

CSCI 403: Databases

4 - Basic DDL

Schema

In the relational model of the database, we used schema as a term to describe the names and attributes of relations and the constraints on the relations in a database. In the SQL standard, schema is defined more broadly as a container for database objects, including tables, constraints, views, indexes, etc. A SQL database can contain many schemas.

- Like namespaces, a schema allows us to separate logical units in a database and avoid name collisions.
- Schemas also makes for easier application of security policies by letting us grant permissions on whole schemas at a time rather than table by table.
- Schemas are generally something created and managed by the database administrator/owner, not by the database user or programmer. (Acknowledging that of course, programmers may also be DBAs or DB owners!)
- You do not (AFAIK) have rights to create schemas in the course database. You have your own schema, named by your Mines id. Anything you create by default will be created in your personal schema. You also have read-only access to a schema named "public".

Catalog

In SQL, the *catalog* can be viewed as either a higher-level container containing schemas, or as the set of system tables within the database describing all of the other objects in the database. All relational DBMSes have some form of system

tables which can be read to determine the structure of objects in the database. To see the system tables in psql, enter the command "\dS" at the prompt.

Table Creation

SQL uses a very "English-like" syntax. To create a table in SQL, you issue a CREATE TABLE command. A somewhat abbreviated syntax of the command looks like:

```
CREATE TABLE [schemaname.]tablename ( {  
columnname datatype [NOT NULL] [UNIQUE]  
[PRIMARY KEY] | table_constraint } [...])
```

Which probably looks like gibberish... you will need to learn how to read SQL documentation, most of which looks like this. Briefly, anything in [] is optional, while the construct { A | B } means you can have A or B. If you see [...], that means you can have more copies of the last optional thing (with a comma separator). Anything else is required.

Here's an example of creating a throwaway table, showing some of the different data types you might use:

```
CREATE TABLE yourid.stuff (  
id serial PRIMARY KEY,  
name text NOT NULL,  
age integer,  
gender char(1),  
salary numeric(9,2),  
favorite_constant double precision,  
date_hired date);
```

This will create a table named "stuff" in the schema "yourid" with several columns, each of a different type. (Note that text and serial types are specific to PostgreSQL, although similar types exist in other DBMSes.) This construction sets the column "id" as a primary key for the table (just the one column in the key), and the name

field is constrained to not contain NULL values. We can also create foreign key references or other constraints within the table creation statement. Table constraints can also be created in a separate declaration in the comma-separated list within the (); here's the same table, but with a two-column primary key (on name and age):

```
CREATE TABLE yourid.stuff (  
  id serial,  
  name text NOT NULL,  
  age integer,  
  gender char(1),  
  salary numeric(9,2),  
  favorite_constant double precision,  
  date_hired date,  
  PRIMARY KEY (name, age));
```

Foreign key constraints can also be created in the CREATE TABLE command:

```
CREATE TABLE yourid.otherstuff (  
  other_id serial PRIMARY KEY,  
  stuff_id integer REFERENCES  
  yourid.stuff (id));
```

is equivalent to

```
CREATE TABLE yourid.otherstuff (  
  other_id serial PRIMARY KEY,  
  stuff_id integer,  
  FOREIGN KEY (stuff_id) REFERENCES  
  yourid.stuff (id));
```

Types

SQL defines a number of types for attributes in your relations, and most DBMSes define additional types. Some types that you will find useful are listed below.

Integers

- INTEGER - 32-bit integers
- SMALLINT - 16-bit integers
- BIGINT - 64-bit integers

Fixed-precision numeric (exact)

- NUMERIC(w,p) - Defines numbers with a maximum of w digits, and a precision of p
- DECIMAL(w,p) - same as NUMERIC(w,p)

Floating point (inexact)

- REAL - 32-bit floating point
- DOUBLE PRECISION - 64-bit floating point

Strings

- CHAR(n) - strings of length exactly n, padded with spaces if necessary
- VARCHAR(n) - variable length strings of max length n
- TEXT - variable length strings, no limit (PostgreSQL type)

Date/Time

- DATE - holds dates. You can enter DATE values as strings in the format 'YYYY-MM-DD', other formats maybe possible depending on DBMS.
- TIME - holds times. Format 'HH:MM:SS', can also add decimal points after for sub-second times. Optionally, timezone can also be included.
- TIMESTAMP - date and time.

Other types

- BOOLEAN - holds true/false values - various formats are compatible
- SERIAL - an auto-incrementing integer type (PostgreSQL type)
- MONEY
- ... many more

Type Conversion

There are several ways to convert one type to another in PostgreSQL. The standard SQL way is to use the CAST function:

```
CAST (expression AS type)
```

e.g.,

```
SELECT CAST('1/2/2016' AS DATE) AS  
foo;
```

Generally speaking, if the database can work out how to convert a type, it will do so without any special effort on your part. In particular, the database can usually figure out how to convert string representations of numbers, dates, times, etc. into the corresponding actual types. NULLs get left as NULLs.

However, you have to be careful when trying to convert, for instance, all entries in a column in

a table - if not all the entries are recognizable as the type you are converting to, the query will fail. For instance, if a string column mostly contains integers, but a few rows have 'unknown' or 'N/A' or some other string, the database will refuse to try converting them to a numeric type.

Another (more compact) way to do casts in PostgreSQL (not portable to other systems) is the `::` operator, which is used as follows:

```
expression::type  
e.g., SELECT '1/2/2016'::DATE AS foo;
```

There are other, more involved ways to do type conversions using functions and/or the CASE statement, but generally you want to avoid the complication of these approaches.

Default values

Another option when creating a table is to set a default value for a column. The default value will be used whenever a new tuple (row) is INSERTed (see next lecture) into the table, but no value is provided for the column. For instance:

```
CREATE TABLE yourid.stuff (  
  id serial PRIMARY KEY,  
  name text NOT NULL,  
  salary numeric(9,2) DEFAULT 0.00,  
  date_hired date DEFAULT CURRENT_DATE);  
creates a table where the salary field defaults to zero, and where the date_hired field defaults to the current date (the date on which the row was created), using the SQL standard function CURRENT_DATE.
```

The SERIAL type mentioned above is actually implemented as an integer type with the default value being obtained via the function `nextval()` applied to a *sequence* object associated with the column. A sequence is just a counter object which can be queried for its current value (`curval()`) or its next value (which also increments the counter). The value of the counter can also be set (using `setval()`).

CREATE... AS

An easy way to create a table from a SELECT query; do

```
CREATE TABLE schemaname.tablename AS  
SELECT ... ;
```

This will create a table with attributes and types determined by the SELECT query result.

Column renaming together with functions, joins, etc., makes this a powerful way to create a new table. Note, however, that this does not create any keys or constraints - these will have to be added on using ALTER TABLE.

Notes on Workflow

Tables/schemas are generally created only occasionally and modified (relatively) seldom thereafter. [You may have a different experience in an agile environment; one reason NoSQL is popular right now is the "schema-less" nature of NoSQL databases.]

When making tables, it is easy to make small mistakes or wish you had done something differently. Changes after the fact are often harder than simply dropping everything and starting over. Thus: make scripts! Your scripts should (optionally) drop everything, then create all of your tables and constraints, load all of your data, etc. Re-run until you are happy with the result. (Having these scripts will help a lot downstream when your application goes into development for version 2...)

ALTER TABLE

When you *do* want to modify a table in place - which of course happens - then you want the ALTER TABLE command. You can do almost anything with this command (see the PostgreSQL documentation), including adding and removing columns, renaming columns, changing data types of columns, adding and removing constraints, setting column default values, etc.