Relational Databases and SQL, Part II
Statistics 650/750
Week 8 Tuesday

Christopher Genovese and Alex Reinhart

16 Oct 2018

Announcements

- Notes and ClassFiles in documents repository (`Lectures` and `ClassFiles/week8`, respectively)
- See Instructions page for details on connecting to SQL server in `documents/Info/SQL/`. There is a README and examples.
- Challenge revisions should submitted by today; Part 2 due next week.

Brief Review

Managing Tables

Command summary:

- `create table NAME (attribute1 type1, attribute2 type2, ...);`
- `alter table NAME ALTERATION;`

  Common ATERATIONS:

  - Rename column: `rename OLD_NAME to NEW_NAME`
  - Add column: `add NAME TYPE [CONSTRAINTS]`
  - Drop column: `drop NAME`
  - Alter column: `alter NAME set CONFIG`
- `drop table NAME;`

Examples: Creating Tables

```sql
create table star (
       id SERIAL PRIMARY KEY,
       ra  numeric,
       dec numeric,
       epoch date,
       magnitude numeric
);
```

```
create table products (
        product_id SERIAL PRIMARY KEY,
        name text,
        price numeric CHECK (price > 0),
        sale_price numeric CHECK (sale_price > 0),
        CHECK (price > sale_price)
);
```

```
create table products (
        product_id SERIAL PRIMARY KEY,
        label text UNIQUE NOT NULL CHECK (char_length(label) > 0),
        price numeric CHECK (price >= 0),
        discount numeric DEFAULT 0.0 CHECK (discount >= 0),
        CHECK (price > discount)
);
```

Examples: Altering Tables

A few examples using the most recent definition of `products` above:

- Change a column name

```
alter table products rename product_id to id;
```

- Let's add a `brand_name` column.

```
alter table products add brand_name text DEFAULT 'generic' NOT NULL;
```

- Let's drop the `discount` column

```
alter table products drop discount;
```

- Let's set a default value for `brand_name`.

```
alter table products alter brand_name SET DEFAULT 'generic';
```

Example: Dropping Tables

```
drop table products;
```

CRUD Review/Summary

- `INSERT` – populate *new* rows of a table with data

  ```
  INSERT INTO <tablename> (<column1>, ..., <columnk>)
         VALUES (<value1>, ..., <valuek>)
         RETURNING <expression|*>;
  ```

  More than one tuple can be given as values, each for a successive row. The `RETURNING` clause is optional, values can be set to `DEFAULT` to specify the default value.

- `SELECT` – generate values from data in a table in table format

  SELECT is a powerful command for generating data. It's most common use is to read (and possibly transform) data in one or more tables, but it can be used in other ways as well.

  It has many forms, but a common way to read data from a table looks like

  ```
  SELECT expressions FROM source WHERE conditions;
  ```

- `UPDATE` – change the values in selected existing rows

  ```
  UPDATE table
      SET col1 = expression1,
          col2 = expression2,
          ...
      WHERE condition;
  ```

  This can have an optional `RETURNING` clause like `INSERT`.

- `DELETE` – drop rows from the table

  ```
  DELETE FROM table WHERE condition;
  ```

  The `WHERE` clause is optional, but without it, you will delete all the table's rows.

Activity

Here, we will do some brief practice with CRUD operations by generating a table of random data and playing with it.

1. Create a table `rdata` with five columns: one `integer` column `id`, two `text` columns `a` and `b`, one `date` `moment`, and one `numeric` column `x`.

2. Use a `SELECT` command with the `generate_series` function to display the sequence from 1 to 100.

3. Use a `SELECT` command with the `random()` function converted to `text` (via `random()::text`) and the `md5` function to create a random text string.

4. Use a `SELECT` command to choose a random element from a fixed array of strings. A fixed text array can be obtained with `('{X,Y,Z}'::text[])` and then indexed using the `ceil` (ceiling) and `random` functions to make a selection. (FYI, `('{X,Y,Z}'::text[])[1]` would give 'X'.) (SQL is 1-indexed.)

5. `SELECT` a random date in 2017. You can do this by adding an integer to `date '2017-01-01'`. For instance, try

   ```
   select date '2017-01-01' + 7 as random_date;
   ```

   For a non-integer type, append `::integer` to convert it to an integer.

6. Use `INSERT` to populate the `rdata` table with 101 rows, where the `id` goes from 1 to 100, `a` is random text, `b` is random choice from a set of strings (at least three in size), `moment` contains random days in 2017, and `x` contains random real numbers in some range.

7. Use `SELECT` to display rows of the table for which `b` is equal to a particular choice.

8. Use `SELECT` with either the `~*` or `ilike` operators to display rows for which `a` matches a specific pattern, e.g.,

   ```
   select * from rdata where a ~* '[0-9][0-9][a-c]a';
   ```

9. Use `SELECT` with the `overlaps` operator on dates to find all rows with `moment` in the month of November.

10. Use `UPDATE` to set the value of `b` to a fixed choice for all rows that are divisible by 3 and 5.

11. Use `DELETE` to remove all rows for which `id` is even and greater than 2. (Hint: `%` is the mod operator.)

12. Use a few more `DELETE`'s (four more should do it) to remove all rows where `id` is not prime.

Joins and Foreign Keys

As we will see shortly, principles of good database design tell us that tables represent distinct entities with a single authoritative copy of relevant data. This is the DRY principle in action, in this case eliminating *data redundancy*.

An example of this in the `events` table are the `persona` and `element` columns, which point to information about students and components of the learning environment. We do **not** repeat the student's information each time we refer to that student. Instead, we use a **link** to the student that points into a separate `Personae` table.

But if our databases are to stay DRY in this way, we need two things:

1. A way to define links between tables (and thus define *relationships* between the corresponding entities).
2. An efficient way to combine information across these links.

The former is supplied by foreign keys and the latter by the operations known as joins. We will tackle both in turn.

Foreign Keys

A **foreign key** is a field (or collection of fields) in one table that *uniquely* specifies a row in another table. We specify **foreign keys** in Postgresql using the `REFERENCES` keyword when we define a column or table. A foreign key that references another table must be the value of a unique key in that table, though it is most common to reference a *primary key*.

Example:

```sql
create table countries (
       country_code char(2) PRIMARY KEY,
       country_name text UNIQUE
);
insert into countries
  values ('us', 'United States'), ('mx', 'Mexico'), ('au', 'Australia'),
         ('gb', 'Great Britain'), ('de', 'Germany'), ('ol', 'OompaLoompaland');
select * from countries;
delete from countries where country_code = 'ol';

create table cities (
       name text NOT NULL,
       postal_code varchar(9) CHECK (postal_code <> ''),
       country_code char(2) REFERENCES countries,
       PRIMARY KEY (country_code, postal_code)
);
```

Foreign keys can also be added (and altered) as *table constraints* that look like `FOREIGN KEY (<key>) references <table>`.

Now try this

```sql
insert into cities values ('Toronto', 'M4C185', 'ca'), ('Portland', '87200', 'us');
```

Notice that the insertion did not work – and the entire transaction was rolled back – because the implicit foreign key constraint was violated. There was no row with country code 'ca'.

So let's fix it. Try it!

```
insert into countries values ('ca', 'Canada');
insert into cities values ('Toronto', 'M4C185', 'ca'), ('Portland', '87200', 'us');
update cities set postal_code = '97205' where name = 'Portland';
```

Joins

Suppose we want to display features of an event with the name and course of the student who generated it. If we've kept to DRY design and used a foreign key for the `persona` column, this seems inconvenient.

That is the purpose of a **join**. For instance, we can write:

```
select personae.lastname, personae.firstname, score, moment
       from events
       join personae on persona = personae.id
       where moment > timestamp '2015-03-26 08:00:00'
       order by moment;
```

Joins incorporate additional tables into a select. This is done by appending to the `from` clause:

`from <table> join <table> on <condition> ...`

where the `on` condition specifies which rows of the different tables are included. And within the select, we can disambiguate columns by referring them to by `<table>.<column>`. Look at the example above with this in mind.

We will start by seeing what joins mean in a simple case.

```
create table A (id SERIAL PRIMARY KEY, name text);
insert into A (name)
       values ('Pirate'),
              ('Monkey'),
              ('Ninja'),
              ('Flying Spaghetti Monster');

create table B (id SERIAL PRIMARY KEY, name text);
insert into B (name)
       values ('Rutabaga'),
              ('Pirate'),
              ('Darth Vader'),
              ('Ninja');
select * from A;
select * from B;
```

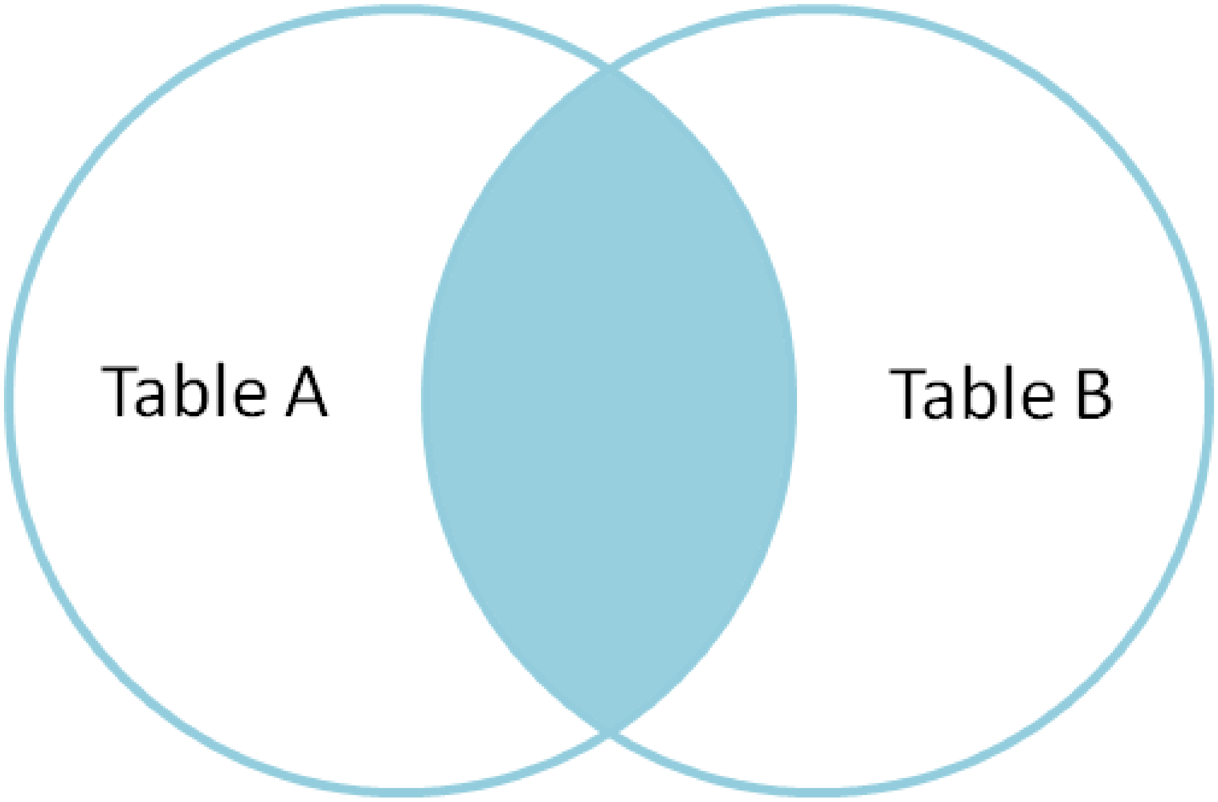
Let's look at several kinds of joins. (There are others, but this will get across the most common types.)

Inner Join

An **inner join** produces the rows for which attributes in **both** tables match. (If you just say `JOIN` in SQL, you get an inner join; the word `INNER` is optional.)

```sql
select * from A INNER JOIN B on A.name = B.name;
```

We think of the selection done by the `on` condition as a *set operation* on the rows of the two tables. Specifically, an inner join is akin to an intersection:

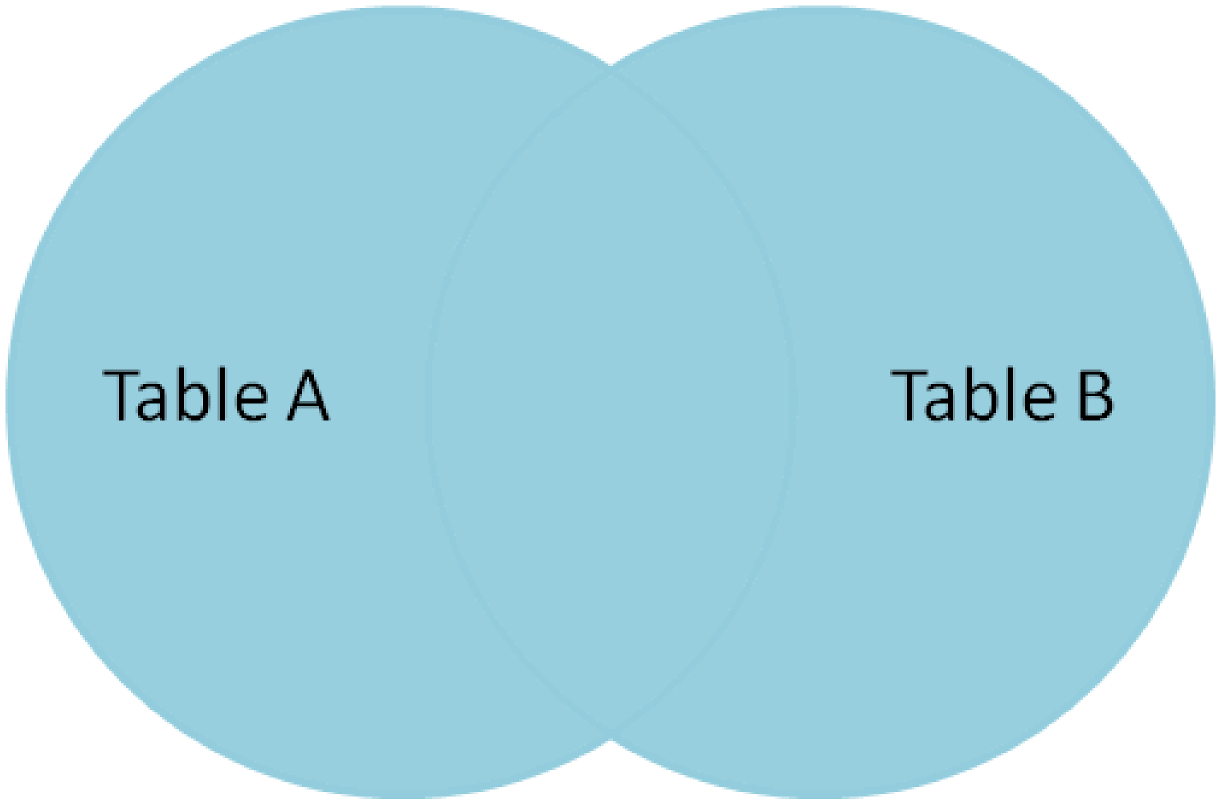


Full Outer Join

A full outer join produces the full set of rows in **all** tables, matching where possible but `null` otherwise.

```sql
select * from A FULL OUTER JOIN B on A.name = B.name;
```

As a set operation, a full outer join is a *union*

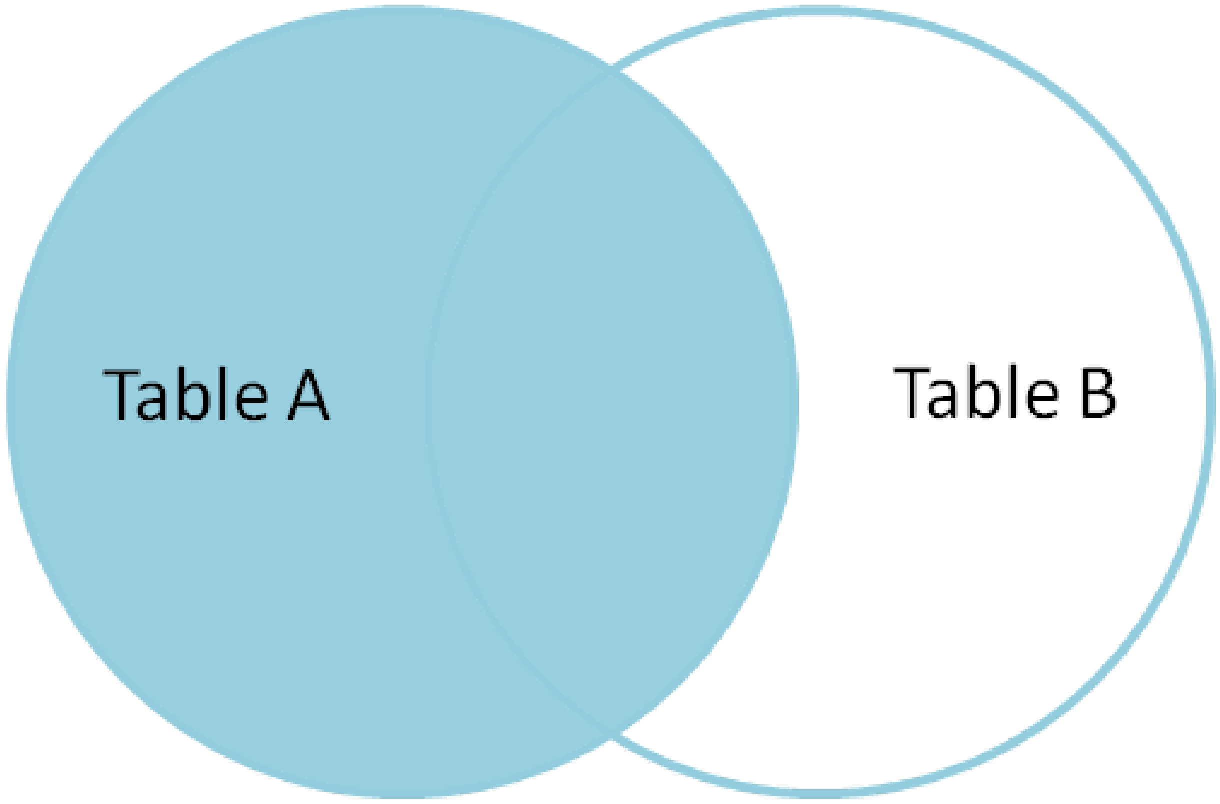


Left Outer Join

A left outer join produces all the rows from A, the table on the "left" side of the `join` operator, along with matching rows from B if available, or `null` otherwise. (`LEFT JOIN` is a shorthand for `LEFT OUTER JOIN` in postgresql.)

```sql
select * from A LEFT OUTER JOIN B on A.name = B.name;
```

A left outer join is a hybrid set operation that looks like:

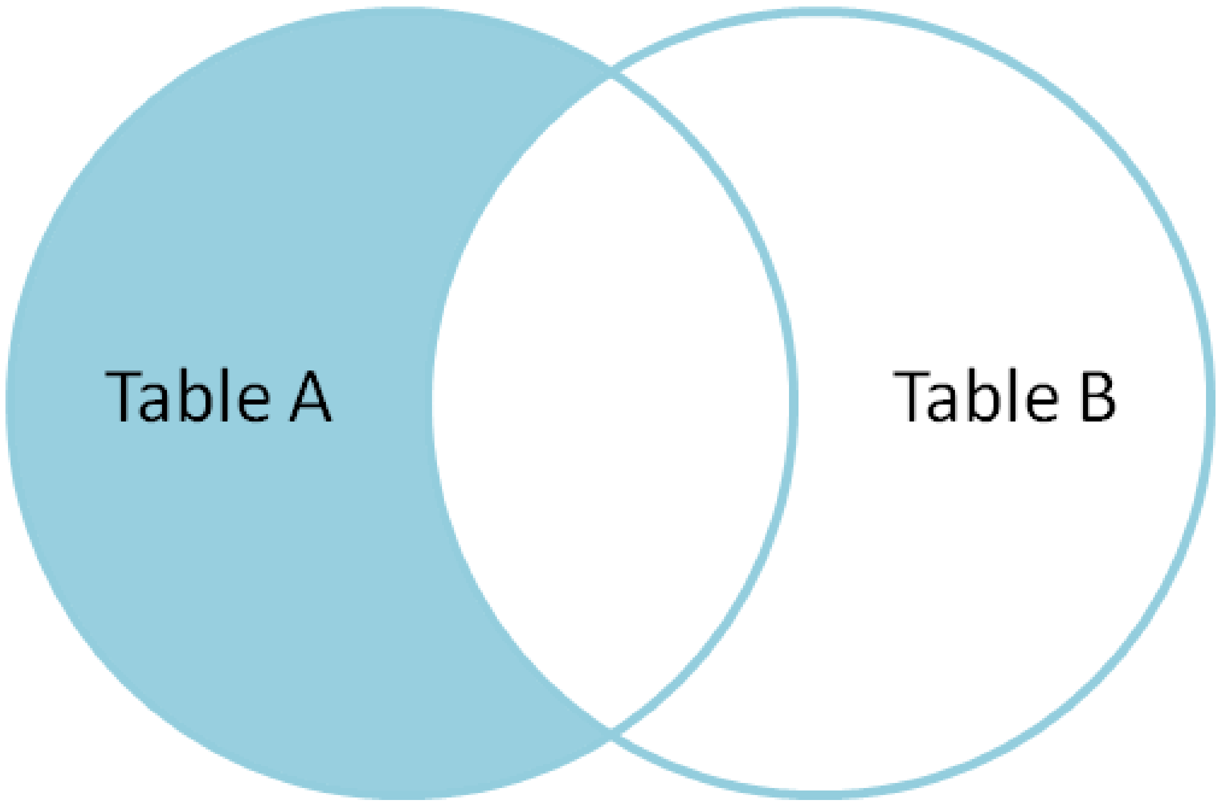


Set Difference

Exercise: Give a selection that gives all the rows of A that are **not** in B.

```sql
select * from A LEFT OUTER JOIN B on A.name = B.name where B.id IS null;
```

This corresponds to a *set difference* operation A - B:

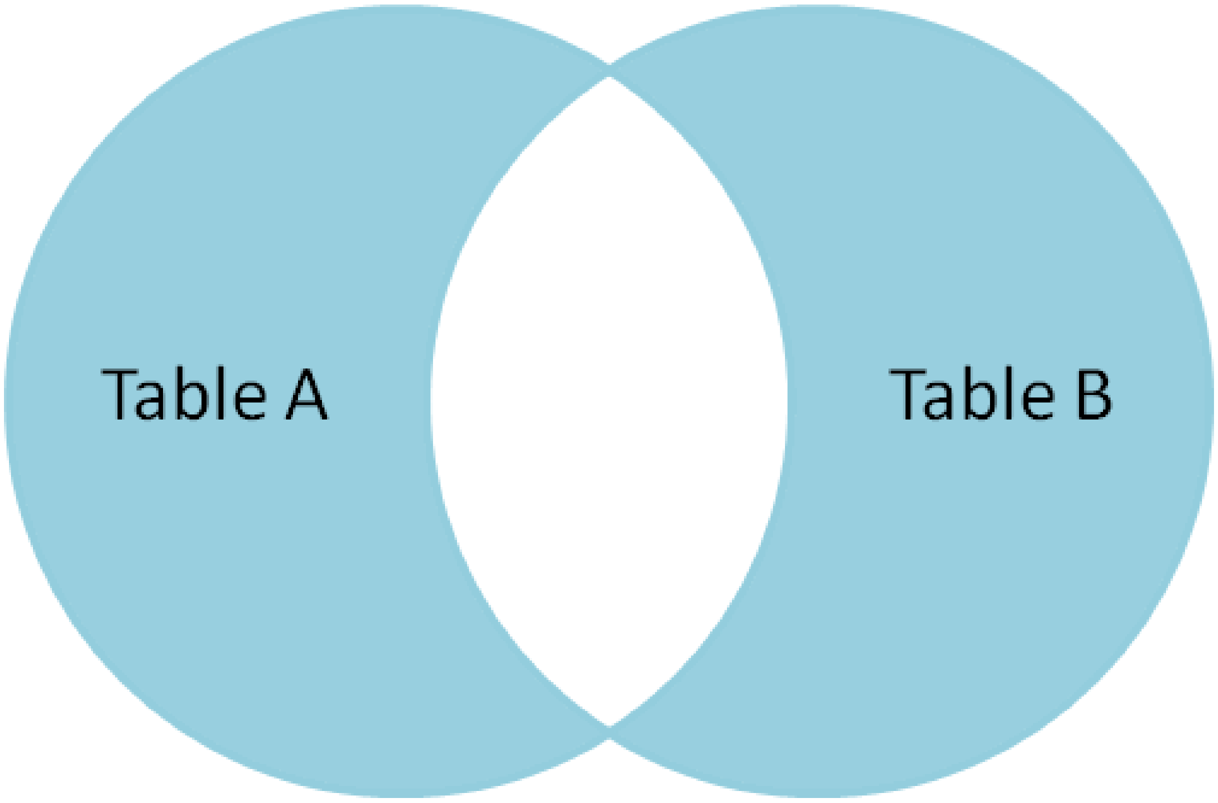


Symmetric Difference

Exercise: Select the rows of A not in B *and* the rows of B not in A.

```
select * from A FULL OUTER JOIN B on A.name = B.name
    where B.id IS null OR A.id IS null;
```

This is the set operation known as a symmetric difference, $A \triangle B = (A - B) \cup (B - A)$:



A slightly more meaningful example

Exercise: Using the `cities` and `countries` tables we created earlier, do the following:

1. List city name, postal code, and country name.

```
select name, postal_code, country_name
    from cities inner join countries
    on cities.country_code = countries.country_code;
```

1. List city name, country, and address as a valid string.

```
select cities.name as city, country_name as country,
       concat(name, ', ', country_name, ' ', postal_code) as address
    from cities inner join countries
    on cities.country_code = countries.country_code;
```

Notice how we can give new names in the produced table (using `AS`) and we can include new columns derived from the old ones.

More:

```
create table venues (
       id SERIAL PRIMARY KEY,
       name varchar(255),
       street_address text,
```

```sql
       type char(7) CHECK (type in ('public', 'private')) DEFAULT 'public',
       postal_code varchar(9),
       country_code char(2),
       FOREIGN KEY (country_code, postal_code)
         REFERENCES cities (country_code, postal_code) MATCH FULL
);
insert into venues (name, postal_code, country_code)
  values ('Crystal Ballroom', '97205', 'us'),
         ('Voodoo Donuts', '97205', 'us'),
         ('CN Tower', 'M4C185', 'ca');
update venues set type = 'private' where name = 'CN Tower';
select * from venues;
```

Now create a social_events table with an automatic id field, a title field that is text and fields starts and ends of type timestamp, and a foreign key for the venue id. Populate it with a few social events. (Timestamps look like '2012-02-15 17:30:00'.)

```sql
create table social_events (
       id SERIAL PRIMARY KEY,
       title text,
       starts timestamp DEFAULT timestamp 'now' + interval '1 month',
       ends timestamp DEFAULT timestamp 'now' + interval '1 month' + interval '3 hours',
       venue_id integer REFERENCES venues (id)
);
insert into social_events (title, venue_id) values ('LARP Club', 3);
insert into social_events (title, starts, ends)
  values ('Fight Club', timestamp 'now' + interval '12 hours', timestamp 'now' + interval '16 hours
insert into social_events (title, venue_id)
  values ('Arbor Day Party', 1), ('Doughnut Dash', 2);
select * from social_events;
```

Exercise: List a) all social events with a venue with the venu names, and b) all social events with venue names even if missing.

```sql
select e.title as event, v.name as venue FROM social_events e JOIN venues v
  on e.venue_id = v.id;
select e.title as event, v.name as venue FROM social_events e LEFT JOIN venues v
  on e.venue_id = v.id;
```

(Recall that JOIN by itself is a shortcut for INNER JOIN, and LEFT JOIN is a shortcut for LEFT OUTER JOIN.)

When we know we will search on certain fields regularly, it can be helpful to create an **index**, which speeds up those particular searches.

```sql
create index social_events_title  on social_events using hash(title);
create index social_events_starts on social_events using btree(starts);

select * from social_events where title = 'Fight Club';
select * from social_events where starts >= '2015-11-28';
```

Exercise

Using the `ALTER TABLE` command, add a text 'organizer' column to the `social_events` table. Add a State/Province column to `cities` table.

Then update `venues` with street addresses, and use a join to create full address labels for mailing the organizer of each event, e.g.

Tyler Durden Organizer: Fight Club 100 Warehouse Road Portland, Oregon 97205 United States

Exercise

We will use the `personae`, `elements`, and `courses` tables defined in `personae-elements.sql` from the from the documents repository. Alter the `events` table so that the `persona` and `element` columns are foreign keys into these new tables.

```sql
alter table events ADD FOREIGN KEY (persona) REFERENCES personae;
alter table events ADD FOREIGN KEY (element) REFERENCES elements;
```

Then use a join to display student names, course numbers (in the form '<department>-<catalog$_{\text{number}}$>'), scores, number of hints, and the date (in format like 'Thu 26 Mar 2015' if possible) for events after 26 March 2015 at 8am.

```sql
select p.lastname, p.firstname,
         c.department || '-' || c.catalog_number as course,
         score,
         hints,
         to_char(moment, 'Dy DD Mon YYYY')
       from events
       join personae as p on persona = p.id
       join courses as c on p.course = c.id
       where moment > timestamp '2015-03-26 08:00:00';
```

Using RDBs from a Programming Language

It's nice to be able to type queries into `psql` and see results, but most often you'd like to do more than that. You're not just making a database to run handwritten queries – you're using it to store data for a big project, and that data then needs to be used to fit models, make plots, prepare reports, and all sorts of other useful things. Or perhaps your code is *generating* data which needs to be stored in a database for later use.

Regardless, you'd like to run queries inside R, Python, or your preferred programming language, and get the results back in a form that can easily be manipulated and used.

Fortunately, PostgreSQL – and most other SQL database systems – use the *client-server* model of database access. The database is a *server*, accessible to any program on the local machine (like the `psql` client) and even to programs on other machines, if the firewall allows it.

This is why you need a password to start `psql`. `psql` connects to the running Postgres server, and to do so it needs a username (the user you logged into SSH as) and a password.

But this also means you can run scripts on your own computer which connect to Postgres with that same username and password.

SQL in R

The RPostgreSQL package provides the interface you need to connect to Postgres from within R. There are similar packages for other database systems, all using a similar interface called DBI, so you can switch to MySQL or MS SQL without changing much code.

To start using Postgres from within R, you need to create a *connection* object, which represents your connection to the server.

```r
library(RPostgreSQL)

con <- dbConnect(PostgreSQL(), user="yourusername", password="yourpassword",
                 dbname="yourusername", host="pg.stat.cmu.edu")
```

con now represents the connection to Postgres. Queries can be sent over this connection. You can connect to multiple different databases and send them different queries.

To send a query, use dbSendQuery:

```r
result <- dbSendQuery(con, "SELECT persona, score FROM events WHERE ...")
```

result is an object representing the result, but *does not* load the actual results all at once. If the query result is very big, you may want to only look at chunks of it at a time; otherwise, you can load the whole thing into a data frame. dbFetch loads the requested number of rows from the result, or defaults to loading the entire result if you'd prefer, all in a data frame.

```r
data <- dbFetch(result) # load all data

data <- dbFetch(result, n=10) # load only ten rows

dbClearResult(result)
```

As a shortcut, dbGetQuery runs a query, fetches all of its results, and clears the result, all in one step.

SQL in Python

Psycopg is a popular PostgreSQL package for Python. It has a different interface; since Python doesn't have native data frames, you can instead iterate over the result rows, where each row is a tuple of the columns. To connect:

```python
import psycopg2

conn = psycopg2.connect(host="pg.stat.cmu.edu", database="yourusername",
                        user="yourusername", password="yourpassword")

cur = conn.cursor()

cur.execute("INSERT INTO foo (bar, baz, spam) "
            "VALUES (17, 'walrus', 'penguin')")
```

If we do a SELECT, we can get the results with a for loop or the fetchone and fetchmany methods:

```
cur.execute("SELECT * FROM events")

# iterating:
for row in cur:
    print(row)

# instead, one at a time:
row = cur.fetchone()
```

The execute method is used regardless of the type of query.

SQL in Other languages

Most modern programming languages have libraries for interfacing with an SQL server. The behavior and organization is usually very similar to those in R and Python. Some (such as Ruby and the modern lisps) offer more syntactic integration – effectively, language constructs that capture SQL structure – that can be a pleasure to use.

A Brief Interlude on Practicing Safe SQL

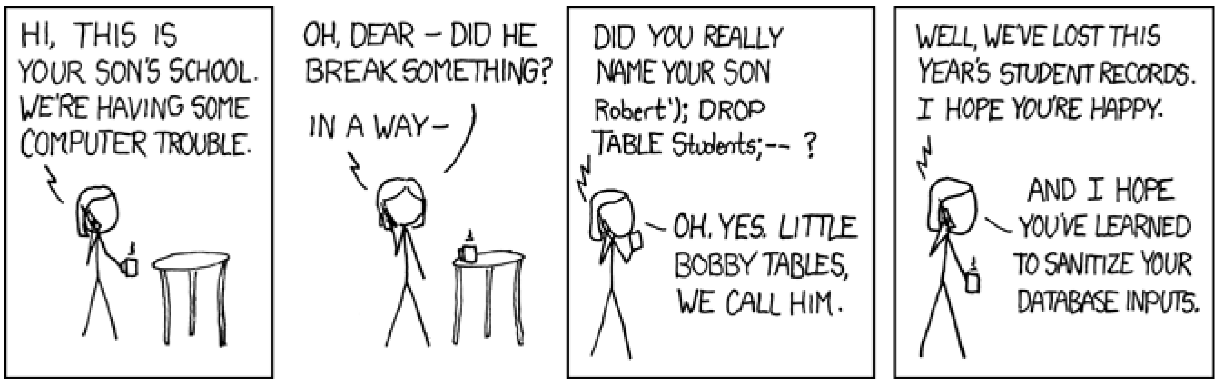
Suppose you've loaded some data from an external source – a CSV file, input from a user, from a website, another database, wherever. You need to use some of this data to do a SQL query.

```
result <- dbSendQuery(paste0("SELECT * FROM users WHERE username = '", username, "' ",
                             "AND password = '", password, "'"))
```

Now suppose username is the string "’; DROP TABLE users;–". What does the query look like before we send it to Postgres?

```
SELECT * FROM users
WHERE username = ''; DROP TABLE users; -- AND password = 'theirpassword'
```

We have *injected* a new SQL statement, which drops the table. Because -- represents a comment in SQL, the commands following are not executed.


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Less maliciously, the username might contain a single quote, confusing Postgres about where the string ends and causing syntax errors. Or any number of other weird characters which mess up the query. Clever attackers can use SQL injection to do all kinds of things – imagine if the `password` variable were `foo’ OR 1=1` – we’d be able to log in regardless of if the password is correct.

We need a better way of writing queries with parameters determined by the code. Fortunately, database systems provide *parametrized queries*, where the database software is explicitly told "this is an input, with this value" so it knows not to treat it as SQL syntax. For example:

```r
username <- "'; DROP TABLE users;--"
password <- "walruses"

query <- sqlInterpolate(con,
                        "SELECT * FROM users WHERE username = ?user AND password = ?pass",
                        user=username, pass=password)

users <- dbGetQuery(con, query)
```

Strings of the form `?var` are replaced with the corresponding `var` in the arguments, but with any special characters escaped so they do not affect the meaning of the query. In this example, `query` is now

```sql
SELECT * FROM users WHERE username = '''; DROP TABLE users;--'
AND password = 'walruses'
```

Note how the single quote at the beginning of `username` is doubled there: that’s a standard way of escaping quotation marks, so Postgres recognizes it’s a quote inside a string, not the boundary of the string.

psycopg2 provides similar facilities:

```python
cur.execute("SELECT * FROM users "
            "WHERE username = %(user)s AND password = %(pass)s",
            {"user": username, "pass": password})
```

You should *always* use this approach to insert data into SQL queries. You may think it’s safe with your data, but at the least opportune moment, you’ll encounter nasal demons.

Database Schema Design Principles

The key design principle for database schema is to keep the design DRY – that is, **eliminate data redundancy**. The process of making a design DRY is called **normalization**, and a DRY database is said to be in “normal form.”

The basic modeling process:

1. Identify and model the entities in your problem
2. Model the relationships between entities
3. Include relevant attributes
4. Normalize by the steps below

Example

Consider a database to manage songs:

```
Album          Artist              Label     Songs
-------------- ------------------- --------- ---------------------------------
Talking Book   Stevie Wonder       Motown    You are the sunshine of my life,
                                             Maybe your baby, Superstition, ...
Miles Smiles   Miles Davis Quintet Columbia  Orbits, Circle, ...
Speak No Evil  Wayne Shorter       Blue Note Witch Hunt, Fee-Fi-Fo-Fum, ...
Headhunters    Herbie Hancock      Columbia  Chameleon, Watermelon Man, ...
Maiden Voyage  Herbie Hancock      Blue Note Maiden Voyage
American Fool  John Couger         Riva      Hurts so good, Jack & Diane, ...
...
```

This seems fine at first, but why might this format be problematic or inconvenient?

- Difficult to get songs from a long list in one column
- Same artist has multiple albums
- "Best Of" albums
- A few thoughts:
  - What happens if an artist changes names partway through his or her career (e.g., John Cougar)?
  - Suppose we want mis-spelled "Herbie Hancock" and wanted to update it. We would have to change every row corresponding to a Herbie Hancock album.
  - Suppose we want to search for albums with a particular song; we have to search specially within the list for each album.

The schema here can be represented as
`Album (artist, name, record_label, song_list)`
where `Album` is the *entity* and the labels in parens are its *attributes*.

To normalize this design, we will add new entities and define their attributes so **each piece of data has a single authoritative copy**.

Step 1. Give Each Entity a Unique Identifier

This will be its primary key, and we will call it `id` here.

Key features of a primary key are that it is unique, non-null, and it never changes for the lifetime of the entity.

Step 2. Give Each Attribute a Single (Atomic) Value

What does each attribute describe? What attributes are repeated in `Albums`, either implicitly or explicitly?

Consider the relationship between albums and songs. An album can have one or more songs; in other words, the attribute `song_list` is non-atomic (it is composed of other types, in this case a list of text strings). The attribute describes a collection of another entity – `Song`.

So, we now have two entities, `Album` and `Song`. How do we express these entities in our design? It depends on our model. Let's look at two ways this could play out.

1. Assume (at least hypothetically) that each song can only appear on *one* album. Then `Album` and `Song` would have a **one-to-many** relationship.

- `Album(id, title, label, artist)`
- `Song(id, name, duration, album_id)`

Question: What do our `CREATE TABLE` commands look like under this model?

2. Alternatively, suppose our model recognizes that while an album can have one or more songs, a song can also appear on one or more albums (e.g., a greatest hits album). Then, these two entities have a **many-to-many** relationship.

   This gives us two entities that look like:

   - `Album(id, title, label, artist)`
   - `Song(id, name, duration)`

   This is fine, but it doesn't seem to capture that many-to-many relationship. How should we capture that?

   - An answer: This model actually describes a *new* entity – `Track`. The schema looks like:
     - `Album(id, title, label, artist)`
     - `Song(id, name, duration)`
     - `Track(id, song_id, album_id, index)`

Step 3. Make All Non-Key Attributes Dependent Only on the Primary Key

This step is satisfied if each non-key column in the table serves to *describe* what the primary key *identifies.*

Any attributes that do not satisfy this condition should be moved to another table.

In our schema of the last step (and in the example table), both the `artist` and `label` field contain data that describes something else. We should move these to new tables, which leads to two new entities:

- `Artist(id, name)`
- `RecordLabel (id, name, street_address, city, state_name, state_abbrev, zip)`

Each of these may have additional attributes. For instance, producer in the latter case, and in the former, we may have additional entities describing members in the band.

Step 4. Make All Non-Key Attributes Independent of Other Non-Key Attributes

Consider `RecordLabel`. The `state_name`, `state_abbrev`, and `zip` code are all non-key fields that depend on each other. (If you know the zip code, you know the state name and thus the abbreviation.)

This suggests to another entity `State`, with name and abbreviation as attributes. And so on.

Exercise

Convert this normalized schema into a series of `CREATE TABLE` commands.

Appendix: PostgreSQL Primer

Getting Help

Type '\?' at the prompt to get a list of meta-commands (these are system, not SQL commands).

A few of these are quite common:

- `\h` provides help on an SQL command or lists available commands
- `\d` list or describe tables, views, and sequences
- `\l` lists databases
- `\c` connect to a different database
- `\i` read input from a file (like source)
- `\o` send query output to a file or pipoe
- \! execute a shell command
- `\cd` change directory
- `\copy` copy data into a table
- `\q` quit psql

Entering SQL Statements

SQL consists of a sequence of *statements*.

Each statement is built around a specific command, with a variety of modifiers and optional clauses.
SQL statements can span several lines, and all SQL statements end in a semi-colon (;).
Keep in mind: strings are delimited by single quotes 'like this', *not* double quotes "like this".
SQL comments are lines starting with `--`.
To get help:

- You can get brief help on any SQL command with `\h <command>`.
- You can get detailed and helpful information on any aspect of postgres through the online documentation.
- The stat server is running version 9.2, that that will be updated if needed.

Appendix: A Few Advanced Maneuvers, Part I

Advanced Example: Text Processing

The purpose of this example is to give a flavor for how postgres can be used for novel data types, like text data. While you may want some of this logic in your program, there can be value in keeping it close to the data as well.

Load *extensions* `fuzzystrmatch`, `cube`, and `pg_tgrm` by doing the following postgres commands:

```
create extension fuzzystrmatch;
create extension cube;
create extension pg_trgm;
```

Get the file `ClassFiles/week4/movies_data.sql` from the `documents` repository and import it into your running psql with `\i movies_data.sql` or something similar.

The `LIKE` (and `ILIKE` for *case-insensitive matching*) is a simple SQL facility for text searching. Try this:

```sql
select title from movies where title ilike 'stardust%';
```

This searches for text that matches a specified pattern: in this case, the word stardust followed by *any number of other characters*. In these patterns `%` and `_` are **wildcard** characters: `%` matches any number (zero or more) of characters and `_` matches any single character. The following forces at least one character after the word stardust:

```sql
select title from movies where title ilike 'stardust_%';
```

These are useful commands, but the types of patterns they can capture are rather simple. A more powerful pattern language is given by **regular expressions**. We'll see some examples today and talk about them in more detail as we proceed through the semester.

The regular expression matching operator is `~`, which is preceded by `!` to negate the search and followed by `*` to make the search case insensitive. Not especially mnemonic!

```sql
select count(*) from movies
       where title !~* '^the.*';
```

This counts how many movies (in table `movies`) *do not start* with the word 'the'. The following counts the movies that have the word 'the' anywhere in the title, where the case of the word is ignored:

```sql
select count(*) from movies
       where title ~* 'the';
```

Compare what happens when we don't ignore case:

```sql
select count(*) from movies
       where title ~ 'the';
```

Regular expressions offer a rich variety of possible patterns; PostgreSQL supports essentially the POSIX version of regular expressions.

We can make searches like this faster by creating an index:

```sql
create index movie_title_pattern on movies (lower(title) text_pattern_ops);
```

This optimizes title searches for case insensitive pattern matching.

The **Levenshtein distance** measures how far apart two words are lexically, effectively counting the number of simple steps (character insertions, deletions, swaps, etc) required to transform one word into another. We can compute this with the `levenshtein()` function:

```sql
select levenshtein('guava', 'guano');
select levenshtein('bat','fads') as fads,
       levenshtein('bat','fad')  as fad,
       levenshtein('bat','fat')  as fat,
       levenshtein('bat','bad')  as bad;
```

This allows us to "fuzzy" search our text data:

```sql
select movie_id, title from movies
  where levenshtein(lower(title), lower('a hard day nght')) <= 3;
select movie_id, title from movies
  where levenshtein(lower(title), lower('a hard day nght')) <= 8;
```

A **trigram** is a group of three consecutive characters taken from a string:

```sql
select show_trgm('Avatar');
```

We can create a trigram index for faster string searching based on the number of matching trigrams.

```sql
create index movies_title_trigram on movies
  using gist(title gist_trgm_ops);
select title from movies where title % 'Avatre';
```

It will be faster even if not too noticeably for this data size.

We can also do full-text searches using a bag of words style match with the `@@` operator. For example:

```sql
select title from movies where title @@ 'night & day';
```

This uses as its lexicon a large dictionary in a specified language (English by default on our server). The bag of words and query vector are stored in data types `ts_vector` and `ts_query`, respectively. We can see the representations using some helper functions:

```sql
select to_tsvector('A Hard Day''s Night'), to_tsquery('english', 'night & day');
```

Notice that the 'A' is missing in the first column: simple, high-frequency words (called 'stop words') are dropped by default. It is possible to configure the list of stop words use (or even make it empty, as is more current common practice).

We can create indexes to speed this search:

```sql
explain select * from movies where title @@ 'night & day';
create index movies_title_searchable on movies
       using gin(to_tsvector('english', title));
explain select * from movies where title @@ 'night & day';
explain select * from movies
        where to_tsvector('english', title) @@ 'night & day';
```

The index does nothing (but add overhead) for the first select because we did not specify the english ts-vector directly. The last search makes it clear, and postgres uses the index to reduce cost substantially.

Finally, we look at multi-dimensional attributes. The `genre` column of `movies` gives a score to each movie in several categories of movie genres as defined by the table `genres`.

Using the `cube` extension, we can search this attribute in useful ways:

```
select * from genres;
select * from movies where title @@ 'star & wars';
select name, cube_ur_coord('(0, 7, 0, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 10, 0, 0, 0)', position) as
  from genres g
  where cube_ur_coord('(0, 7, 0, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 10, 0, 0, 0)', position) > 0;
```

This lists what genres the movie Star Wars belongs to (and with what scores).

Now we can find movies that have similar genre assignments:

```
select title,
       cube_distance(genre, '(0, 7, 0, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 10, 0, 0, 0)') as dist
       from movies
       ORDER BY dist
       LIMIT 16;
```

This gives the 16 movies closest in genre to 'Star Wars'.

We can also look at movies in a bounding box using the `cube_enlarge()` function and the `@>` special contains operator. The following gives the 10 closest movies to 'Mad Max' in a size 5 cube (in the 18-dimensional genre space) around that movies genre vector:

```
select m.movie_id, m.title
       from movies as m,
           (select genre, title from movies
                 where title = 'Mad Max') as s
       where cube_enlarge(s.genre, 5, 18) @> m.genre AND
            s.title != m.title
       order by cube_distance(m.genre, s.genre)
       limit 10;
```

And there's much more in this vein...

Subqueries

You can use select query in `()`'s within a `WHERE` clause:

```
select title, starts from social_events where
    venue_id in (select id from venues where name ~ 'room');
```

There are various other functions/operators that can be used on subqueries as well, such as `in`, `not in`, `exists`, `any`, `all`, and `some`.

For example, this looks like an inner join:

```
select title, starts from social_events where
    exists (select 1 from venues where id = social_events.venue_id);
```

How would you do this with a join?

Aggregate Functions and Grouping

Aggregate functions operate on one or more attributes to produce a summary value. Examples: `count`, `max`, `min`, `sum`.

Add Pittsburgh (and perhaps some other cities) to the cities table. Add two or more Pittsburgh venues and one or more social events at each of those venues.

Let's count the number of social events at one of those venues:

```sql
select count(*) from social_events where venue_id = 4;
```

Exercise: Given a city name (like 'Pittburgh'), count the total number of social events within that city.

We often want to apply aggregate functions not just to whole columns but to **groups of rows** within columns. This is the province of the `GROUP BY` clause.

We could write

```sql
select count(*) from events where venue_id = 1;
select count(*) from events where venue_id = 2;
select count(*) from events where venue_id = 3;
select count(*) from events where venue_id = 4;
```

But that is tedious and gives four separate scalars.

Instead, we use the `GROUP BY` modifier:

```sql
select venue_id, count(*) from events group by venue_id;
```

You can apply conditions on grouped queries. Instead of `WHERE` for those conditions, you use `HAVING`, with otherwise the same syntax. Short version: `WHERE` select rows, and `HAVING` selects groups.

```sql
select venue_id, count(*) from events group by venue_id
    having venue_id is not NULL;
select venue_id, count(*) from events group by venue_id
    having count(*) > 1 AND venue_id is not NULL;
```

Window Functions

Aggregate functions let you summarize mulitiple rows, but they summarize *with a single value for each group*. **Window functions** are similar except they let us **tag each row with the summary**.

(Make sure you have at least one venue with more than one social event in the example to follow. For instance,

```sql
insert into social_events (title, venue_id)
  values ('Valentine''s Day Party', 1), ('April Fool''s Day Party', 1);
```

Notice the double " to escape the single quote.)

Now, compare

```
select venue_id, count(*) from social_events group by venue_id
    order by venue_id;
select venue_id, count(*) OVER (PARTITION BY venue_id)
    from social_events order by venue_id;
```