *namatabel* AS SELECT.... . Berikut contohnya misalkan kita ingin membuat tabel baru yang mana semua atributnya sama dengan tabel *satu*:

```
 db_personal=> SELECT * FROM satu;
 nama  |   marga    | state
--- --+--------+-------
 Wati     | Shinta  | YK
 Nidha  | Adam       | DE
(2 rows)

 db_personal=> SELECT nama, marga, state
 db_personal-> INTO newsatu
 db_personal-> FROM satu;
SELECT

 db_personal=> \d newsatu
              Table "newsatu"
 Attribute |         Type |  Modifier
------------+--------- -------+---------
 nama      | character varying(10) |
 marga     | character varying(10) |
 state             | character(2) |

 db_personal=> SELECT * FROM newsatu;
 nama  |   marga    | state
-----+--------+-------
 Wati    | Shinta  | YK
 Nidha | Adam       | DE
(2 rows)
```

Tidak ada ketentuan yang mengharuskan tabel baru yang dibuat harus memiliki semua atribut yang terdapat pada tabel *satu.* Misalkan tabel *satu* memiliki tiga atribut yaitu *nama, marga, state*. Kemudian kita ingin membuat tabel baru dengan dua buah atribut yaitu *nama* dan *marga*, maka hal semacam ini dapat juga dilakukan dengan menggunakan perintah SELECT.

```
 db_personal=> SELECT nama, marga
 db_personal-> INTO satudua
 db_personal-> FROM satu ;
```

SELECT

```
 db_personal=> \d satudua
               Table "satudua"
 Attribute |         Type |  Modifier
---------- ----+--------- -------+---------
 nama      | character varying(10) |
 marga     | character varying(10) |

 db_personal=> SELECT * FROM satudua;
   nama      |   marga
----- ---+---------
   Wati             | Shinta
   Nidha      | Adam
(2 rows)
```

OPERASI FILE

Bab 13 – Operasi File

Deskripsi

Bab ini akan membahas pemindahan data dari database PostgreSQL ke dalam file data sehingga bisa di olah menggunakan aplikasi spreatsheet.

Obyektif

Pada bab ini diharapkan peserta dapat memahami hubungan database PostgreSQL dengan file data, dan melakukan ekport database langsung ke raw file.

Outline

- Menggunakan Perintah COPY
- Format File COPY
- DELIMITERS

Menggunakan Perintah COPY

Perintah COPY dapat digunakan untuk memindahkan data yang terdapat di dalam tabel pada *database* kita ke sebuah direktori. Kita juga dapat mengembalikan data tersebut ke posisi semula, untuk meng-COPY data tersebut gunakan perintah COPY *nama_tabel* TO *'namadirektori/nama_file_baru'* dan untuk mengembalikan data tersebut pada *database* asalnya gunakan perintah
COPY *nama_tabel* TO *'nama_file'*,
Berikut adalah contoh penggunaan COPY ;

```
 db_personal=> CREATE TABLE tes_copy (
 db_personal(>                        nama VARCHAR(10),
 db_personal(>                        state CHAR(2),
 db_personal(>                        kota VARCHAR(10)
 db_personal(> );
CREATE

 db_personal=> INSERT INTO tes_copy
 db_personal-> VALUES ('Pamungkas', 'BD', 'Cipanas');
INSERT 70641 1

 db_personal=> INSERT INTO tes_copy
 db_personal-> VALUES ('Wulantini', 'MD', 'Madura');
INSERT 70642 1

 db_personal=> COPY tes_copy TO '/tmp/hasil.out';
ERROR:  You must have Postgres superuser privilege to do a COPY              directly to
or from a file.  Anyone can COPY to stdout or from         stdin.  Psql's \copy command also
works for anyone.
```

perintah *copy* harus dijalankan melalui superuser postgres, untuk itu kita harus melakukan login ulang (*relogin*) sebagai *user* Postgres.

```
 db_personal=> \c db_personal postgres
You are now connected to database db_personal as user postgres.

 db_personal=> COPY tes_copy TO '/tmp/hasil.out';
COPY

 db_personal=> SELECT * FROM tes_copy;
   nama    | state  |  kota
-----------+-- --+---------
 Pamungkas | BD     | Cipanas
 Wulantini | MD    | Madura
(2 rows)
```

sebelum menyalin kembali data (tabel) tersebut ke *database* asalnya, hapus terlebih dahulu data (tabel) pada *database* tersebut agar tidak terjadi penumpukan:

```
 db_personal=> DELETE FROM tes_copy;
DELETE 2

 db_personal=> SELECT * FROM tes_copy;
 nama  | state   |  kota
-----+-- ---+-------
(0 rows)
```

kemudian kita salin kembali file tersebut ke dalam tabel *tes_copy* pada *database* db_personal, berikut contohnya:

```
 db_personal=> COPY tes_copy FROM '/tmp/hasil.out';
COPY
```

```
 db_personal=> SELECT * FROM tes_copy;
   nama          | state  |  kota
----------+-- --+---------
 Pamungkas  | BD     | Cipanas
 Wulantini       | MD    | Madura
(2 rows)
```

Format File COPY

Penggunaan perintah COPY....TO dapat memindahkan (*export*) data ke dalam aplikasi lain, dan perintah COPY....FROM dapat meng-*import* data dari aplikasi yang lain. Jika kita ingin mengkonstruksikan sebuah file untuk digunakan dengan perintah COPY atau membaca file tersebut dengan cara menyalinnnya ke aplikasi lain, maka pembaca harus memahami tentang format file COPY. Jika hasil *copy* data tidak beraturan artinya antara kolom tidak dipisahkan oleh spasi maka kita dapat menggunakan perintah *SED* yang tersedia pada prompt yang berfungsi untuk mengatur jarak atau spasi pada file tersebut. Untuk lebih jelasnya tentang *sed* lihat pada *manual pages*-nya.

```
 db_personal=> \q

$ cat /tmp/hasil.out
Pamungkas       BD      Cipanas
Wulantini           MD      Madura

$ sed 's/   /<TAB>/g'   /tmp/hasil.out
Pamungkas       BD      Cipanas
Wulantini           MD      Madura
```

DELIMITERS

DELIMITERS merupakan sebuah opsi default yang disediakan oleh PostgreSQL, ketika kita menyalin sebuah data menggunakan DELIMITERS maka otomatis data tersebut akan terbentuk secara teratur sesuai dengan simbol delimiter yang kita gunakan. Misalkan kita gunakan simbol *pipe* (|), ketika data tersebut berhasil disalin maka antara data pada masing-masing kolom akan dipisahkan oleh simbol *pipe* (|).

```
 db_personal=>  COPY tes_copy TO '/home/andhie/hasil.out ' USING DELIMITERS '|';
COPY

 db_personal=> \q

$ cat hasil.out
Pamungkas | BD | Cipanas
Wulantini | MD | Madura

 db_personal=> DELETE FROM tes_copy;
DELETE 2

 db_personal=> SELECT * FROM tes_copy;
 nama   | state   | kota
-- ---+-----+------
(0 rows)
```

Kemudian coba kita salin kembali file *hasil.out* ke dalam tabel *tes_copy* dengan tidak mengikutkan DELIMITERS. Hasil yang didapatkan tidak sesuai dengan tabel semula yang mana kolom *state* dan *kota* kosong atau tidak terdapat data. Ini dikarenakan kita tidak mengikutkan DELIMITERS. Kemudian coba lakukan copy ulang dengan mengikutkan DELIMITERS, seperti contoh di bawah ini:

```
 db_personal=>  COPY tes_copy FROM '/home/andhie/hasil.out';
```

```
    COPY
     db_personal=> SELECT * FROM tes_copy;
        nama    | state  | kota
    -------------+-- --+------
     Pamungkas         |          |
     Wulantini |          |
    (2 rows)
     db_personal=> DELETE FROM tes_copy;
    DELETE 2

     db_personal=>  COPY tes_copy FROM '/home/andhie/hasil.out' USING             DELIMITERS
'|';
    COPY

     db_personal=> SELECT * FROM tes_copy;
       nama          | state   |  kota
    ----------+-----+---------
     Pamungkas  | BD      | Cipanas
     Wulantini       | MD     | Madura
    (2 rows)
```

COPY Tanpa File

COPY dapat juga digunakan tanpa file, artinya pemindahan data menggunakan perintah COPY tidak mesti harus pada sebuah direktori dalam bentuk file. Ini dikarenakan kita dapat memindahkan data tersebut ke input dan output yang sama pada sebuah lokasi yang biasa digunakan oleh *psql.* Nama khusus dari lokasi tersebut adalah *stdin* yang merupakan representasi dari *input* psql dan *stdout* representasi dari *output* psql, lihat contoh di bawah ini:

```
     db_personal=> SELECT * FROM tes_copy;
       nama    | state   |  kota
    -------------+-----+---------
     Pamungkas |  BD     | Cipanas
     Wulantini |  MD    | Madura
    (2 rows)

     db_personal=> COPY tes_copy TO stdout USING DELIMITERS '|';
    Pamungkas|BD|Cipanas
    Wulantini|MD|Madura

     db_personal=> DELETE FROM tes_copy;
    DELETE 2

     db_personal=> SELECT * FROM tes_copy;
     nama   | state  | kota
    ------+----+------
    (0 rows)
```

Sebagai catatan pada penggunaan COPY FROM *stdin* kita harus memasukkan data sesuai dengan yang tercantum pada *stdout*, dan tanda (/.) artinya keluar dari COPY FROM *stdin*.

```
     db_personal=> COPY tes_copy FROM stdin USING DELIMITERS '|';
    Enter data to be copied followed by a newline.
    End with a backslash and a period on a line by itself.
    >> Pamungkas|BD|Cipanas
    >> Wulantini|MD|Madura
    >> \ .

     db_personal=> SELECT * FROM tes_copy;
       nama    | state   |  kota
    -------------+-----+---------
     Pamungkas |  BD     | Cipanas
```

```
 Wulantini |  MD    | Madura
(2 rows
```

MANAJEMEN POSTGRESQL

Bab 14 – Manajemen PostgreSQL

Deskripsi

Bab ini akan membahas pengelolaan server database PostgreSQL, bab ini akan lebih fokus pada pekerjaan administratif server dan beberapa tambahan utilitas yang belum dibahas pada bab-bab sebelumntya.

Obyektif

Pada bab ini diharapkan peserta dapat beberapa proses dalam pengelolaan server database seperti melakukan backup, restore atau cleaning up.

Outline

- Membuat User dan Group
- Membuat Database
- Konfigurasi Hak Akses
- Backup Database
- Restore Database
- Sistem Tabel
- Cleaning-Up
- Variabel

Explain

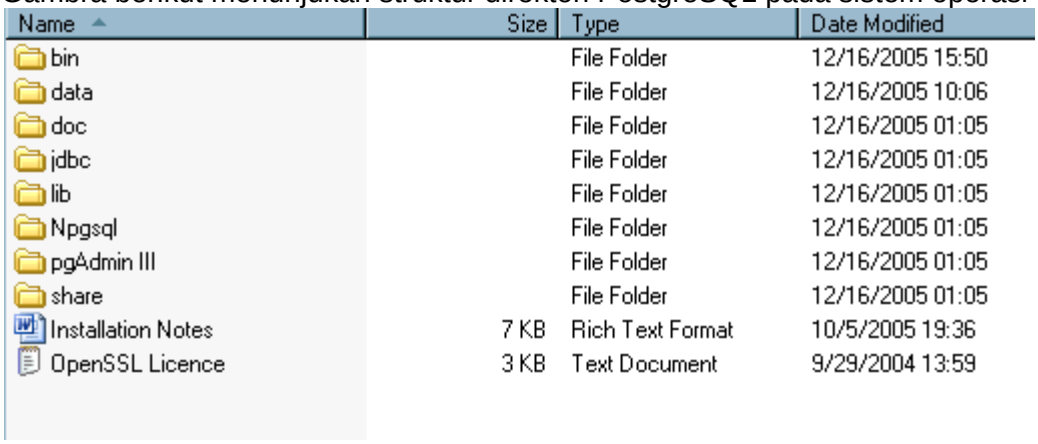
Pada bab ini akan dibahas secara singkat mengenai hal-hal yang berkaitan administrasi database server, di antaranya cara membuat *database*, *user, user group, security,* membackup data dan optimasi sebuah *database*.

File

Ketika PostgreSQL telaj terinstal, dengan sendirinya file-file postgres terbentuk di dalam *home* direktori khusus yaitu **var/lib/pgsql**. Namun, jika PostgreSQL diinstal dari *source code* **tar.gz** (tarball), file PostgreSQL terletak di direktori **usr/local/pgsql**. Direktori ini berisi semua file yang dibutuhkan oleh PostgreSQL yang masih dibagi lagi dalam berbagai subdirektori. Subdirektori tersebut antara lain.

/bin	Program *command-line*, seperti psql, createuser, dll.
/data	File konfigurasi dan berbagai informasi semua *database*.
/data/base	Subdirektori untuk setiap database.
/doc	Dokumentasi PostgreSQL.
/include	Berisikan file library yang dibutuhkan oleh PostgreSQL untuk melakukan koneksi dengan berbagai aplikasi yang menggunakannya.
/lib	Berisikan librari yang digunakan oleh berbagai bahasa pemrograman.
/man	Manual page PostgreSQL .

Sedangkan dalam instalasi di windows file akan diletakan pada direktori C:\Program Files\PostgreSQL\ Versi. Sedangkan struktur file yang digunakan tidak jauh berbeda. Gambra berikut menunjukan struktur direktori PostgreSQL pada sistem operasi windows.



Name	Size	Type	Date Modified
bin		File Folder	12/16/2005 15:50
data		File Folder	12/16/2005 10:06
doc		File Folder	12/16/2005 01:05
jdbc		File Folder	12/16/2005 01:05
lib		File Folder	12/16/2005 01:05
Npgsql		File Folder	12/16/2005 01:05
pgAdmin III		File Folder	12/16/2005 01:05
share		File Folder	12/16/2005 01:05
Installation Notes	7 KB	Rich Text Format	10/5/2005 19:36
OpenSSL Licence	3 KB	Text Document	9/29/2004 13:59

Gambar Struktur File Database PostgreSQL pada Windows

Membuat User dan Group

Kita dapat membuat sebuah user database baru pada postgres dari user Linux, dan ini hanya dapat dilakukan oleh postgres superuser. Setiap pembuatan user, harus ditentukan juga rule terhadap user yang bersangkutan. Rule tersebut misalnya apakah user yang akan

dibuat juga dapat membuat user baru dan apakah juga dapat membuat database.
Dari user Linux (root), ketikkan *su postgres* untuk login ke postgres super user.

```
#  su postgres
[postgres@localhost root]$  cd
[postgres@localhost pgsql]$
```

Kemudian, buatlah user baru yang diinginkan.

```
[postgres@ localhost pgsql]$  createuser coba1
Shall the new user be allowed to create databases? (y/n) y
Shall the new user be allowed to create more new users? (y/n) n
CREATE USER
```

Sedangkan apabila dilakukan dari prompt psql di Windows, bisa langsung di ketikan perintah createuser-nya.

Selain dari luar psql perintah pembuatan user juga dapat dilakukan dari prompt psql. Untuk bisa membuat user dari psql maka kita harus login sebagai postgres atau user yang mempunyai privileges untuk membuat user. Database yang digunakan untuk membuat user adalah database *template1* yang telah disediakan postgres, sehingga untuk melakukannya harus login ke dalam database *template1*.

```
[postgres@localhost pgsql]$ psql template1
Welcome to psql 8.1.1, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit
template1=#
```

Berikut kita akan membuat sebuah user dilengkapi dengan password serta izin pembuatan database pada user tersebut. Agar password-nya berfungsi, File *pg_hba.conf* yang terletak dalam direktori ***/***data perlu dikonfigurasi.

```
template1=# CREATE USER tes2 WITH password 'bintang' CREATEDB;
CREATE USER
```

Gunakan perintah **\h CREATE USER** untuk mendapatkan keterangan yang lebih lengkap tentang pembuatan user baru.

```
template1=# \h CREATE USER
Command:     CREATE USER
Description: Creates a new database user
Syntax:
CREATE USER username
    [ WITH
     [ SYSID uid ]
     [ PASSWORD 'password' ] ]
    [ CREATEDB   | NOCREATEDB ] [ CREATEUSER | NOCREATEUSER ]
    [ IN GROUP     groupname [, ...] ]
    [ VALID UNTIL  'abstime' ]
```

Untuk memastikan apakah user **coba1** dan **tes2** telah aktif, lakukan perintah dibawah ini.

```
template1=> \c template1 coba1
You are now connected to database template1 as user coba1.

template1=> \c template1 tes2
You are now connected to database template1 as user tes2.
```

User group merupakan tempat berkumpul dari user-user yang terdapat dalam PostgreSQL, artinya kita dapat mengelompokkan beberapa user ke dalam sebuah user group. Pengelompokkan ini berguna untuk kemudahan pemberian otoritas (GRANT dan REVOKE). Berikut contoh pembuatan user group.

```
template1=# CREATE GROUP contoh ;
CREATE GROUP
```

Kemudian, kita dapat memasukkan satu atau lebih user ke dalam sebuah user group, berikut perintahnya.

```
template1=# ALTER GROUP contoh ADD USER coba1, tes2, andhi ;
ALTER GROUP
```

Membuat Database

Membuat sebuah database, dapat dilakukan dari sistem operasi Linux (superuser postgres) dan dari database *template1* yang telah disediakan postgres, database *template1* telah ada ketika postgres pertama kali diinstal. Gunakan perintah **CREATE DATABASE namadatabase** untuk membuat sebuah database dan untuk menghapusnya gunakan perintah **DROP DATABASE namadatabase**, kedua perintah ini berlaku di dalam prompt psql database **template1**. Sementara perintah membuat database baru dari sistem operasi prompt postgres adalah **createdb namadatabase** dan **dropdb namadatabase** untuk menghapusnya. Untuk lebih detail lihat *manual page* **createdb** dan **dropdb**.
Contoh pembuatan sebuah database dari user Linux (postgres superuser), langkah pertama login ke superuser postgres dari user root.

```
[root@localhost root]# su postgres
[postgres@ localhost root]$
```

Langkah berikut buatlah database baru dengan nama *tesdb*

```
[postgres@localhost root]$ createdb tesdb
CREATE DATABASE
```

Kemudian, gunakan *dropdb* untuk menghapus lagi database tersebut

```
[postgres@localhost root]$ dropdb tesdb
DROP DATABASE
```

Berikut contoh membuat database dari prompt psql database **template1**, langkah pertama login ke dalam database *template1*.
Dari prompt postgres gunakan **psql** untuk login ke database **template1**, berikut perintahnya.

```
[postgres@localhost root]$ psql template1
Welcome to psql 8.1.1, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit
template1=#
```

Setelah berhasil login gunakan perintah *create database* untuk membuat database baru.

```
template1=# CREATE DATABASE tesdb ;
CREATE DATABASE
```

Kemudian, coba hapus lagi database *tesdb*

```
template1=# DROP DATABASE tesdb ;
DROP DATABASE
```

Konfigurasi Hak Akses

PostgreSQL memberikan keleluasaan pengguna untuk melakukan pengontrolan terhadap pengaksesan database. Kita juga dapat membatasi IP address yang mengakses database PostgreSQL, dan memberi password pada setiap user database. Seluruh kegiatan ini dapat dikonfigurasi melalui file **pg_hba.conf** yang terletak pada direktori **/data** dalam home direktori postgres. Jika database postgres yang pembaca miliki diperoleh dengan cara menginstal paket file RPM, direktori **/data** terletak pada **/var/lib/pgsql**, tetapi jika diperoleh dengan cara mengkompilasi paket tarball, direktori **/data** terletak di dalam direktori **/usr/local/pgsql**.

Pada konfigurasi lokal, secara default PostgreSQL membiarkan seluruh user untuk mengakses database tanpa diharuskan mengisi password pada masing-masing database, ini dilakukan dengan cara memberi opsi **trust**. Berikut contohnya.

```
local        all                                                  trust
host         all         127.0.0.1       255.255.255.255          trust
```

Tambahkan opsi **password** jika ingin agar semua database ketika login harus mengisikan password terlebih dahulu. Ini berlaku hanya pada komputer (alamat) lokal, bukan pada **host**. Alamat host 127.0.0.1 merupakan **localhost**, ini akan berfungsi jika ingin mengakses database postgres menggunakan tool **pgaccess**.

```
local        all                                                  password
host         all         127.0.0.1       255.255.255.255      trust
```

Kita juga dapat memilih database mana yang ingin diberi password, ini dapat dilakukan dengan cara menambahkan nama database yang ingin diberi izin akses (password). Berikut contohnya, misalkan kita menginginkan agar database **coba** diberi password.

```
local        all                                                  trust
local       db_personal                                           password
host         all         127.0.0.1       255.255.255.255       trust
```

Jika kita ingin agar database yang dimiliki dapat diakses oleh komputer lain (client) dengan IP address 192.168.7.1 sampai pada batas maksimum IP address 255.255.255.255 maka tambahkan baris berikut ini.

```
host         all         192.168.7.1       255.255.255.255   trust
```

Agar proses otorisasi atau izin akses pada seluruh client dijalankan ketika mengakses database postgres dari komputer database server, tambahkan opsi **password** seperti terlihat di bawah ini.

```
host         all         192.168.7.1       255.255.255.255   password
```

Backup Database

PostgreSQL menyediakan sebuah utiliti untuk membackup database PostgreSQL, sehingga jika database mengalami kerusakan atau lainnya, dapat di restore kembali. Utiliti yang ddigunakan adalah **pg_dump**, database yang di-backup berupa semua objek seperti tabel, isi tabel, view, fungsi, procedure, rule dan lainnya. Kita juga dapat mem-backup secara keseluruhan database yang terdapat pada database PostgreSQL menjadi sebuah file, ini dapat dilakukan dengan utiliti **pg_dumpall**. Data yang di-backup keluar dari database

PostgreSQL dengan menggunakan **pg_dump** tidak akan mengalami perubahan apapun artinya data tersebut masih tetap dalam bentuk script SQL.
Berikut contohnya, misalkan kita ingin mem-backup database coba dari prompt postgres, *satu* adalah nama database-nya, *tmp* merupakan nama direktori yang akan dituju untuk menyimpan database *satu*, *satu.dump* adalah nama file dari database *satu*.
Gunakan perintah di bawah ini untuk melakukan proses *dump*.

```
[postgres@localhost root]$ pg_dump satu > /tmp/satu.dump
```

Kemudian hapuslah database *satu* dan buatlah database baru.

```
postgres@localhost root]$ dropdb satu
DROP DATABASE

[postgres@localhost root]$ createdb satu
CREATE DATABASE
```

Restore Database

Setelah dibackup maka untuk mengembalikan kembali (restore) dapat digunakan perintah psql, karena tidak ada utiliti khusus untuk melakukan restore. Untuk mengembalikan (restore) database *satu* pada database postgres dapat menggunakan **psql** dari prompt **root**, berikut contohnya.

```
[postgres@localhost root]$ psql satu < /tmp/satu.dump
```

Selain dari prompt postgres, kita juga dapat mengembalikan database *satu* menggunakan prompt *psql* (dari dalam database postgres) seperti contoh berikut. Perintah yang digunakan adalah \i yang diikuti dengan lokasi dan nama file bakcup.

```
$ psql satu rofiq
Welcome to psql 8.1.1, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit
 db_personal=>
```

Berikut perintahnya.

```
 db_personal=> \i  /tmp/satu.dump
```

Utilitas **pgdump_all** juga dapat digunakan apabila akan meng-upgrade PostgreSQL karena sebelum upgrade diperlukan proses backup data sehingga bisa di restore setelah instalasi PostgreSQL diupgrade. Proses backup tidak perlu dilakukan satu per satu pada setiap databasenya, sebab utiliti *pg_dumpall* dapat digunakan untuk mem-backup atau memindahkan secara keseluruhan semua database yang terdapat di dalamnya ke sebuah direktori. Berikut adalah contoh melakukan dump semua database, yang kemudian akan di restore dengan contoh perintah berikutnya,

```
[postgres@localhost root]$  pg_dumpall > /tmp/db.out
```

R*estore* kembali file *db.out* ke dalam database *tempalte1*.

```
[postgres@localhost pgsql]$ psql -e template1 < /tmp/db.out
```

Sistem Tabel

Data yang terdapat pada PostgreSQL disimpan dalam tabel sistem postgres, nama depan

dari tiap tabel tersebut dimulai dengan ***pg***_. Tabel tersebut berisi informasi tentang tipe data, fungsi, operator, database, user, dan group. Tabel *pg_log* merupakan sebuah file binari dan bukan sebuah tabel yang nyata, tabel *pg_shodaw* berisi password user dan tidak terlihat pada user biasa. Sebagian besar sistem tabel dihubungkan menggunakan *object identification number* OIDs. Untuk menampilkkan semua tabel sistem pada prompt *psql* dapat digunakan perintah **\sD**.

Nama	Content
pg_aggregate	Aggregate
pg_attribute	Colum
pg_class	Tabel
pg_database	Database
pg_description	Komentar
pg_group	Group
pg_index	Index
pg_log	Status transaksi
pg_operator	Operator
pg_proc	Fungsi
pg_rewrite	Rule dan view
pg_shadow	User
pg_trigger	Trigger
pg_type	Type

Cleaning-Up

Ketika postgres melakukan aktivitas *query* dan *update* sebuah *row*, maka akan meninggalkan sisa sampah dalam direktori *database* tersebut. PostgreSQLmenyediakan perintah VACUUM untuk membersihkan direktori *database* tersebut dari sampah tadi, perintah VACUUM dapat dijalankan pada waktu tertentu sesuai keinginan. Ada dua perintah atau cara untuk menjalankan *vacuum*. Pertama jika kita ingin membersihkan sebuah tabel maka perintahnya *VACUUM nama_tabel* dan yang kedua jika ingin membersihkan secara menyeluruh atau sekaligus semua tabel, perintahnya *VACUUM;*. Untuk menampilkan prosesnya gunakan perintah *VERBOSE.*

Berikut contohnya.

```
    db_personal=# vacuum ;
```

Tambahkan opsi *verbose* untuk melihat proses yang sedang berlangsung.

```
    db_personal =# vacuum verbose;
    NOTICE:  --Relation pg_type--
    NOTICE:  Pages 3: Changed 0, reaped 2, Empty 0, New 0; Tup 170: Vac 11, Keep/VTL 0/0,
Crash 0, UnUsed 0, MinLen 106, MaxLen 109; Re-using: Free/Avail. Space 5040/492;
EndEmpty/Avail. Pages 0/1. CPU 0.00s/0.00u sec.
    NOTICE:  Index pg_type_oid_index: Pages 2; Tuples 170: Deleted 11. CPU 0.00s/0.00u sec.
    NOTICE:  Index pg_type_typname_index: Pages 2; Tuples 170: Deleted 11. CPU 0.00s/0.01u
sec.
    NOTICE:  Rel pg_type: Pages: 3 --> 3; Tuple(s) moved: 4. CPU 0.00s/0.00u sec.
    NOTICE:  --Relation pg_attribute--
    NOTICE:  Pages 16: Changed 6, reaped 7, Empty 0, New 0; Tup 1059: Vac 128, Keep/VTL 0/0,
Crash 0, UnUsed 0, MinLen 98, MaxLen 98; Re-using: Free/Avail. Space 13220/13220;
EndEmpty/Avail. Pages 0/7. CPU 0.00s/0.01u sec.
    ............
```

```
    db_personal=> VACUUM identitas;
    VACUUM

    db_personal=> VACUUM VERBOSE teman;
    NOTICE:  --Relation teman--
    NOTICE:  Pages 1: Changed 0, reaped 1, Empty 0, New 0; Tup 7: Vac 0, Keep/VTL 0/0, Crash
0, UnUsed 18, MinLen 128, MaxLen 168; Re-using: Free/Avail. Space 6936/0; EndEmpty/Avail.
Pages 0/0. CPU 0.00s/0.00u sec.
    NOTICE:  Index teman_idx: Pages 2; Tuples 7: Deleted 0. CPU 0.00s/0.00u sec.
    VACUUM
    db_personal=>
    db_personal=> -- perintah di bawah ini untuk membersihkan seluruh sampah pada direktory
database db_personal

    db_personal=> VACUUM VERBOSE;
    NOTICE:  --Relation aray--
    NOTICE:  Pages 1: Changed 0, reaped 1, Empty 0, New 0; Tup 2: Vac 0, Keep/VTL 0/0,
    .
    .
    .
    Crash 0, UnUsed 0, MinLen 56, MaxLen 56; Re-using: Free/Avail. Space 0/0; EndEmpty/Avail.
Pages 0/0. CPU 0.00s/0.00u sec.
    VACUUM
    db_personal=>
```

Gunakan *vacuum analyze* untuk mempercepat hasil query, sebab perintah ini akan membuat statistik yang akan dimanfaatkan oleh *query optimizer*:

```
    db_personal =# vacuum verbose analyze;
    NOTICE:  --Relation pg_type--
    NOTICE:  Pages 3: Changed 0, reaped 1, Empty 0, New 0; Tup 170: Vac 0, Keep/VTL 0/0, Crash
0, UnUsed 11, MinLen 106, MaxLen 109; Re-using: Free/Avail. Space 4996/0; EndEmpty/Avail.
Pages 0/0. CPU 0.00s/0.00u sec.
    NOTICE:  Index pg_type_oid_index: Pages 2; Tuples 170: Deleted 0. CPU 0.00s/0.00u sec.
    NOTICE:  Index pg_type_typname_index: Pages 2; Tuples 170: Deleted 0. CPU 0.00s/0.00u sec.
    NOTICE:  Analyzing...
    NOTICE:  --Relation pg_attribute--
    NOTICE:  Pages 14: Changed 0, reaped 1, Empty 0, New 0; Tup 1059:

    .......
```

Antarmuka Pemrograman PostgreSQL

Psql sangat bagus dan ideal untuk memasukkan perintah interaktif SQL dan untuk menjalankan *script* secara otomatis pasa PostgreSQL. Antarmuka bahasa pemrograman yang didukung PostgreSQL dapat dilihat pada tabel berikut;

Interface	Bahasa	Proses	Keuntungan
LIBPQ	C	Dikompile	Native Interface
LIBPGEASY	C	Dikompile	Meneyederhanakan C
ECPG	C	Dikompile	ANSI embedded SQL C
LIBPQ++	C++	Dikompile	Object-oriented C
ODBC	ODBC	Dikompile	Konektivitas aplikasi
JDBC	Java	Kompile dan interpreted	Portability
PERL	Perl	Di artikan / diterjemahkan	Text processing

Interface	Bahasa	Proses	Keuntungan
PGTCLSH	TCL/TK	Di artikan / diterjemahkan	Interfacing dan window
PYTHON	Python	Di artikan / diterjemahkan	Object-oriented
PHP	HTML	Di artikan / diterjemahkan	Web pages dinamis

Psql

Psql merupakan *query tool* yang digunakan PostgreSQLuntuk memulai atau bekerja dengan *database* Postgres. Pada bagian ini akan dijelaskan tentang psql.

Perintah Query Buffer

Perintah edit (\e) pada psql membebaskan user melakukan editing pada *query buffer*, perintah ini akan memindahkan isi dari *query buffer* ke dalam dafault editor. Ketika sebuah perintah atau *query* dieksekusi maka secara otomatis *query* tersebut akan tersimpan pada direktori */tmp/psql.* Untuk menghapus atau mengeditnya kita gunakan perintah \e.

Fungsi	Command	Argumen
Print	\p	
Execute	\g atau ;	File atau | perintah
Quit	\q	
Clear	\r	
Edit	\e	file
Backslash help	\?	
SQL help	\h	topic
File Include	\i	file
Output ke file atau perintah	\o	File atau | perintah
Menulis buffer ke file	\w	file
Show/save history query	\s	file
Menjalankan subshell	\!	perintah

Perintah Umum (General Command)

Operation	Perintah
Connect ke datanase lain	\connect *namadatabase*
Copy file tabel ke/dari database	\copy *namatabel* to | from *namafile*
SET variabel	\set variabel atau \set variabel value
Unset variabel	\unset variabel
Set format output	\pset *option* atau \pset *option value*
Echo	\echo *string* atau \echo `command`
Echo ke \o output	\gecho *string* atau \gecho `command`
Copyright	\copyright
Memilih karakter encoding	\encoding *newencoding*

Opsi Format Output

Perintah *pset* mengontrol format output yang digunakan oleh *psql* dan hanya menampilkan data *row.* Tabel di bawah ini merupakan daftar format perintah yang tersedia. Berikut tabel dan contoh penggunaannya.

Format	Parameter	Option

Format	Parameter	Option
Field alignment	Format	Tidak diluruskan, diluruskan, html, atau latex
Field separator	fieldsep	separator
Satu field per baris	expanded	
Row only	tuples_only	
Row separator	recordsep	separator
Tabel title	title	title
Tabel border	border	0,1, atau 2
Menampilkan value NULL	null	null_string
Tabel HTML tags (label)	tableattr	tags
Page output	pager	command
Format	**Parameter**	**Option**
Field alignment	Format	Tidak diluruskan, diluruskan, html, atau latex
Field separator	fieldsep	separator
Satu field per baris	expanded	
Row only	tuples_only	
Row separator	recordsep	separator
Tabel title	title	title
Tabel border	border	0,1, atau 2
Menampilkan value NULL	null	null_string
Tabel HTML tags (label)	tableattr	tags
Page output	pager	command

Berikut ini beberapa contoh dari perintah *pset*:

penggunaan perintah \pset untuk format *row only*

```
 db_personal=> SELECT * FROM anggota;
 id  |      nama
- -+---------
 26 | ANDHIE
 83 | RWATIA
(2 rows)

 db_personal=> SELECT NULL;
 ?column?
----------
(1 row)

 db_personal=> \pset tuples_only
Showing only tuples.

 db_personal=> SELECT * FROM anggota;
 26 | ANDHIE
 83 | RWATIA

 db_personal=> SELECT NULL;  -- hasilnya kosong

 db_personal=> \pset null '(anggota)'
Null display is '(anggota)'.

 db_personal=> SELECT * FROM anggota;
 26 | ANDHIE
```

```
 83 | RWATIA

 db_personal=> SELECT NULL;
 (anggota)
```

berikut ini contoh \pset untuk format tabel border

```
 db_personal=> SELECT * FROM anggota;
 id  |      nama
----+--------------
 26 | ANDHIE
 83 | RWATIA
(2 rows)

 db_personal=> \pset border 2
Border style is 2.

 db_personal=> SELECT * FROM anggota;
+----+---------------+
 | id  |    nama       |
+----+---------------+
 | 26 | ANDHIE     |
 | 83 | RWATIA      |
+- -+----------------+
(2 rows)
```

Variabel

Sebuah variabel dapat di-SET dengan perintah *backslash*-set (\set), dan untuk menghapusnya lagi gunakan perintah *backslash-unset* (\unset). Berikut tabel dan contohnya:

Format	Perintah	Argumen
Field alignment	\a	
Field separator	\f	Separator
Satu field per baris	\x	
Row only	\t	
Tabel title	\C	Title
Enable HTML	\H	
Tabel HTML tags	\T	Tags

Berikut ini contoh penggunaan variabel *psql*:

```
 db_personal=> \set var_no 4
 db_personal=> SELECT :var_no;
 ?column?
---------------
        4
(1 row)

 db_personal=> \set operator SELECT
 db_personal=> :operator :var_no;
 ?column?
---------------
        4
(1 row)

 db_personal=> \set var_string '\'Hallo World\''
 db_personal=> \echo :var_string
'Hallo World'

 db_personal=> SELECT :var_string;
```

```
  ?column?
------------------
 Hallo World
(1 row)

 db_personal=> \set var_date `date`
 db_personal=> \echo :var_date
Tue Mar 26 08:17:14 WIT 2002

 db_personal=> \set var1_date '\''`date`'\''
 db_personal=> \echo :var1_date
'Tue Mar 26 08:19:29 WIT 2002'

 db_personal=> SELECT :var1_date;
           ?column?
----------------------------------------
 Tue Mar 26 08:19:29 WIT 2002
(1 row)
```

Explain

Sebagai sebuah database handal POstrgreSQL juga menyediakan tool untuk merencanakan query. Perencanaan query sangat diperlukan untuk menangani database yang besar, karena struktur query akan sangat berpengaruh terhadap kinerja database dan aplikasi yang menggunakannya. Dengan menggunakan EXPLAIN dapat dilihat proses apa yang dilakukan oleh sebuah query dan melihat waktu yang diperlukan untuk menjalankannya. Apabila query yang dijalankan semakin kompleks, maka informasi yang diberikan oleh explain juga akan semakin komplek.

Berikut adalah contoh penggunaan EXPLAIN;

```
db_personal=> explain select * from tbl_personal;
                         QUERY PLAN
------------------------------------------------------------
 Seq Scan on tbl_personal  (cost=0.00..1.06 rows=6 width=61)
```

Hasil dari EXPLAIN tersebut adalah berupa informasi yang dapat di jelaskan sebagai berikut;

- Estimasi start-up cost (cost yang digunakan sebelum hasil output ditampilkan, misalnya cost yang digunakan untuk melakukan sorting, dll)
- Estimasi total cost (cost total yang digunakan untuk melakukan seleksi semua row)
- Estimasi jumlah row yang di seleksi
- Estimasi rata-rata lebar row (dalam byte) yang diseleksi.

Cost dalam hal ini merupakan ukuran unit dalam disk-page yang di gunakan, dimana 1.0 sama dengan pembacaan satu sequensial disk-page.

Berikut adalah contoh EXPLAIN untuk query yang lebih kompleks;

```
db_personal=> explain select "txt_NamaDepan" From tbl_Personal UNION Select "txt
_Pekerjaan" from tbl_personal as Nama;
                                    QUERY PLAN

------------------------------------------------------------------------------
----
 Unique  (cost=2.46..2.52 rows=12 width=24)
   ->  Sort  (cost=2.46..2.49 rows=12 width=24)
         Sort Key: "txt_NamaDepan"
         ->  Append  (cost=0.00..2.24 rows=12 width=24)
               ->  Seq Scan on tbl_personal  (cost=0.00..1.06 rows=6 width=14)
```

```
                ->  Seq Scan on tbl_personal nama  (cost=0.00..1.06 rows=6 width=
24)
```