CSCI 403
Database Management

9

Miscellaneous DDL

DEFAULT COLUMN VALUES, SEQUENCES, SERIAL

Defaults

Provide a default value for a column
- Only for new rows (e.g., by INSERT statement)
- Only if a value is not provided in the INSERT

```
CREATE TABLE foo (
    x text,
    y integer DEFAULT -1
);
INSERT INTO foo (x) VALUES ('orange');
INSERT INTO foo (x, y) VALUES ('apple', 42);
SELECT * FROM foo;
   x    | y
--------+----
 orange | -1
 apple  | 42
```


Defaults with Functions

Default expression can be a function:

```
CREATE TABLE person (
    first text, last text,
    hire_date date DEFAULT current_date );
```

However, cannot contain column names:

```
CREATE TABLE person (
    first text, last text,
    fullname text DEFAULT first || ' ' || last );
ERROR:  cannot use column references in default expression
```


Sequences

- Database object which generates integer sequences
  - Defaults to 1, 2, 3, …, $2^{63} - 1$ (at least in Postgres)
  - Can initialize with different settings:
    - Starting values
    - Increments
    - Ascending/descending
    - Cyclic/terminating
- Great e.g. for generating unique ID values
  - Can be used with default column value (more soon)

Creating/Using Sequences

```
-- create a sequence with default settings
CREATE SEQUENCE foo_seq;

-- get next value in sequence
-- note the parameter is a string!
SELECT nextval('foo_seq');

-- get most recently generated value
-- only if used nextval in session!
SELECT currval('foo_seq');

-- reset sequence value
SELECT setval('foo_seq', 101);
```

There are some subtleties to working with sequences: see https://www.postgresql.org/docs/9.5/static/functions-sequence.html

Serial Type (PostgreSQL)

- Sequences mostly used to generate ID values
  - Create a sequence associated with a table
  - Set default for ID column to call nextval(…)
- Serial type does all of this for you:

```
CREATE TABLE my_stuff (
    id serial PRIMARY KEY,
    stuff text);

\d my_stuff
                    Table "public.my_stuff"
 Column |  Type   | Nullable |               Default
--------+---------+----------+--------------------------------------
 id     | integer | not null | nextval('my_stuff_id_seq'::regclass)
 stuff  | text    |          |
Indexes:
    "my_stuff_pkey" PRIMARY KEY, btree (id)
```


Serial Type (PostgreSQL)

```
CREATE TABLE my_stuff (
    id serial PRIMARY KEY,
    stuff text);

INSERT INTO my_stuff (stuff)
VALUES ('yellow'), ('blue');

SELECT * FROM my_stuff;
```

```
 id | stuff
----+--------
  1 | yellow
  2 | blue
```


INDEXES

Indexes

We talked about indexes – created for:

- Primary keys
- Unique column subsets

Can also create on non-unique column (subsets):

```
CREATE INDEX foo_x_idx ON foo (x);
```

Lots of options – read the docs!

https://www.postgresql.org/docs/9.5/static/sql-createindex.html

VIEWS

Views

Database objects which act like saved queries:

```
CREATE VIEW mines_cs_course_data
AS
  SELECT mc.crn, mc.course_id,
         mc.section, mc.instructor,
         mcf.office, mcf.email
  FROM mines_courses AS mc,
       mines_cs_faculty AS mcf
  WHERE mc.instructor = mcf.name;
```

A view acts like a table for SELECTs:

```
SELECT * FROM mines_cs_course_data
WHERE mc.course_id LIKE 'CSCI4%';
```

Views 2

- Views do not store data:
  - A view is just a saved, named query
  - Underlying data lives in tables involved in query
  - Queries on views essentially rewrite query
- Why views:
  - Simplify complicated queries for end users
  - Security – give specific users access to specific subsets of data

Views 3

Usually you should treat views as read-only:

- Use SELECT
- Don't use INSERT/UPDATE/DELETE

However, some databases allow modification queries, sometimes with complex mechanisms.

Outside the scope of this course!

ALTER

ALTER TABLE

Lets you modify a table just about any way:

- Rename table
- Add/remove columns
- Change column name/type
- Add/remove/modify constraints, defaults

https://www.postgresql.org/docs/9.5/static/sql-altertable.html

ALTER X

Generally, if you want to modify a DB object, there is an ALTER command for it.

I've pretty much only used ALTER TABLE, though.

DROP

DROP X

How you get rid of DB objects.
As with ALTER, there is a DROP for just about every object type.

E.g.

```
DROP TABLE foo;
```

Note some objects may have dependencies on others, e.g., foreign keys – use CASCADE to get rid of dependent objects:

```
DROP TABLE foo CASCADE;
```

Very useful clause – IF EXISTS – great in scripts:

```
DROP TABLE IF EXISTS foo;
```


"Read The Fine Manual"

RTFM

How To Read PostgreSQL Docs

[stuff] – this element is optional

{ a | b | c } – choose one of a, b, or c

… – you can repeat the previous element

Anything else – required

Example on next page.

(Simplified) Docs Example

```
CREATE TABLE [IF NOT EXISTS] table_name ( [
  {  column_name column_data_type [column_constraint […]]
   | table_constraint }
  [,…]
  ] )
;

where column_constraint is:
[ CONSTRAINT constraint_name ]
{  NOT NULL
 | CHECK (expression)
 | DEFAULT default_expr
 | UNIQUE
 | PRIMARY KEY
 | REFERENCES reftable [ ( refcolumn ) ]
}

and table_constraint is:
Etc.
```


(Simplified) Docs Example

```
CREATE TABLE [IF NOT EXISTS] table_name ( [
  {  column_name column_data_type [column_constraint […]]
   | table_constraint }
  [,…]
  ] )
;
```

[IF NOT EXISTS] is optional, but means something here – sometimes optional elements are truly optional, like [AS] in a SELECT query

{ *column_name* etc. | *table_constraint* } means you can define a column or a table constraint (primary key, foreign key, etc.)

[,…] means you can optionally repeat the last element after putting in a comma. Here, the last element is either a column definition or a table constraint.

Up Next

- Next lecture:
  Subqueries.