DDL & Schema Design

Springboard

Download Demo Code

Goals

- Learn SQL Commands to Create, Update, and Remove Databases & Tables
- Understand the Basics of Database Schema Design
- Learn How to Properly Model Relational Data

DDL Basics

Creating and Dropping Databases

```
CREATE DATABASE yet_another_db;

DROP DATABASE yet_another_db;
```

Same as shell commands **createdb** and **dropdb**

Creating Tables

```
jane=# CREATE DATABASE library;
CREATE DATABASE

jane=# \c library
You are now connected to database "library" as user "jane".
library=#
```

```
CREATE TABLE books (
  id SERIAL PRIMARY KEY,
  title TEXT,
  author TEXT,
  genre TEXT,
  page_count INTEGER,
  publisher TEXT,
  publication_date DATE
);
```

Inspecting Tables in PostgreSQL

Listing the tables in the database

```
library=# \dt
```

Listing the column names and types in a specific table

```
library=# \d+ books
```

Dropping Tables

```
DROP TABLE users;
```

Column Data Types

Integer
Integer numbers

Float
Floating point numbers (you can specify the precision)

Text
Text Strings

Varchar
Text Strings, but limited to a certain size

Boolean
True or False

Date
Date (without time)

Timestamp
Date and time

Serial
Auto-incrementing numbers (used for primary keys)

Note: Other Types

There are lots of other types, including specialized, less-common types for fixed-precision math (**NUMERIC** or **DECIMAL**), handling geospatial information, currency, and more!

NULL

NULL is a special value in SQL for "unknown".

It's **not** the same thing as 0 or an empty string!

NULL values are ok when you really might have missing/unknown data

But generally, they're a pain, so it can be a good idea to make fields not nullable

Primary Keys

Every table should have a "primary key", a unique way to identify rows

Primary keys *must be*:

- Unique
- Not Null

Primary keys *should be*:

- Unchanging (it's a pain when primary keys change)

Constraints

Constraints are a basic form of validation. The database can prevent basic types of unintended behavior.

- Primary Key (every table must have a unique identifier)
- Unique (prevent duplicates in the column)
- Not Null (prevent null in the column)
- Check (do a logical condition before inserting / updating)
- Foreign Key (column values must reference values in another table)

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    phone_number TEXT UNIQUE,
    password TEXT NOT NULL,
    account_balance FLOAT CHECK (account_balance > 0)
);
```

Column Manipulation

Adding / Removing / Renaming columns

```
ALTER TABLE books ADD COLUMN in_paperback BOOLEAN;

ALTER TABLE books DROP COLUMN in_paperback;

ALTER TABLE books RENAME COLUMN page_count TO num_pages;
```

Structuring Relational Data

Modeling Our Movies Database

From our joins exercise involving movies, studios, actors, and roles, we can see that:

- one studio has many movies
- one actor has many roles
- one movie has many actors

Before we write out the DDL, we'll visualize this a few ways.

As A Spreadsheet

Check out this color-coded spreadsheet.

Crow's Foot Notation

Preferably, we will draw diagrams with Crow's Foot Notation, which is a standard way to represent schemas.

DDL for Movies

Let's look at the DDL from the earlier example

```
CREATE TABLE studios (
  id SERIAL PRIMARY KEY,
  name TEXT NOT NULL,
  founded_in DATE
);

CREATE TABLE movies (
  id SERIAL PRIMARY KEY,
  title TEXT NOT NULL,
  release_year INTEGER,
  runtime INTEGER,
  rating TEXT,
  studio_id INTEGER REFERENCES studios
);
```

Controlling Delete Behavior with DDL

```
CREATE TABLE movies (
  id SERIAL PRIMARY KEY,
  title TEXT NOT NULL,
  release_year INTEGER,
  runtime INTEGER,
  rating TEXT,
  studio_id INTEGER REFERENCES studios ON DELETE SET NULL
);
```

```
CREATE TABLE movies (
  id SERIAL PRIMARY KEY,
  title TEXT NOT NULL,
  release_year INTEGER,
  runtime INTEGER,
  rating TEXT,
  studio_id INTEGER REFERENCES studios ON DELETE CASCADE
);
```

Many-to-Many DDL

```
CREATE TABLE movies (
  id SERIAL PRIMARY KEY,
  title TEXT NOT NULL,
  release_year INTEGER NOT NULL,
  runtime INTEGER NOT NULL,
  rating TEXT NOT NULL
);

CREATE TABLE actors (
  id SERIAL PRIMARY KEY,
  first_name TEXT NOT NULL,
  last_name TEXT NOT NULL,
  birth_date DATE NOT NULL
);

CREATE TABLE roles (
  id SERIAL PRIMARY KEY,
  movie_id INTEGER REFERENCES movies ON DELETE CASCADE,
  actor_id INTEGER REFERENCES actors ON DELETE CASCADE
);
```

Best Practices

Normalization

Normalization is a database design technique which organizes tables in a manner that reduces redundancy and dependency of data.

It divides larger tables to smaller tables and links them using relationships.

Normalization Bad Example

Consider the following products table. There are strings with multiple values in the **color** column, making it difficult to query.

products

id	color	price
1	red, green	05.00
2	yellow	10.00

Normalized Example

products

id	price
1	05.00
2	10.00

colors

id	color
1	red
2	green
3	yellow

products_colors

id	color_id	product_id
1	1	1
2	2	1
3	3	2

Another Bad Example

Consider the following purchases table. It's bad because `store_location` is fully dependent on `store_id`.

purchases

customer_id	store_id	store_location
1	1	New York
1	3	Boston
2	2	San Francisco
3	1	New York

Normalized Example

stores

store_id	store_location
1	New York
3	Boston
2	San Francisco

purchases

customer_id	store_id
1	1
1	3
2	2
3	1

Indexing

A *database index* is a special data structure that efficiently stores column values to speed up row retrieval via ***SELECT*** and ***WHERE*** (i.e. "read") queries.

For instance, if you place an index on a **username** column in a **users** table, any query using username will execute faster since fewer rows have to be scanned due to the efficient structure.

Index Efficiency

In general, database software (including PostgreSQL) use tree-like data structures to store the data, which can retrieve values in logarithmic time $O(\log N)$ instead of linear $O(N)$ time.

Translation: If we have 1,000,000 rows and are looking for a single column value, instead of examining every row, we can examine approximately $\log_2(1000000) = 20$ rows to get our answer, which is an incredible improvement!

Why Not Index Everything?

There is a tradeoff with indexing! For every indexed column, a copy of that column's data has to be stored as a tree, which can take up a lot of space

Also, every INSERT and UPDATE query becomes more expensive, since data in both the regular table AND the index have to be dealt with.

How to Create an Index in PostgreSQL

Indexing is part of DDL, but indexes can be created or dropped at any time. The more records in the database at the time of creation, the slower the indexing process will be.

```
CREATE INDEX index_name ON table_name (column_name);
```

You can also create a multi-column index, which is useful if you are constantly querying by two fields at once (e.g. first_name and last_name):

```
CREATE INDEX index_name ON table_name (column1_name, column2_name);
```

How to Drop an Index in PostgreSQL

Drop an index:

```
DROP INDEX full_name;
```

Note: When to Index

Indexes are used in every PostgreSQL table by default on the primary key column.

In general, if you are building an application that is more read-heavy than write-heavy, indexes are your friend and can be safely placed on columns that are used frequently in queries to speed up performance.

However, there are other index types besides the default that may be more efficient for your data, so definitely read up on some PostgreSQL performance optimizations here and here.