Tutorial PostgreSQL
Basis Data
CSF2600700
Semester Genap 2018/2019

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia
FACULTY OF COMPUTER SCIENCE

Note!

Bacalah dengan teliti agar tidak ada yang terlewat. Dokumen yang dikumpulkan adalah ***capture* SQL *query* beserta *output* sesuai soal pada bagian percobaan (*indexing*) dan Latihan (*trigger*)** . Pastikan **terdapat username SSO** pada *capture*-nya.

Tutorial ini menggunakan skema basis data SIKIRIM **hasil kelanjutan dari tutorial 1**. Jangan lupa **set search path** terlebih dahulu.

I. Index dan View

A. View

View merupakan sebuah tabel virtual yang dibuat menggunakan SQL *query*. View tidak disimpan dalam *database* seperti halnya *base relation*.
Berikut ini sintaks untuk dapat membuat sebuah *view*.

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] VIEW name
[ ( column_name [, ...] ) ] AS query
```

Untuk membuat *view* yang dapat menampilkan daftar pengguna beserta idnya, kita bisa menjalankan perintah berikut.

```
CREATE VIEW daftar_customer AS
SELECT id_customer, nama_customer
FROM CUSTOMER;
```

Setelah *view* berhasil dibuat, kita dapat melakukan *query* menggunakan view tersebut seperti halnya *query* pada *base relation*.
Misalnya, perintah *select* ke *view* daftar customer sebagai berikut.

```
SELECT * FROM daftar_customer;
```

View biasanya hanya untuk penyimpanan sementara. Untuk menghapusnya, kita bisa menggunakan perintah berikut.

```
DROP VIEW daftar_customer;
```

B. Indexing

Dalam sebuah basis data, *index* bertujuan untuk melakukan pencarian, terutama pada tabel dengan jumlah data yang sangat banyak sehingga saat kita mencari sebuah data berdasarkan kolom tertentu, data tersebut akan lebih mudah untuk ditemukan. Secara *default*, *primary key* merupakan sebuah *index* dalam basis data yang diinisiasi sebagai kolom kunci dalam sebuah tabel.
Pembuatan *index* dapat dilakukan dengan format sintaks berikut.

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ name ] on
table [ USING method ] ( { column | ( expression ) }
[ COLLATE collation ] [ opclass ] [ ASC | DESC ]
[ NULLS { FIRST | LAST } ] [, ... ] )
  [ WITH (storage_parameter = value [, ...] ) ]
    [ TABLESPACE tablespace ]
    [ WHERE predicate ];
```

Keterangan:

1. Tanda [ ] menyatakan pilihan, boleh tidak digunakan
2. Tanda | menyatakan pilihan yang dapat digunakan
3. Keterangan lebih lanjut bisa ditemukan pada tautan berikut: http://www.postgresql.org/docs/9.1/static/sql-createindex.html

Kita bisa mencoba membuat *index* untuk tabel CUSTOMER berdasarkan namanya. Jika tidak memberi spesifikasi apapun, *index* yang dibuat akan menggunakan *method* btree dan diurutkan secara *ascending*.

```
CREATE INDEX customer_name_idx ON CUSTOMER(nama_customer);
```

Mari kita buat *index* untuk tabel MITRA berdasarkan id. Pada contoh ini, kita menggunakan *method* hash.

```
CREATE INDEX mitra_nama_idx ON MITRA using hash (nama_mitra);
```

Index komposit adalah *index* dengan dua atau lebih kolom dalam suatu tabel. Kali ini, kita buat *index* komposit untuk tabel TRACKING berdasarkan id_tracking dan nomor_resi yang akan diurutkan secara *descending*.

```
CREATE INDEX tracking_idResi_idx
ON TRACKING (id_tracking, nomor_resi DESC);
```

Untuk satu tabel, kita dapat memiliki lebih dari satu *index*. Sebagai contoh, berikut ini dua *index* yang berbeda untuk tabel MITRA.

```
CREATE INDEX mitra_nama_idx ON MITRA (nama_mitra);
CREATE INDEX mitra_alamat_idx ON MITRA (alamat_mitra);
```

Lihatlah apakah *index* tersebut sudah ada di tabel CUSTOMER, MITRA, dan TRACKING. Untuk melihat *index* yang ada pada suatu tabel, kita dapat menjalankan perintah berikut.

```
\d table_name;
```

Untuk menghapus semua *index* yang telah dibuat sebelumnya, kita dapat menjalankan perintah berikut.

```
DROP INDEX [ IF EXISTS ] name;
```

Sebagai contoh, perintah berikut akan menghapus *index* dari tabel CUSTOMER.

```
DROP INDEX customer_name_idx;
```

C. Explain

EXPLAIN adalah perintah yang dapat digunakan untuk menampilkan estimasi eksekusi dari sebuah *query*. Berikut ini cara menggunakannya.

```
EXPLAIN [ ANALYZE ] statement;
```

Keterangan: *statement* adalah *query* yang ingin diketahui eksekusinya.

Bagian terpenting dari data yang ditampilkan adalah *execution cost* yang merupakan estimasi waktu yang diperlukan untuk menjalankan *query* tersebut. Adanya opsi ANALYZE membuat *query* dijalankan. Hal ini berguna untuk membandingkan waktu estimasi dengan waktu sebenarnya.

Sebagai contoh, jalankan perintah berikut.

```
EXPLAIN ANALYZE
SELECT * FROM CUSTOMER
WHERE nama_customer LIKE '%Alan%';
```

Hasil eksekusi *query*-nya akan seperti di bawah ini. *Execution time* menunjukkan waktu yang dibutuhkan untuk mengeksekusi *query* tersebut.

```
izzan.fakhril=> EXPLAIN ANALYZE
izzan.fakhril-> SELECT * FROM CUSTOMER
izzan.fakhril-> WHERE nama_customer LIKE '%Alan%';
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on customer  (cost=0.00..12.88 rows=2 width=322) (actual time=0.028..0.033 rows=1 loops=1)
   Filter: ((nama_customer)::text ~~ '%Alan%'::text)
   Rows Removed by Filter: 13
 Planning time: 0.108 ms
 Execution time: 0.068 ms
(5 rows)
```

D. Percobaan

Diberikan 4 buah *query* sebagai berikut.

```
SELECT * FROM MITRA ORDER BY id_kota DESC;
```

```
SELECT * FROM CUSTOMER WHERE alamat LIKE '%Depok%';
```

```
SELECT * FROM SHIPPING ORDER BY nama_penerima ASC;
```

```
SELECT * FROM CUSTOMER ORDER BY nama_customer DESC;
```

1. Jalankan perintah **`EXPLAIN ANALYZE`** dengan statement berupa keempat *query* di atas. Simpan hasil yang ditampilkan pada laporan Anda.

2. Buatlah *index* berikut:
   a. `nama_mitra_idx` pada tabel `MITRA` kolom `nama_mitra`
   b. `jenis_mitra_idx` pada tabel `MITRA` kolom `jenis_mitra`
   c. `nama_kota_idx` pada tabel `KOTA` kolom `nama_kota`
   d. `nomor_resi_idx` pada tabel `TRACKING` kolom `nomor_resi`

   Tampilkan *index* untuk setiap tabel dalam laporan Anda.

3. Jalankan kembali keempat *query* di atas menggunakan perintah **`EXPLAIN ANALYZE`**. Simpan hasil yang ditampilkan pada laporan Anda.

4. Bandingkan *cost time* jika tidak menggunakan index dan jika menggunakan index. Manakah yang lebih baik, menggunakan *index* atau tidak menggunakan index? Berikan penjelasan.

Tutorial PostgreSQL
Basis Data
CSF2600700
Semester Genap 2018/2019

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia
FACULTY OF COMPUTER SCIENCE

II. Trigger dan Stored Procedure

Apa itu *stored procedure* dan *function*?

Stored procedure dan *function* (atau *user-defined function*) merupakan **bagian dari bahasa prosedural** di dalam SQL. Bahasa ini biasa disebut PL/SQL. Di dalam PostgreSQL, bahasa ini disebut **PL/pgSQL**. Dengan keduanya, kita bisa menambahkan berbagai **elemen prosedural**, seperti *loop*, perhitungan kompleks, dan masih banyak lagi. Kita juga bisa mengembangkan fungsi yang kompleks yang **tidak bisa dicapai** dengan *statement* SQL biasa.

PL/pgSQL sudah mempunyai fitur *function*, akan tetapi *procedure* baru diperkenalkan di versi 11. Oleh karena itu, kita hanya akan memakai sintaks ***function***. Meskipun begitu, kita tetap dapat membuat *stored procedure* dengan sintaks *function*.

Bagaimana cara membuatnya?

Function dalam PostgreSQL dapat dibuat dengan sintaks berikut.

```
CREATE [ OR REPLACE ] FUNCTION
<schema_name>.<function_name> (
       [ [ <arg_mode_1> ] [ <arg_name_1> ] <arg_type_1> ],
       [ [ <arg_mode_2> ] [ <arg_name_2> ] <arg_type_2> ],
       [, ... ] )
[ RETURNS <return_type> ] AS
$$
      [ DECLARE
      <variable_name_1> <variable_type_1>;
      :
      :
      ]
      BEGIN
      -- <statements>
      END;
$$
LANGUAGE plpgsql;
```

Catatan:
`<arg_mode_1>` dapat diisi dengan `IN`, `OUT`, maupun `INOUT`

Sekarang, kita akan mencoba hitung berapa kali suatu kota di-*tracking* dan menyimpannya sebagai kolom baru. Mulailah dengan menambah kolom baru terlebih dahulu pada tabel KOTA.

```
ALTER TABLE KOTA ADD COLUMN jumlah_tracking INT DEFAULT 0;
```

Setelah itu, definisikan *function* yang akan menghitung jumlah *tracking* dari suatu kota. Contoh *function*-nya adalah sebagai berikut.

```
CREATE OR REPLACE FUNCTION
hitung_jumlah_tracking (masukan_id INTEGER)
RETURNS INTEGER AS
$$
    DECLARE
        jml_tracking INTEGER;
    BEGIN
        SELECT COUNT(T.*) INTO jml_tracking
        -- Ket: perintah INTO akan menyimpan
        -- hasil count ke variabel jml_tracking
        FROM TRACKING T JOIN KOTA K
            ON K.id_kota = T.id_kota
        WHERE K.id_kota = masukan_id;

        UPDATE KOTA K
            SET jumlah_tracking = jml_tracking
            WHERE K.id_kota = masukan_id;
        RETURN jml_tracking;
    END;
$$
LANGUAGE plpgsql;
```

Kita dapat menjalankan *function* yang sudah dibuat dengan sintaks berikut.

```
SELECT schema_name.function_name ([<arg1>, <arg2>, ...]);
```

Sebagai contoh, terapkan *function* ini pada baris di **KOTA** dengan id_kota = 27.

```
SELECT hitung_jumlah_tracking(27);
```

Setelah itu, lakukan *query* pada tabel **KOTA**.

```
=> SELECT * FROM KOTA ORDER BY jumlah_tracking DESC;
 id_kota |      provinsi       |             nama_kota             | jumlah_tracking
---------+---------------------+-----------------------------------+-----------------
      27 | Jawa Barat          | Depok                             |               4
       5 | Aceh                | Sabang                            |               0
       6 | Aceh                | Subulussalam                      |               0
       7 | Bali                | Denpasar                          |               0
-- More  --
```

Dapat dilihat bahwa baris dengan id_kota = 27 sudah di-*update* menjadi nilai bukan nol.

Untuk menerapkan *function* ini ke semua elemen di tabel KOTA, kita bisa melakukan *query* berikut.

```
SELECT hitung_jumlah_tracking(id_kota) FROM KOTA;
```

Dengan demikian, `hitung_jumlah_tracking()` akan diterapkan ke semua elemen di kolom KOTA. Selain *update* nilai, *function* `hitung_jumlah_tracking()` juga mengembalikan nilai sehingga kita akan bisa melihat hasil *query* berupa tabel dengan kolom `hitung_jumlah_tracking`.

Selain dengan cara tersebut, kita juga bisa melakukan *looping* di *function* untuk meng-*update* setiap baris di tabel KOTA. Berikut ini contoh *function*-nya.

```
CREATE OR REPLACE FUNCTION
hitung_jumlah_tracking_semua_kota ()
RETURNS void AS
$$
    DECLARE
        temp_row RECORD;
    BEGIN
        FOR temp_row IN
            SELECT
                K.id_kota  AS id_kota,
                COUNT(T.*) AS jml_tracking
            FROM TRACKING T
                JOIN KOTA K ON K.id_kota = T.id_kota
            GROUP BY K.id_kota
        LOOP
            UPDATE KOTA K
                SET jumlah_tracking = temp_row.jml_tracking
                WHERE K.id_kota = temp_row.id_kota;
        END LOOP;
    END;
$$
LANGUAGE plpgsql;
```

Kemudian, jalankan *function* tersebut seperti biasa.

```
SELECT hitung_jumlah_tracking_semua_kota();
```

Kita juga bisa melihat daftar *stored procedure* yang ada dengan perintah berikut.

```
\df
```

Untuk menghapus *function* yang sudah dibuat, kita dapat menjalankan perintah berikut.

```
DROP FUNCTION FUNCTION hitung_tiket_dipesan(INTEGER);
```

Tutorial PostgreSQL
Basis Data
CSF2600700
Semester Genap 2018/2019

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia
FACULTY OF COMPUTER SCIENCE

Apa itu *trigger*?

Trigger merupakan operasi pada sebuah tabel yang **otomatis dijalankan** ketika ada kejadian tertentu. Kejadian ini bisa berupa ketika melakukan INSERT, UPDATE, atau DELETE. Agar *trigger* bisa bekerja, kita perlu **membuat** *stored procedure* terlebih dahulu, baru kemudian **menyambungkan** suatu *trigger* dengan *stored procedure* tersebut ke suatu tabel.

Format sintaks PL/SQL yang digunakan untuk membuat *stored procedure* yang akan dipanggil *trigger* ini **sama** seperti function biasa. Akan tetapi, `<return_type>` harus **bernilai *trigger*** dan **tidak boleh mempunyai argumen**.

Mari kita coba tangani kasus untuk mencegah masuknya *tracking* yang statusnya **sudah "Delivered"**. Mulailah dengan membuat *function trigger*-nya dulu.

```
CREATE OR REPLACE FUNCTION cek_barang_dikirim()
RETURNS trigger AS
$$
    DECLARE
        id_status_dikirim SMALLINT;
        ada_status_dikirim BOOLEAN;
    BEGIN
        SELECT id_status INTO id_status_dikirim
        FROM STATUS S
        WHERE nama_status = 'Delivered';

        -- TG_OP: variabel untuk mengetahui apakah
        -- jenis operasinya itu INSERT, UPDATE,
        -- atau yang lainnya.

        IF (TG_OP = 'INSERT') THEN
            SELECT EXISTS(
                SELECT S.id_status
                FROM STATUS S JOIN TRACKING T
                    ON T.id_status = S.id_status
                WHERE T.nomor_resi = NEW.nomor_resi
                AND S.id_status = id_status_dikirim
            ) INTO ada_status_dikirim;

            IF ada_status_dikirim THEN
                RAISE EXCEPTION 'GALAT: Barang sudah dikirim!';
            END IF;
            RETURN NEW;
        END IF;
    END;
$$
LANGUAGE plpgsql;
```

Setelah itu, buat *trigger*-nya. Berikut in sintaks untuk membuat *trigger* di dalam PostgreSQL.

```
CREATE TRIGGER <trigger_name>
{ BEFORE | AFTER } { INSERT | UPDATE | DELETE [ OR ... ] }
ON <schema name>.<table name>
[FOR EACH {ROW | STATEMENT}]
EXECUTE PROCEDURE <function_name> ([ <arg1>, ...]);
```

Tanda kurung kurawal {} di atas menandakan bahwa kita harus memilih satu di antara pilihan yang tersedia.

Berdasarkan sintaks tersebut, kita dapat membuat *trigger* yang akan menjalankan *function* `cek_barang_dikirim()` **sebelum** elemen baru dimasukkan.

```
CREATE TRIGGER cek_barang_dikirim_trigger
BEFORE INSERT ON TRACKING
FOR EACH ROW
EXECUTE PROCEDURE cek_barang_dikirim();
```

Untuk memeriksa apakah *trigger* bekerja atau tidak, kita bisa menjalankan *event* yang memicu trigger tersebut. Sebagai contoh, coba masukkan suatu elemen yang sudah distatuskan "Delivered" ke tabel TRACKING.

```
INSERT INTO TRACKING VALUES
    ('89050', 'DPKR500476292617',
     '2016-11-25 14:45:00', 5, 16, 'Rani');
```

Amati apakah *trigger* tersebut berhasil (elemen tak ada di tabel TRACKING dan ada pesan *error*) atau gagal. Bandingkan bila tabel TRACKING dimasukkan data yang tidak akan menimbulkan *error*.

```
INSERT INTO TRACKING VALUES
    ('89052', 'SMG1B04681391516',
     '2016-11-26 14:45:00', 5 , 16, 'Rani Lagi');
```

Tutorial PostgreSQL
Basis Data
CSF2600700
Semester Genap 2018/2019

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia
FACULTY OF COMPUTER SCIENCE

Latihan

1. Buat atribut baru `jumlah_shipping` bertipe INT ke tabel CUSTOMER. Lalu, buatlah function untuk mengisi atribut tersebut. Atribut ini akan dipakai untuk menghitung berapa banyak *shipping* yang dimiliki seorang CUSTOMER.

   Setelah itu, jalankan *function* tersebut dan lihat perubahan yang terjadi pada tabel CUSTOMER.

2. Buatlah *trigger* untuk setiap kejadian INSERT, UPDATE, dan DELETE pada tabel SHIPPING yang akan memperbaharui kolom jumlah_shipping di tabel CUSTOMER.

   Sebagai contoh, jika dilakukan INSERT, `jumlah_shipping` otomatis bertambah.

   Lakukan uji coba dengan perintah-perintah berikut dan lihat perubahan dan perbedaan sebelum dan sesudah perintah dijalankan. Misalnya, tampilkan terlebih dahulu `jumlah_shipping` awal. Kemudian, lakukan operasi INSERT. Setelah itu, tampilkan `jumlah_shipping` setelah operasi INSERT tersebut. Lakukan pula untuk operasi UPDATE dan DELETE.

```
INSERT INTO SHIPPING
    (nomor_resi, id_kota_asal, id_kota_tujuan,
    id_jenis_barang, nilai_barang, berat,
    id_mitra, id_layanan, id_customer,
    nama_penerima, alamat_penerima, telp_penerima)
VALUES
    ('TEST123', 1, 1,
    'B3', 1000, 10,
    '12400', 'L2', '16914',
    'Pak Eko', 'Jagakarsa', '081082083084');
```

```
DELETE FROM SHIPPING WHERE id_customer = '16910';
```

```
UPDATE SHIPPING
SET    id_customer = '16909'
WHERE  id_customer = '16914';
```

3. Buatlah *trigger* untuk setiap event INSERT pada tabel SHIPPING yang akan mencegah pemasukan *shipping* jika identitas pelanggan (yaitu nama, nomor telepon, alamat) sama dengan identitas penerima dan kota asal juga sama dengan kota tujuan. Definisi identitas yang sama di sini adalah sama secara *case-insensitive*.

   Lakukan uji coba dengan perintah-perintah berikut

```
INSERT INTO SHIPPING
    (nomor_resi, id_kota_asal, id_kota_tujuan,
    id_jenis_barang, nilai_barang, berat,
    id_mitra, id_layanan, id_customer,
    nama_penerima, alamat_penerima, telp_penerima)
VALUES
    ('TEST321', 1, 1,
    'B3', 1000, 10,
    '12400', 'L2', '16914',
    'James Gonzales', 'Blitar', '082123456789');
```

Selamat Mengerjakan :)