This is your data on ACID

Christopher R. Genovese

Department of Statistics & Data Science

Thu 06 Nov 2025
Session #19

Plan

Intro

Plan

Intro

Database Concepts

Plan

Intro

Database Concepts

ACID

Plan

Intro

Database Concepts

ACID

SQL

Plan

Intro

Database Concepts

ACID

SQL

Joins and Foreign Keys

Plan

Intro

Database Concepts

ACID

SQL

Joins and Foreign Keys

A Deeper Dive

Announcements

- In documents: lecture notes, fpc updated draft
- Download postgresql
- Plan
- **Homework**:
  - **ATTN** assignment due .
  - **zippers** assignment due tomorrow.

Plan

How do you store your data?

You have a data set that you want to use in your work. How do you store it so that you can use it most effectively?

A few common scenarios:

How do you store your data?

You have a data set that you want to use in your work. How do you store it so that you can use it most effectively?

A few common scenarios:

- **Keep data in a text file** (e.g., CSV file)

  Pros Easy to read, easy to edit, easy to archive and transfer

  Cons No checking of data, low data density, hard to search/query, must keep in sync with version used in software, requires separate file for documentation that must be kept in sync

  Questions

  1. If you read the data into an R data frame and change it, which is the authoritative copy of the data?
  2. If you make a mistake editing the file, what happens?
  3. What is needed to put a comma or newline in a CSV field?

How do you store your data?

You have a data set that you want to use in your work. How do you store it so that you can use it most effectively?

A few common scenarios:

- **Keep data in an encoded file** (e.g., `.Rdata`/`.rds`, HDF5, Numpy/`.npy`)

  Pros High data density (i.e., compact)

  Cons Same problems as for text file but more intense because you need a program to read the data

How do you store your data?

You have a data set that you want to use in your work. How do you store it so that you can use it most effectively?

A few common scenarios:

- **Keep data in an R/Python data frame**

  Pros Easy to use from a program, can attach metadata (e.g., documentation)

  Cons Not persistent, requires translation to use from another platform

How do you store your data?

You have a data set that you want to use in your work. How do you store it so that you can use it most effectively?

A few common scenarios:

- **Keep data in memory**

  Pros Very fast access, flexible structuring of the data
  Cons Not persistent, not accessible in parallel

Databases

A **database** is a structure for organizing information that can be *efficiently* accessed, updated, and managed – at scale. It is designed to address the many problems of these more informal approaches.

Features:

- Efficient operations
- Strong data-integrity guarantees
- Concurrent transactions (some)
- Rich data descriptions
- Remote access
- Distributed representations (some)
- Customizable and optimizable in various ways

Many types, many models, including:

- Relational (Examples: PostgreSQL, mySQL, Oracle, SQLite)
- Key-Value (Examples: Redis, Riak, Memcached)
- Columnar (Examples: HBase, Cassandra, Accumulo)
- Immutable (Example: DAtomic)
- Document (Examples: MongoDB, CouchDB, DynamoDB)
- Graph (Example: Neo4j)

We will focus on Relation DBs.

Relational Databases

Relational Databases have been the dominant model for databases since the 1970s and are still very important. There are many new types of databases now available that offer various trade-offs in performance and use.

The name comes from a way to think about tables and data frames.

From Interlude F: An **n-ary relation** on sets $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$ is a set $\mathcal{R} \subset \mathcal{A}_1 \times \mathcal{A}_2 \times \cdots \times \mathcal{A}_n$. The sets $\mathcal{A}_i$ are called the *domains* of the relation, and n is its *arity*.

Relational databases view **tables** as instances of relations

Attribute 1	Attribute 2	...	Attribute n
v_1	v_2	...	v_n

Attribute i has a **type** $\mathcal{A}_i$, and each "row" of the table is an element

$$\langle v_1, v_2, \ldots, v_n \rangle \in \mathcal{A}_1 \times \mathcal{A}_2 \times \cdots \times \mathcal{A}_n.$$

As long as the rows are unique (which we will require implicitly or explicitly), a **table** is just a subset of such tuples – that is, a **relation**.

Relational Databases (cont'd)

Relational databases tend to be **design-first** systems:

1. Design a *schema* for your data
2. Check that your schema satisfies the properties of *normalization*
3. Create *tables* (aka relations) corresponding to the entities in your schema
4. Enter *data* into those tables.

A properly designed schema allows flexible and powerful queries.

Relational Databases (to be called RDBs or databases for the next few classes) are commonly manipulated and queried using a (mostly) standardized language called **SQL**, which stands for Structured Query Language.

We will be using a powerful, open-source RDB system called **PostgreSQL** (aka "postgres"). It is fast, flexible, reliable, and can handle large data sets. It is highly compliant to the ANSI-standard for SQL and has some nice extensions. And it has been used successfully for many production systems over many years.

How do you use your data?

With rich data sets, we have the potential to answer many different kinds of questions. As we work with the data, we construct new view of the data, new summaries, new statistics and then pose new questions.

This involves cycles of **query** and **update**.

The basic operations on our data involve adding variables, changing values, creating summaries, selecting data that meet certain criteria, adding or removing cases that meet those criteria, and establishing relationships between different entities in our data.

These fundamental operations are called **CRUD** operations:

1. Create
2. Read
3. Update
4. Delete

These fundamental operations are supported in many frameworks and platforms and contexts, with varying syntax.

Examples: `http`, `dplyr`, arrays and dictionaries and environments

Even though we will focus on SQL, the concepts apply quite broadly.

Plan

Intro

Database Concepts

ACID

SQL

Joins and Foreign Keys

A Deeper Dive

Database Concepts: Client-Server Model

Most (not all!) SQL database systems are based on a client-server model.

Server A process running continuously on some server. Accepts requests (potentially from many users at the same time) to update and query data, and stores the canonical version of the database.

Client Any program that can connect to the server, send queries, and receive results from it.

The client and server need not be on the same computer, or even on the same continent.

Often, companies with large databases will have a central database server with huge hard drives and lots of memory; business systems (like an inventory tracker or logistics system) will send their queries to this database to keep it up-to-date, while analysts will sit on their laptops and send queries to produce reports for their bosses.

A key feature of the client-server model is that the server can serve multiple clients at the same time. The server goes to extraordinary efforts to ensure that it can process the analyst's queries while simultaneously recording every new sale on the company's website – and while keeping all results consistent and complete.

Database Concepts: Tables (relations) and Schemas

The basic unit of data storage in an RDB is the **table**. Tables are also sometimes called *relations*, as we saw earlier. A single database typically has many tables.

A table is defined by its *attributes*, represented by columns, each of which has (at least) a **name** and a **type**.

In practice, it is common for a table to represent an ***entity*** being modeled with our data.

Each *row* of a table – an *instance* of the relation – defines a mapping from attribute names to values. Unlike data frames in R or Python, the rows **do not have an order**.

There is no notion of row indices unless you create a column for that purpose, though a basic practice is to provide a unique id for every instance.

A **schema** is the specification of the various tables in our databases, including the names and types of their attributes and the links or references across tables. Some database architectures encourage *schema design* before populating the databases; others are looser. There are (dis)advantages to both.

Database Concepts: Attribute Types

Table: `Events`

id	time	persona	element	latency	score	feedback
17	2015-07-11 09:42:11	3271	97863	329.4	240	Consider...
18	2015-07-11 09:48:37	3271	97864	411.9	1000	
19	2015-07-08 11:22:01	499	104749	678.2	750	The mean is...
22	2015-07-30 08:44:22	6742	7623	599.7	800	Try to think of...
24	2015-08-04 23:56:33	1837	424933	421.3	0	Please select...
32	2015-07-11 10:11:07	499	97863	702.1	820	What does the...
99	2015-07-22 16:11:27	24	88213	443.0	1000	

What are the attribute names and types for this table?

The **type** of a piece of data describes the set of possible values that data can have and the operations that can apply to it.

In an RDB, we specify the type of each data attribute in advance in the schema.

Database Concepts: Attribute Types (cont'd)

Postgres, for instance, supports a wide variety of data types, including:

- Numeric Types, such as integers, fixed-precision floating point numbers, arbitrary precision real numbers, and auto-incrementing integer (`serial`).
- Text, including fixed-length and arbitrary character strings.
- Monetary values
- Date and Time Stamps
- Boolean values
- Geometric types, such as points, lines, shapes
- Elements in sets
- JSON structures

When you can, *use the built-in types*.

For example, store dates and times using the date and time types, **not** as strings. Databases usually support functions to operate directly on these types and will ensure they are valid (i.e. you can't enter February 29, 2015 at 15:44pm).

See the Postgres documentation on "Data Types" for details and for more examples.

Database Concepts: References

We think of tables as representing entities that we are modeling in our problem.

For example, each row of `Events` represents a single "event" of some sort; each persona in the `Personae` table represents a single student in a single class (in a specified term).

We link tables to define **relationships among entities**.

For example, each `persona` is linked to many events, while each event has a single associated `persona` and `element`.

A good *design* of the database tables can make it more efficient to query these relationships.

Database Concepts: Unique, primary, and foreign keys

It is valuable (even necessary) in practice for each row of a database table to be distinct. To that end, it is common to define a **unique key** – one or more attributes whose collective values uniquely identify every row.

In the `Events` table above, `id` is a unique key consisting of a single attribute.

There may be more than one unique key in a table, some resulting from the joint values of several attributes. One of these keys is usually chosen as the **primary key** – the key that is used in queries and in other tables to identify particular rows.

In the `Events` table, `id` is also the primary key for the table.

In practice, the primary key is often an auto-incrementing, or `serial`, integer.

When a table's primary key is used as an attribute in another table, it acts as a *link* to a row in the first table.

A key used in this way is called a **foreign key**. Columns that store foreign keys are used for linking and cross-referencing tables efficiently.

In the Events table above, the `persona` and `element` attributes are foreign keys, referencing other tables, which I have not shown you.

Plan

Intro

Database Concepts

ACID

SQL

Joins and Foreign Keys

A Deeper Dive

ACID guarantees

An RDB stores our data, and we read and operate on that data through requests sent to the database. These requests are formally called **transactions**.

Modern RDBs may receive many transactions at once, often operating on the same pieces of data. Particular care is needed to ensure that transactions are performed reliably and consistently.

For example, consider what would happen in the following cases:

- A transaction for a commercial payment is transfering money from your bank account and to another account. But the process ends after the money is deducted from one account but before adding it to the other.
- A similar transaction completes *just* before the power goes out in the server room
- A similar transaction completes even though you don't have enough money in your account to make the payment.

These are all boundary cases, but they can happen. And if they do, the viability of the entire system can be compromised.

ACID guarantees (cont'd)

So, RDBs are designed to make several strong guarantees about their performance, the so-called ACID guarantees:

- **Atomic**

  A transaction either succeeds entirely or fails leaving the database unchanged.

- **Consistent**

  A transaction must change the database in a way that maintains all defined rules and constraints.

- **Isolated**

  Concurrent execution of transactions results in a transformation that would be obtained if the transactions were executed serially.

- **Durable**

  Once a transaction is committed, it remains so even in the face of crashes, power loss, and other errors.

This is another advantage of RDBs over ad hoc data storage.

Plan

Intro

Database Concepts

ACID

SQL

Joins and Foreign Keys

A Deeper Dive

SQL

SQL, for *Structured Query Language*, is the standard language for interacting with relational databases.

It is ultimately quite flexible and powerful, but it does have a 1970s feel to it.

We can interact with our databases through our programming language of choice, but all of this is mediated by SQL. So it is worth learning at least the basics.

SQL consists of a sequence of *statements*.

Each statement is built around a specific command, with an often large variety of modifiers and optional clauses.

Keep in Mind:

- Command names and modifiers are **not** case-sensitive.
- SQL statements can span several lines, and **all** SQL statements end in a semi-colon (`;`).
- Strings are delimited by **single quotes** 'like this', *not* double quotes "like this".
- SQL comments are lines starting with `--`.

PostgreSQL

We use **PostgreSQL** as our database engine. It's effective, battle tested, open source, and has a rich body of extensions. There are other engines, all quite similar.

We can interact with postgres manually with the `psql` command, which opens up a REPL.

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

A few of these are quite common:

- `\q` quit psql
- `\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 pipe
- `\!` execute a shell command
- `\cd` change directory

The PostgreSQL manual is a well-written and valuable resource.

A Simple Example

You can try the following (or copy it from the notes or `commands-1.sql`).

```
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)
);

INSERT INTO products (name, price, sale_price) values ('furby', 100, 95);
insert into products (name, price, sale_price)
       values ('frozen lunchbox', 10, 8),
              ('uss enterprise', 12, 11),
              ('spock action figure', 8, 7),
              ('slime', 1, 0.50);

select * from products;
select name, price from products;
select name as product, price as howmuch from products;
```

Typing `\d` at the prompt will show list our new table.

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.

First, you need to create a *connection* object, which represents your connection to the server.

```
library(RPostgreSQL)
con <- dbConnect(PostgreSQL(), user = "yourusername", password = "yourpassword",
                 dbname = "yourusername", host = "sculptor.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`:

```
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.

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

Shortcut: `dbGetQuery` runs a query, fetches all 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:

```
import psycopg2

conn = psycopg2.connect(host="sculptor.stat.cmu.edu", database="yourusername",
                        user="yourusername", password="yourpassword")
cur = conn.cursor()
baz = "walrus"
spam = "penguin"

cur.execute("INSERT INTO foo (bar, baz, spam) "
            "VALUES (17, %s, %s)", (baz, spam))
```

Notice the use of interpolation to put variables into the query – see the Safe SQL section below.

SQL in Python (cont'd)

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.

If you use pandas for your data frames, you can also convert the results of any query directly into a pandas data frame:

```
import pandas as pd
d = pd.read_sql_query("SELECT foo, bar FROM events WHERE id = %(id)s",
                      conn, params = {'id': 17})
```

Plan

Intro

Database Concepts

ACID

SQL

Joins and Foreign Keys

A Deeper Dive

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.

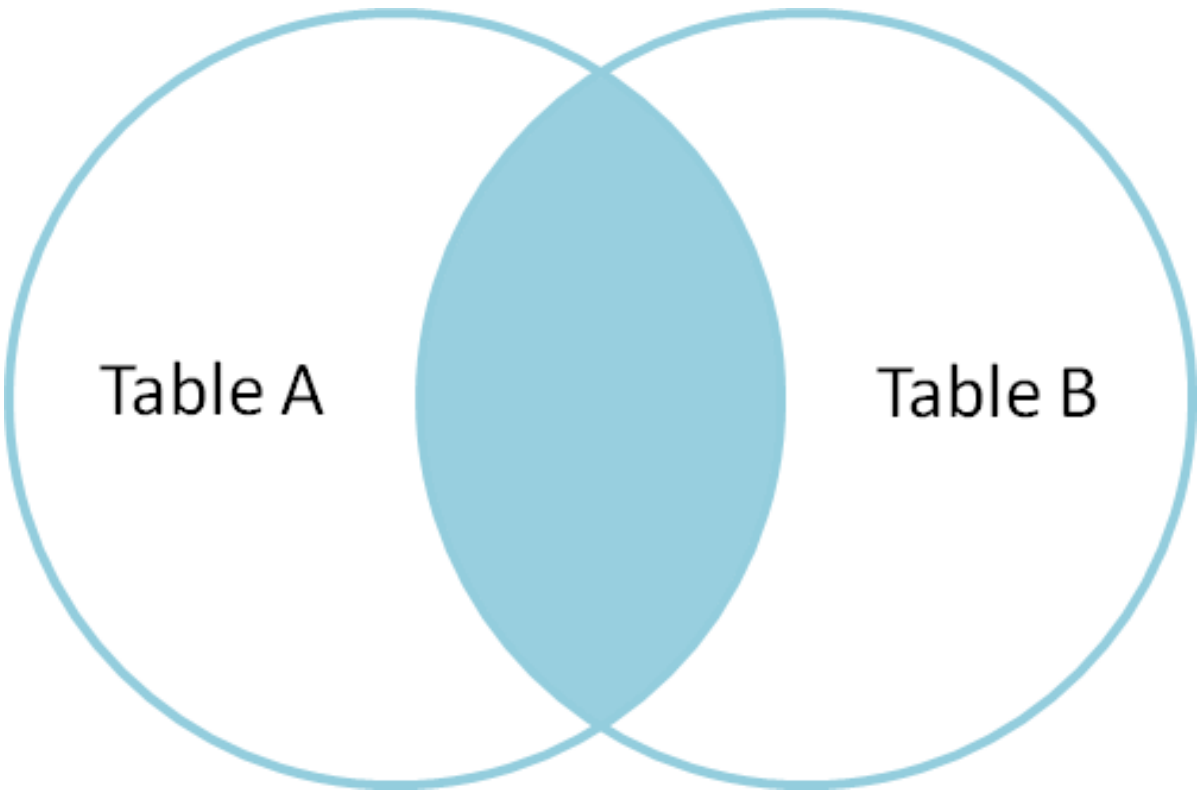
```
create table A (id SERIAL PRIMARY KEY, name text);
insert into A (name)
       values ('Pirate'),
              ('Monkey'),
              ('Ninja'),
              ('Flying Spaghetti Monster');

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:

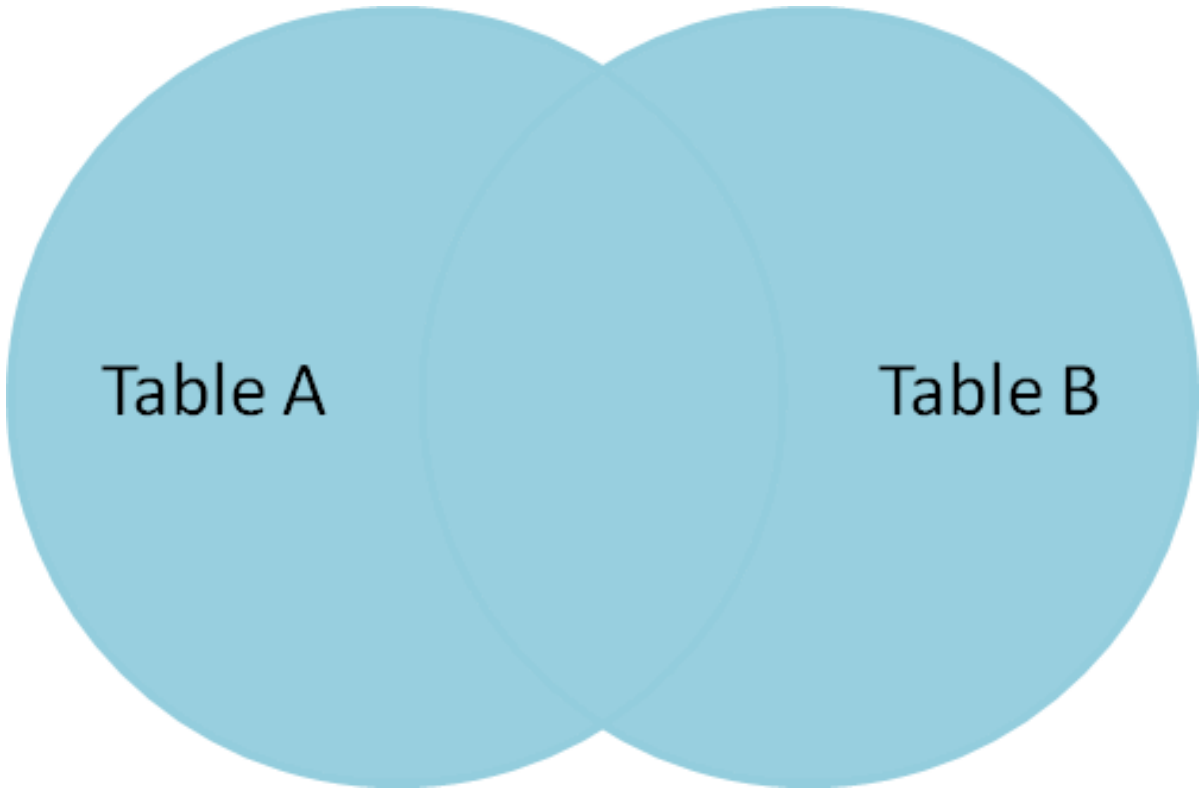


Full Outer Join

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*

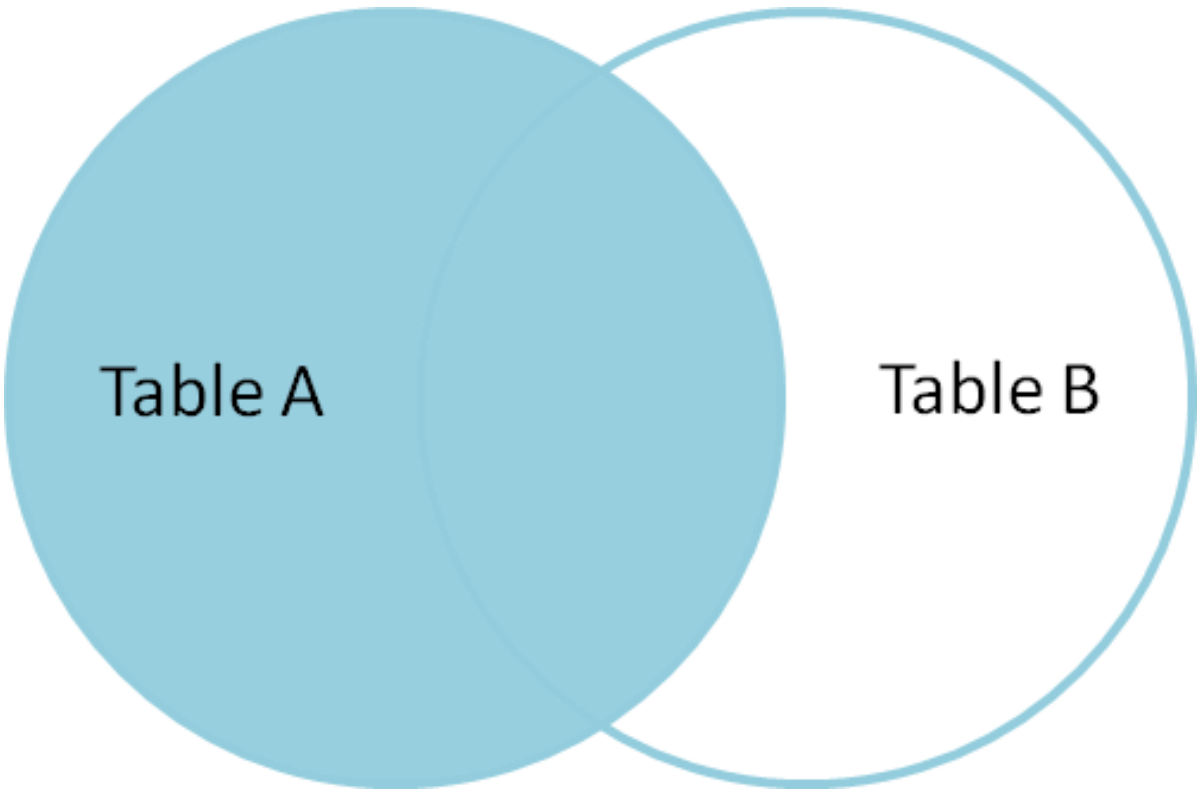


Left Outer Join

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:

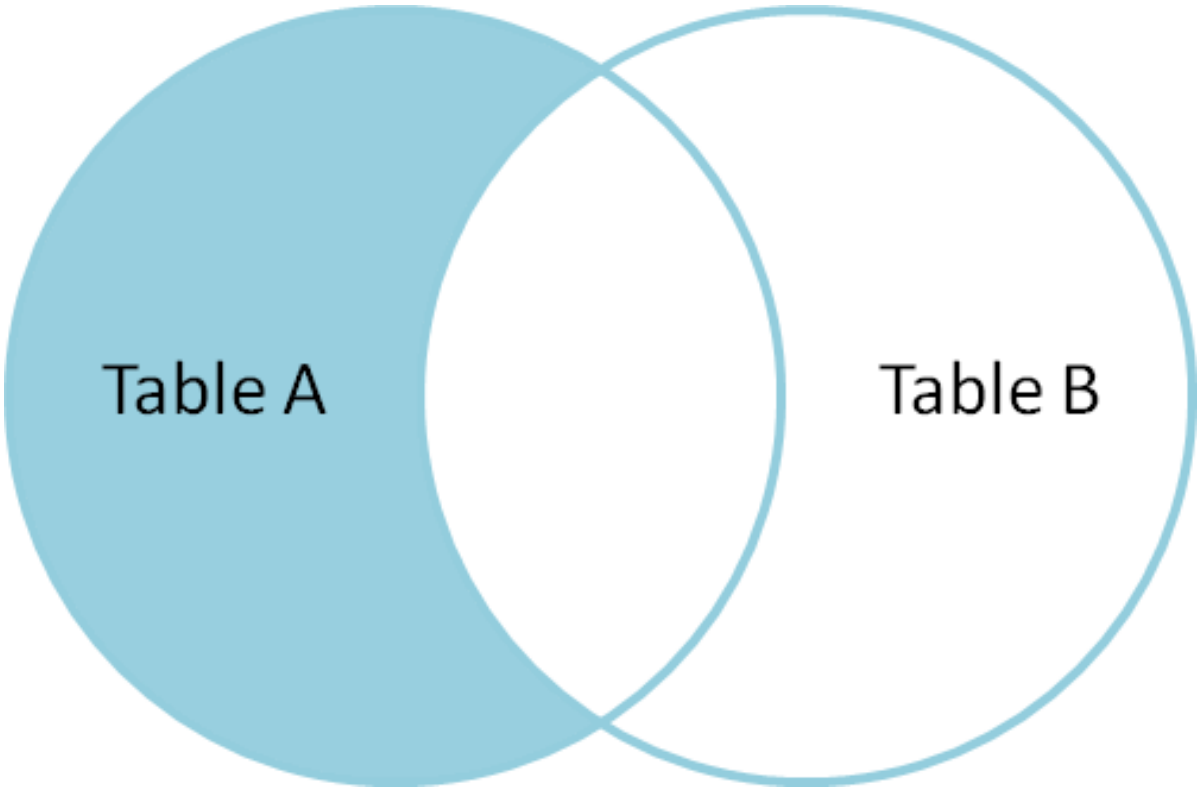


Set Difference

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:

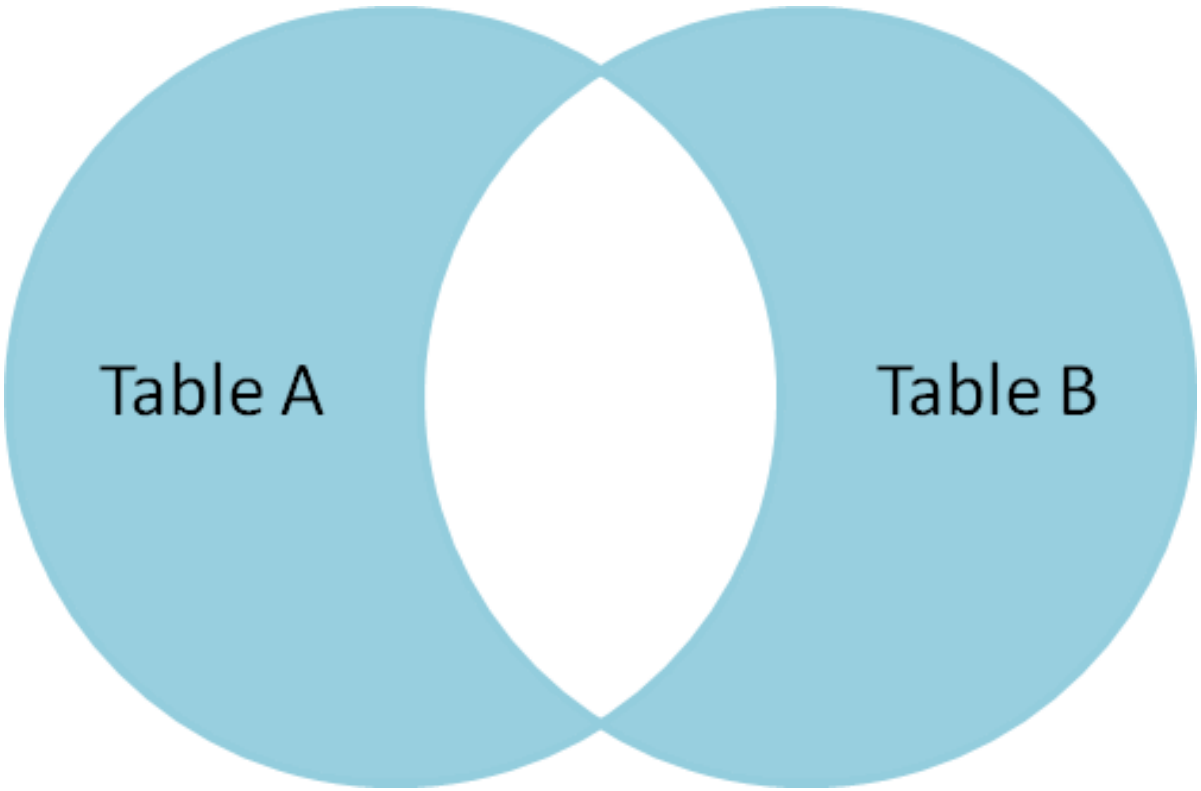


Symmetric Difference

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

Intro

Database Concepts

ACID

SQL

Joins and Foreign Keys

A Deeper Dive

Some Practice

Switching to psql...

The End