SQL: The Sequel

Christopher R. Genovese

Department of Statistics & Data Science

Tue 19 Nov 2024
Session #22

Plan

Activity: SQL Exercises

Plan

Activity: SQL Exercises

Joins and Foreign Keys

Plan

Activity: SQL Exercises

Joins and Foreign Keys

Variations

Plan

Activity: SQL Exercises

Joins and Foreign Keys

Variations

Issues on SQL in Code

Plan

Activity: SQL Exercises

Joins and Foreign Keys

Variations

Issues on SQL in Code

Schema Design Principles

Announcements

- Continuation Session from Last Time
- **Reading**: From last time:
  - Code Design with Types and Concepts (`data` means `type` and other syntax variations, but you'll get it.)
  - Some possibly helpful sites; check one out:
    - pgexercises.com
    - sqlbolt.com
    - sql-practice.com – Structured Practice
    - mystery.knightlab.com – An SQL Murder Mystery
    - selectstarql.com
  - Deriving the Z-combinator and Classes with the Z Combinator
- **Homework**: `parser-combinators`, next one of:
  - `classification-tree-basic`
  - `sym-spell`
  - `migit3`
  - `regex-derivatives`
  - Alternates: `laser-tag`, `dominoes`

Plan

Activity: SQL Exercises

Joins and Foreign Keys

Variations

Issues on SQL in Code

Schema Design Principles

Activity: SQL Exercises

See other file

Plan

Activity: SQL Exercises

Joins and Foreign Keys

Variations

Issues on SQL in Code

Schema Design Principles

Links Between Tables

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*.

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:

```
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>`.

Foreign Keys

Now try this

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

Joins

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

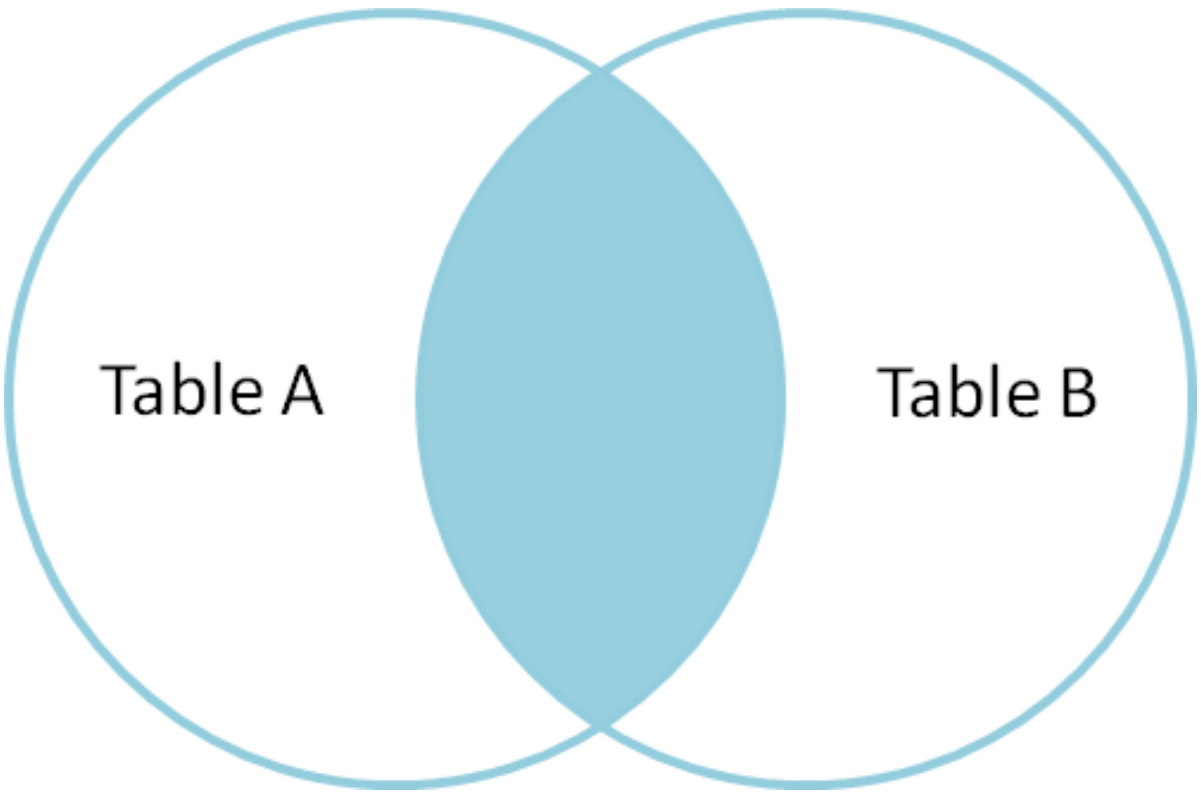
```
create table A (id SERIAL PRIMARY KEY, name text);
insert into A (name)
       values ('Pirate'),
              ('Monkey'),
              ('Ninja'),
              ('Flying Spaghetti Monster');

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;
```

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.)

```
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:

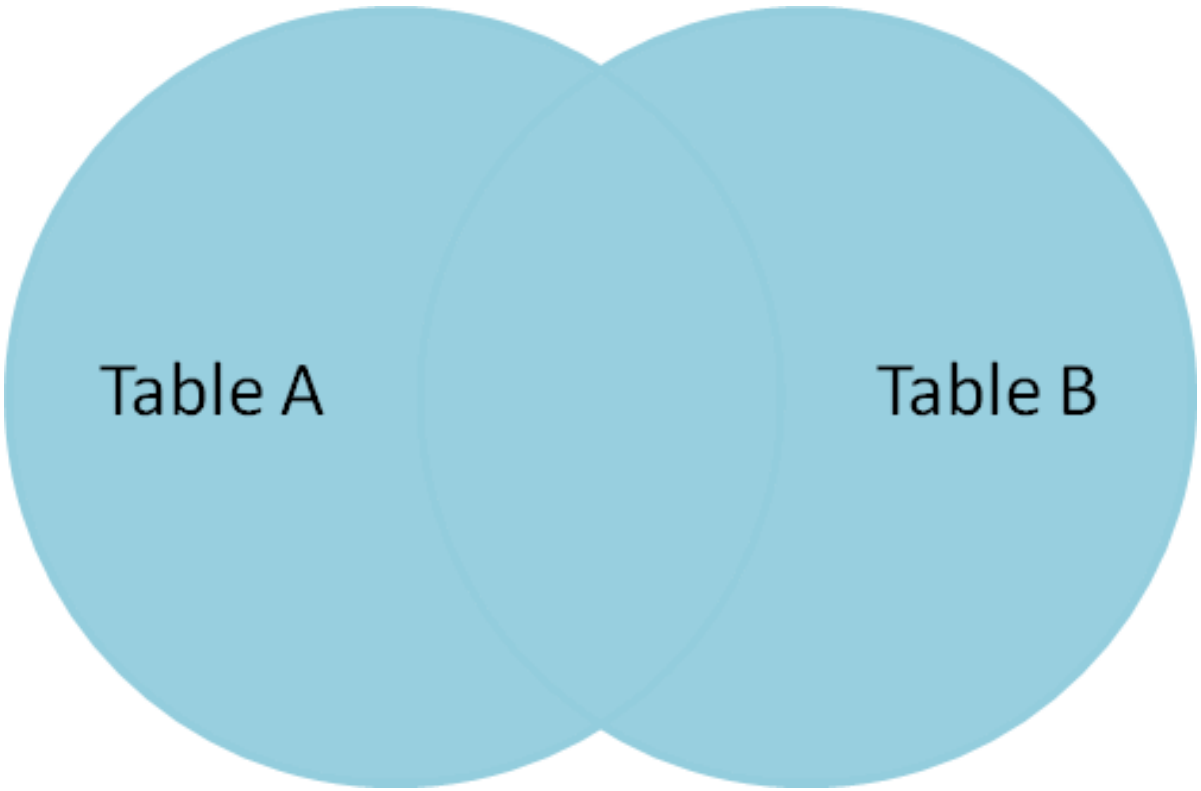


Full Outer Join

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

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

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

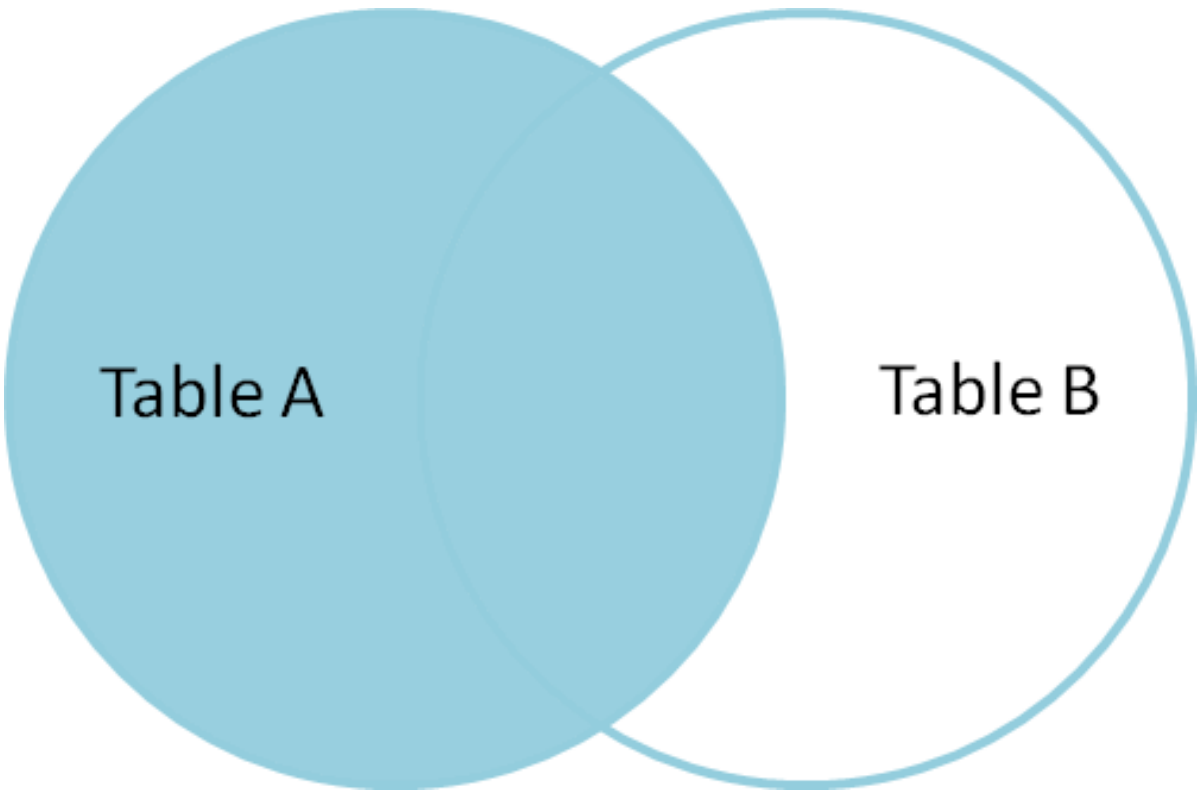


Left Outer Join

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.)

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

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

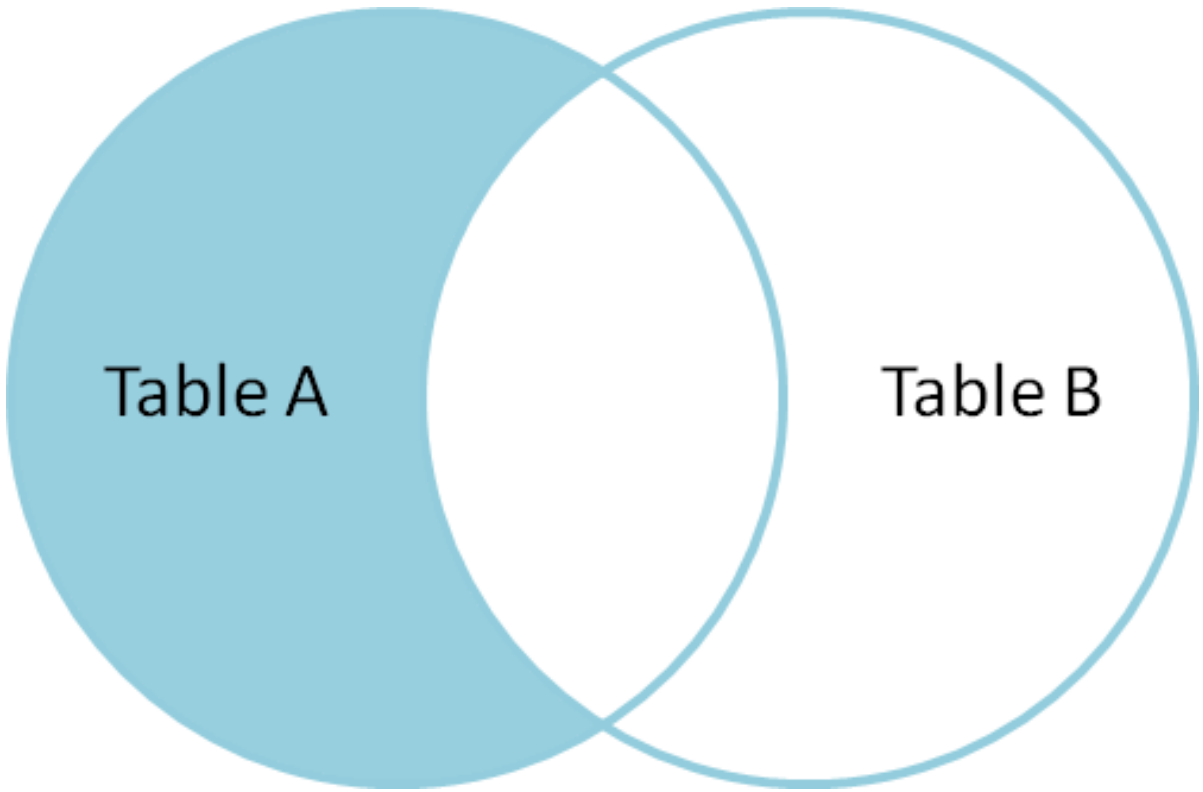


Set Difference

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

```
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:

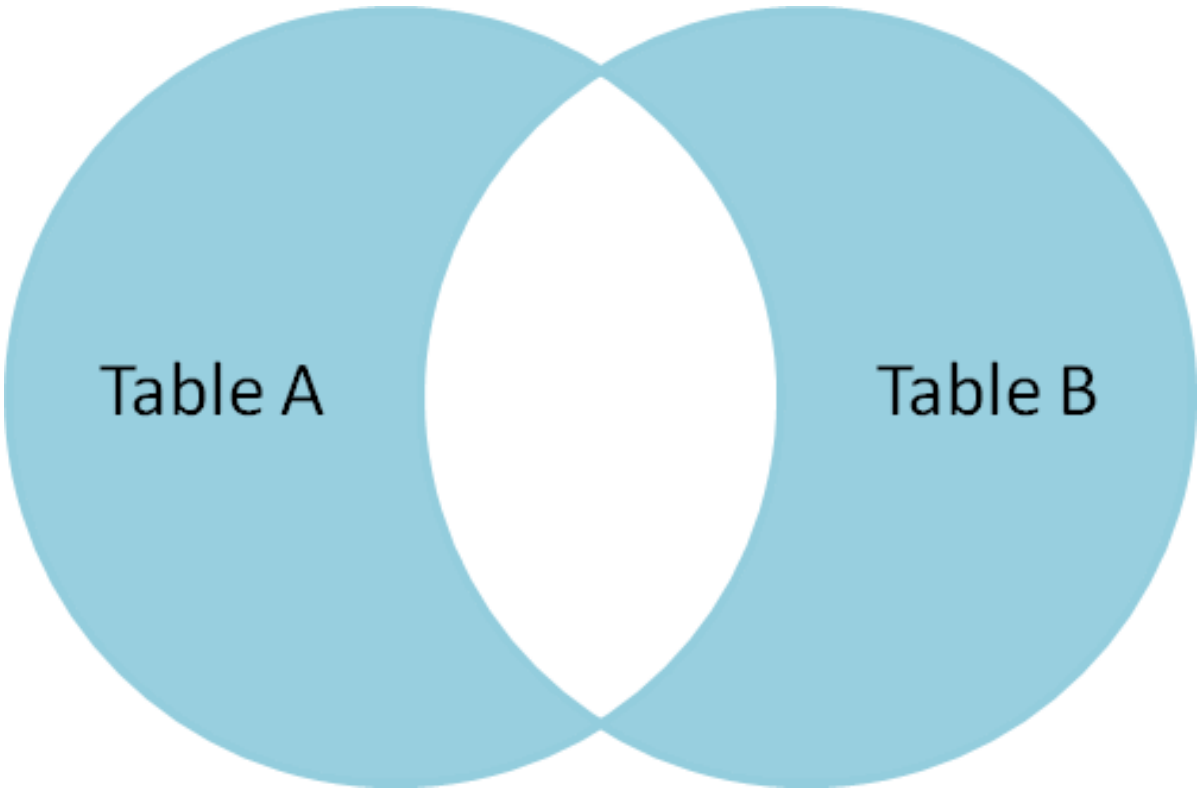


Symmetric Difference

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 corresponds to a *symmetric set difference* operation $A \Delta B$



A Simple join Exercise

Using the `cities` and `countries` tables created in the commands file, do the following:

1. List city name, postal code, and country name for each city.
2. List city name, country, and address as a valid string.
   (The `concat()` function takes multiple strings as arguments and concatenates them together.)

Plan

Activity: SQL Exercises

Joins and Foreign Keys

Variations

Issues on SQL in Code

Schema Design Principles

Aggregate Functions and Grouping

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

For example, let's calculate the average score obtained by students:

```
select avg(score) from events;
```

This produces a single row: the average score. Any aggregate function takes many rows and reduces them to a single row. This is why you **can't** write this:

```
select persona, avg(score) from events;
```

Try it; why does Postgres complain?

Aggregate Functions and Grouping

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. It groups the data according to a specific value, and aggregate functions then produce a single result **per group**.

For example, if I wanted the average score for each separate user, I could write:

```
select persona, avg(score) as mean_score
from events
group by persona
order by mean_score desc;
```

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.

```
select persona, avg(score) as mean_score
from events
where moment > '2020-10-01 11:00:00'::timestamp
group by persona
having avg(score) > 50
order by mean_score desc;
```

Common Table Expressions

Common Table Expressions, introduced via `with`, introduces temporary tables that can be used for a query.

Simple example:

```
with a as (
    select x from generate_series(1, 10) as x
), b as (
    select y from generate_series(11, 20) as y
)
select * from a, b;
```

Common Table Expressions (cont'd)

Subsequent terms in with can refer to earlier terms. What does this do?

```
WITH regional_sales AS (
    SELECT region, SUM(amount) AS total_sales
    FROM orders
    GROUP BY region
), top_regions AS (
    SELECT region
    FROM regional_sales
    WHERE total_sales > (SELECT SUM(total_sales)/10 FROM regional_sales)
)
SELECT region,
       product,
       SUM(quantity) AS product_units,
       SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

Common Table Expressions (cont'd)

CTEs can be *recursive*.

Try this:

```
WITH RECURSIVE t(n) AS (
    VALUES (1)      -- Base case
  UNION ALL
    SELECT n+1 FROM t WHERE n < 100   -- Inductive Case
)
SELECT sum(n) FROM t;
```

Common Table Expressions (cont'd)

CTEs can be fairly powerful. Challenge CTE for Binary Tree traversal (DFS, BFS).

Common Table Expressions (cont'd)

CTEs can be fairly powerful. Challenge CTE for Binary Tree traversal (DFS, BFS).

```
create table tree (
    id integer,
    data integer,
    left_child integer,
    right_child integer
);
insert into tree (id,,data, left_child, right_child)
  values (1, 10, 2, 3),
         (2, 20, 4, 5),
         (3, 30, 6, 7),
         (4, 40, 8, NULL),
         (5, 50, NULL, 9),
         (6, 60, NULL, NULL),
         (7, 70, NULL, NULL),
         (8, 80, NULL, NULL),
         (9, 90, NULL, NULL);
```

Common Table Expressions (cont'd)

CTEs can be fairly powerful. Challenge CTE for Binary Tree traversal (DFS, BFS).

```
WITH RECURSIVE search_tree(id, data, left_child, right_child, path) AS (
    SELECT t.id, t.data, t.left_child, t.right_child, ARRAY[t.id]
    FROM tree t
    WHERE t.id = 1  -- Root is base case
  UNION ALL
    SELECT t.id, t.data, t.left_child, t.right_child, path || t.id
    FROM tree t, search_tree st
    WHERE t.id = st.left_child OR t.id = st.right_child
)
SELECT * FROM search_tree ORDER BY path;
```

Common Table Expressions (cont'd)

CTEs can be fairly powerful. Challenge CTE for Binary Tree traversal (DFS, BFS).

```
WITH RECURSIVE search_tree(id, data, left_child, right_child, path, depth)
    SELECT t.id, t.data, t.left_child, t.right_child, ARRAY[t.id], 0
    FROM tree t
    WHERE t.id = 1   -- Root is base case
  UNION ALL
    SELECT t.id, t.data, t.left_child, t.right_child, path || t.id, depth +
    FROM tree t, search_tree st
    WHERE t.id = st.left_child OR t.id = st.right_child
)
SELECT * FROM search_tree ORDER BY depth;
```

Plan

Activity: SQL Exercises

Joins and Foreign Keys

Variations

Issues on SQL in Code

Schema Design Principles

Storing Your Password

The code last time stores your password right in the source file. This is a **bad idea**.

If the code is ever shared with anyone, posted online, or otherwise revealed, anyone who sees it now has your database username and password and can view or modify any of your data.

If you commit the file to Git, your password is now in your Git history **forever**.

Fortunately, there are ways to work around this.

Storing Your Password: R

Run this R code:

```
file.edit(file.path("~", ".Rprofile"))
```

This will create a file called `~/.Rprofile` in your home directory and open it for editing. In this file, write something like

```
DB_USER <- "yourusername"
DB_PASSWORD <- "yourpassword"
```

Save and close the file. Start a new R session. The `DB_USER` and `DB_PASSWORD` variables will be defined in *any R script you run*, so you can use them in your code.

And since the `.Rprofile` is not in your assignments repository, you won't accidentally commit it to your Git history.

Storing Your Password: Python

Python doesn't have something like `~/.Rprofile`. Instead, when you do an assignment that requires SQL access, create a separate file `credentials.py` defining your username and password in variables.

You can `import credentials` and then use `credentials.DB_USER` and `credentials.DB_PASSWORD` in your other code files.

To avoid accidentally committing `credentials.py`, create (or modify, if it exists) a file called `.gitignore` in the root of your `assignments` repository.

Add the line `credentials.py` to it. This will make Git ignore any files called `credentials.py`, so you don't accidentally commit them.

You can use the `cryptography` or `bcrypt` packages to encrypt your stored password data as well, though the keys will need to be accessible in most cases.

SQL in Unit Tests

Question: If you have secret SQL credentials, but your unit tests need to check functions that access the database, how can your tests run on our "public" server? How can anyone else run your tests?

Answer: They can't. This is fine for our class; in a business environment, you would use a secret-management system to ensure code can always access the passwords it needs, or set up special test databases for development. But we're keeping things simple here.

So instead you should do the following:

1. Ensure SQL access is limited to the functions that really need it. Most of your code doesn't need to access SQL – it just needs to be provided the right data in a convenient form. Only a few functions need to run SQL queries and put the data into the right form for the rest of your code to use. If you follow this design principle, most of your functions can still be tested.
2. Set your tests to be skipped if the credentials are not available.

For point 2, most unit testing frameworks provide ways of marking tests as being skipped in certain circumstances.

SQL in Unit Tests

For example, in Python:

```
import pytest
import psycopg2

try:
    import credentials
except ImportError as e:
    no_database = True
else:
    no_database = False

@pytest.mark.skipif(no_database, reason="No database credentials available")
def test_some_database_thingy():
    conn = psycopg2.connect(...)

    # do stuff
```

SQL in Unit Tests

In R:

```
library(testthat)

test_that("database access works", {
  skip_if_not(exists("DB_USER"))

  # do some database stuff
})
```

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?

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.

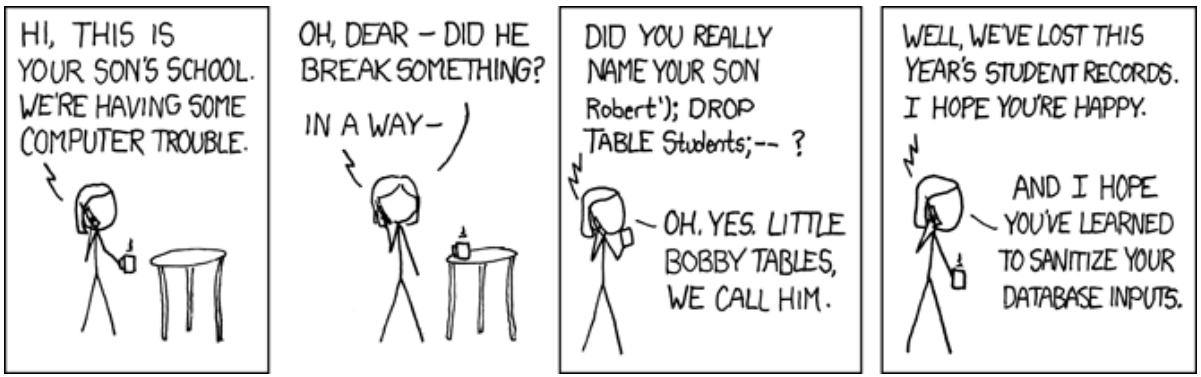
```
result <- dbSendQuery(paste0("SELECT * FROM users ",
                             "WHERE username = '", username, "' ",
                             "AND password = '", password, "'"))
```

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'
```

Practicing Safe SQL (cont'd)

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



(source: xkcd)

Practicing Safe SQL (cont'd)

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. . . .

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 without knowing the right password!

We need a better way of writing queries. Database systems provide ***parametrized queries***, where you are explicit about input parameters that are not SQL syntax. For example:

```
username <- "'; DROP TABLE users;--"
password <- "walruses"
query <- sqlInterpolate(con,
                        "SELECT * FROM users WHERE username = ?user AND password =
                        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

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

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.

Practicing Safe SQL (cont'd)

psycopg2 provides similar facilities:

```
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.

Plan

Activity: SQL Exercises

Joins and Foreign Keys

Variations

Issues on SQL in Code

Schema Design Principles

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?

Example (cont'd)

- 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.

Step 1. Give Each Entity a Unique Identifier

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**.

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.

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

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.

The End