SQL and DDL Tidbits May 14, 2013

DDL Statements
"--" signifies a comment

Defining a primary key
ALTER TABLE location ADD CONSTRAINT loc_pk PRIMARY KEY (lid);
- automatically defines columns in a primary key as NOT NULL
- max number of chars in constraint name = 63

Defining a foreign key
ALTER TABLE location ADD CONSTRAINT location_tz_fk FOREIGN KEY (tzone)
REFERENCES TimeZone(tzone) MATCH FULL;

Renaming a Database
ALTER DATABASE oldname RENAME TO newname;
- user must be owner of the database AND have CREATEUSER privilege

Gotcha
"CREATE TABLE tablename (column_name ..." generates syntax error
- extra space not allowed between "("and "column_name"

"Not Equals" Symbol

< > (no space between symbols)

!= is deprecated

Database Level Privileges
if user A creates a db, then user B automatically has access to it
- users A and B must be known to psql through the createuser command

Table Level Privileges
if user A creates a table, users have NO privileges on the table (unless they are granted)
(see Douglas pp741-742)

Date and Time Datatypes
SELECT CURRENT_TIMESTAMP generates current time

data type TIMESTAMP defaults to TIMESTAMP WITHOUT TIME ZONE

internally, date/times are in UTC

all date and time datatypes have precision of microseconds

DATE value = "08-01" is not valid

– must specify year to create valid DATE field

Date and Time Functions

```
select * from precip where obstime   >  'today';
        - today surrounded by single quotes
```

```
select * from TextProduct
        where EXTRACT (DAY from CURRENT_TIMESTAMP - postingtime) < 5;
```

To list all records in the Height table which were posted in the last 20 minutes:

```
SELECT * FROM Height WHERE obstime > (now()  -  interval  ’20  min’);
```

To insert a record containing a NULL DATE value through an INSERT statement:

```
INSERT INTO tablename VALUES(…,NULL,…);
```

In ecpg, recommend using dttoasc only. dttofmtasc function exists but does not work properly.

Two more examples:

```
select timestamp '2010-06-26 00:00:00' - timestamp  '2008-11-07 00:00:00';
 ?column?
----------
 596 days
(1 row)
```

```
select justify_interval(timestamp '2010-06-26 00:00:00' - timestamp  '2008-11-07 00:00:00');
   justify_interval
-----------------------
 1 year 7 mons 26 days
(1 row)
```

Column Types

INT2
FLOAT4
FLOAT8

FLOAT is the same as FLOAT8

`A cast from float to int rounds, it doesn't truncate.`

Column Naming

- column names must begin with a letter or underscore

- max of 64 char

JOINS
Postgres has LEFT/RIGHT/FULL OUTER joins available. It uses the ANSI standard format for OUTER join.
- many views in IHFS db contain the keyword OUTER
- Example:

```
CREATE VIEW locview (...
SELECT ...
FROM  location x0   left outer join  riverstat x1
ON x0.lid = x1.lid
WHERE  … ;
```

Statement Timing
SET STATEMENT_TIMEOUT TO nn;
- where nn = number of milliseconds
- server run-time configuration parameter
- value of 0 (default) turns off timer
- see Section 16.4.7.1 of PostGreSQL 7.4.7 Documentation

Cursors
EXEC SQL CLOSE statement closes the cursor and frees all resources related to the cursor

Postgres Version 8.3 has "WHERE CURRENT OF … " clause for cursors. This was available in Informix.

Must close a cursor before reopening it

Checking Query Plans
EXPLAIN SELECT … ;
EXPLAIN ANALYZE SELECT …;

If multiple indexes are defined on a table, the optimizer determines which index to use.
"A two-column index is bigger and hence more expensive to search than a one-column index --- perhaps quite substantially so …"

Character Strings
Character strings in SQL statements must be denoted by single quotes (')

Example:
psql: SELECT * FROM height WHERE lid = 'ABCD1';

To select lids from the Location table which begin with lower case characters:

```
SELECT  lid  FROM  Location  WHERE  lid SIMILAR TO ‘[a-z]%’ ;
```

To change all characters of a column’s values to upper case:

```
UPDATE  <tablename> SET  <columnname> TO UPPERCASE (<columnname>);
```

To list all Location table identifiers with all lower case characters:

```
SELECT  lid  FROM  Location  WHERE  lid  ~  (‘[a-z]’);
```

“~” is a POSIX regular expression operator similar to “LIKE” (See Section 9.6.3)

Select records from the HourlyPP table where any char of the hourly_qc field = D
(D signifies the result of a disaggregation)
(note that the hourly_qc field is defined as char (24))

```
SELECT  *  FROM  HourlyPP WHERE hourly_qc  ~  (‘D’);
```

Print only the first X chars of a char column (version 8.3 and later)

```
create table t1 (g char(10));
insert into t1 values ('abcdefg'),('hijklmno');

select overlay(g placing ' ' from 3 to 10);  -- display only the first 2 chars of column g
```

Temp tables
postgres allows the user to create a temp table with the same name as a real table - the temp table will "mask" the real table during the session - temp table is dropped at the end of the session - idea can be used for testing

Cascading Updates and Deletes
as part of a CREATE TABLE statement, a column can be defined as "ON UPDATE CASCADE" - this will cause updates to "cascade" from parent table to child table - can also be set up for deletes - see Momjian pp161, 162

Granting Superuser Priviledges

```
ALTER USER <username> WITH SUPERUSER
```

Schemas

To change a schema in psql:

```
set search_path to <schema_name>;
```

Locking Tables
postgres has a LOCK statement but has no UNLOCK statement - ending the transaction
unlocks the table

Dropping Roles

Execute “REASSIGN OWNED …” followed by “DROP OWNED…” to remove a role

Must be done for each database

NULLs

For unique indexes, NULLs are considered not equal to each other, and multiple NULL can be stored in a unique index:

```
CREATE TABLE uniqtest (x INTEGER);

CREATE UNIQUE INDEX i_uniqtest ON uniqtest (x);

INSERT INTO uniqtest VALUES (1), (NULL), (NULL);

SELECT * FROM uniqtest;
   x
--------
      1
 (null)
 (null)
```

Other Tidbits
Return the IP address of computer that postgres is running on

```
SELECT inet_server_addr();
```

Returning only the “first” 10 records of a SELECT:

```
SELECT  *  FROM … LIMIT  10;
```

Note that without an “order by” clause, this query is free to return any 10 records. Over the course of time, if additional inserts and deletes are done on the table, the query may return a different set of records.

Returning unique column values

```
SELECT  DISTINCT lid FROM CurPP;
```

Q: I have a table column I want to change from a boolean to a smallint changing false to 0 and true to 1. How do I do that?

```
A: ALTER TABLE ALTER col_name TYPE SMALLINT
  USING CASE WHEN col_name THEN 1 ELSE 0 END;
```

Changing the size of a column:
Until now, I was not familiar with any sensible mechanism to simply change the size in PG. But yesterday, Tom Lane himself suggested something ubercool in the list.

Let's assume for the sake of simplicity that your table is called "TABLE1" and your column is "COL1". You can find the size of your "COL1" column by issuing the following query on the system tables:

```
SELECT atttypmod FROM pg_attribute
WHERE attrelid = 'TABLE1'::regclass
AND attname = 'COL1';

atttypmod
-----------
24
(1 ROW)
```

This means that the size is 20 (4 is added for legacy reasons, we're told). You can now conveniently change this to a varchar(35) size by issuing this command:

```
UPDATE pg_attribute SET atttypmod = 35+4
WHERE attrelid = 'TABLE1'::regclass
AND attname = 'COL1';

UPDATE 1
```

Note that I manually added the 4 to the desired size of 35 again, for some legacy reasons inside PG. Done. That's it. Should we check?

```
\d TABLE1

TABLE "public.TABLE1"
COLUMN  |  TYPE                  | Modifiers
--------+------------------------+-----------
```

```
COL1    | CHARACTER VARYING(35) |
```

How can I get list of views that are using given column in table?

```
 SELECT distinct dependee.relname
 FROM pg_depend
 JOIN pg_rewrite ON pg_depend.objid = pg_rewrite.oid
 JOIN pg_class as dependee ON pg_rewrite.ev_class =
dependee.oid
 JOIN pg_class as dependent ON pg_depend.refobjid =
dependent.oid
 JOIN pg_attribute ON pg_depend.refobjid =
pg_attribute.attrelid
     AND pg_depend.refobjsubid = pg_attribute.attnum
 WHERE dependent.relname = <tablename>
 AND pg_attribute.attnum > 0
 AND pg_attribute.attname = <columnname>;
```

Dealing with big tables:
Q: I have a table which currently has about 500 million rows. For the most part, the situation is going to be that I will import a few hundred million more rows from text files once every few months but otherwise there won't be any insert, update or delete queries. I have created five indexes, some of them multi-column, which make a tremendous difference in performance for the statistical queries which I need to run frequently (seconds versus hours.) When adding data to the table, however, I have found that it is much faster to drop all the indexes, copy the data to the table and then create the indexes again (hours versus days.) So, my question is whether this is really the best way. Should I write a script which drops all the indexes, copies the data and then recreates the indexes or is there a better way to do this?

A: Yes, that's actually recommended practice for such cases.

----

varchar vs varchar (n)

Q: Is there any practical difference between defining a column as a varchar(n) vs. a varchar vs. a text field?

A: No except for your already noted exception that you can limit the size of varchar.

Rewriting Tables On Disk

VACUUM FULL is one of three ways a table can get rewritten.   Besides ALTER TABLE, the other way is with the CLUSTER command. An ALTER TABLE is the only one of the three that *may* rewrite the table - the other two are guaranteed to do so.

To determine if a table was rewritten, use the internal system column **ctid**.

Naturally, you do not want to perform this test using your actual table. In this example, we will create a simple dummy table. As long as the column types are the same as your real table, you can determine if the change will do a table rewrite on your version of PostgreSQL.

The aforementioned ctid column represents the physical location of the table's row on disk. This is one of the rare cases in which this column can be useful. The ctid value consists of two numbers: the first is the "page" that the row resides in, and the second number is the slot in that page where it resides. To make things confusing, the page numbering starts at 0, while the slot starts at 1, which is why the very first row is always at ctid (0,1). However, the only important information for this example is determining if the ctid for the rows has changed or now (which indicates that the physical on-disk data has changed, even if the data inside of it has not!).

To display the ctid column:

```
-- Note: the ctid column is never included as part of '*'
postgres=# SELECT ctid, * FROM babies;
 ctid  | gender | births
-------+--------+--------
 (0,1) | Girl   |      1
 (0,2) | Boy    |      1
(2 rows)
```

Example of an update query to update multiple columns in tbl1 from

```
columns in tbl2:

UPDATE tbl1
SET col3=t2.col3, col4=t2.col4, col5=t2.col5
FROM tbl2 t2 WHERE t2.col1="criteria"
```