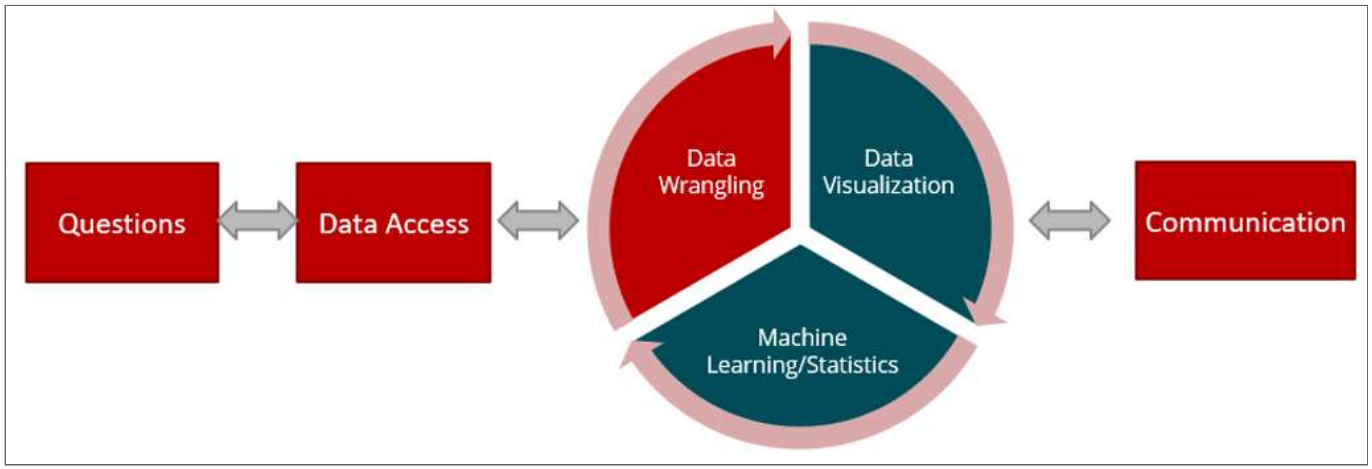
Working With SQL

Overview

- The Big Picture - Data Engineering Pipeline
- Relational Databases
- Data Types
- CRUD
- Creating a Table
- Table Constraints
- Inserting Values
- Updating Values
- Deleting rows from a Table and Dropping Tables
- Querying Data from a Database Table

Do you remember the Data Science Process that was introduced last lecture?

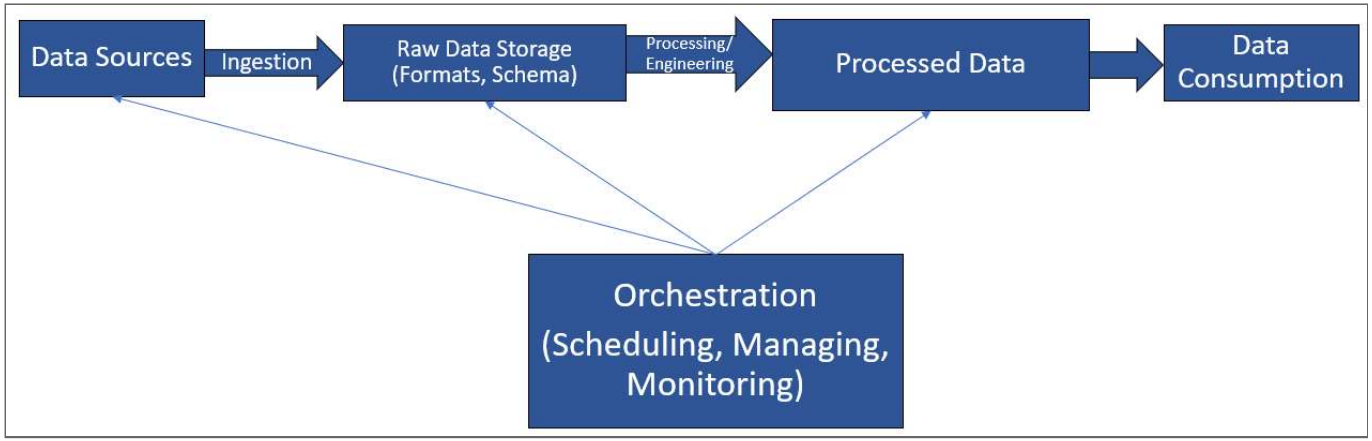
Machine learning has much more to it than just the model.



The Data Science Process

1. The Big Picture - Data Engineering Pipeline

The big picture in machine learning modeling lies in the **Data Engineering pipeline concept**. A data pipeline consists of a series of connected processes that move the data from one point to another, possibly transforming the data along the way.



The General Processes in Data Engineering Pipeline

Data sources can be found publicly in data archives, e.g. Kaggle or GitHub. You may also create your data!

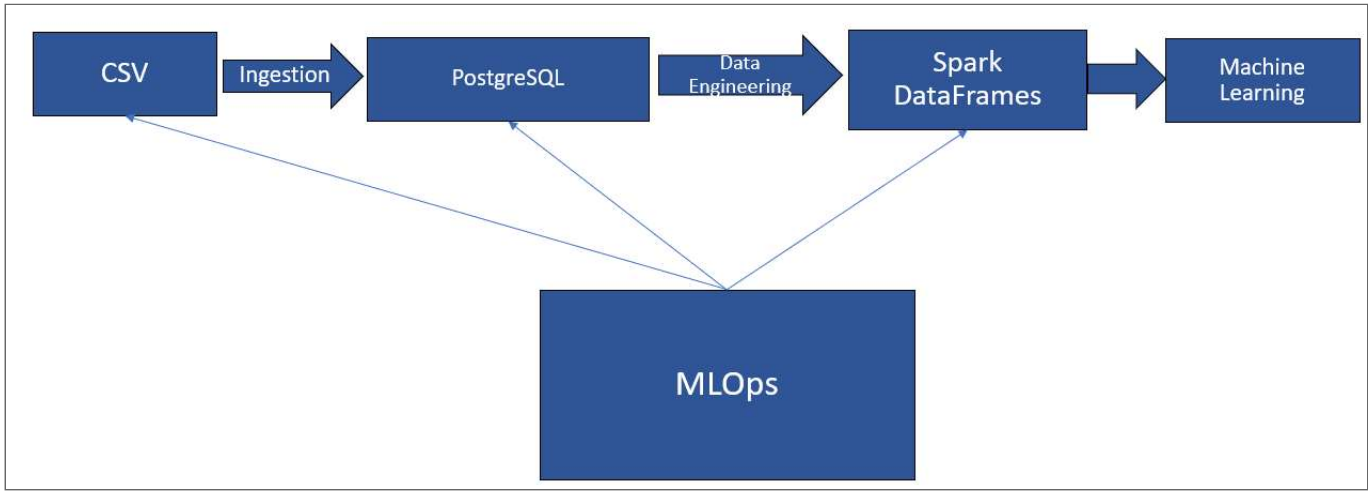
Data ingestion is the process of moving data from disparate operational systems to a central location such as a data warehouse, a data lake, or database, to be processed and made conducive for data analytics. In ingestion, you may ingest data from:

- Relational data sources,
- File-based data sources, or
- Message queues

Ingested data can be stored in a better format by being stored into:

- data warehouses,
- data lakes,
- NoSQL, or
- Relational databases

In our course, we will use the following pipeline:



We will discuss each step in detail throughout the course

Relational Databases - One Option to Store Your Ingested Data

- A relational database (or RDB) is a structure for organizing information that one may think of initially as a supercharged version of a data frame. (The word “relational” does not actually indicate that relationships can exist between separate units in your database. It actually has to do with the concept of relational algebra, first proposed in this paper by E. F. Codd.)

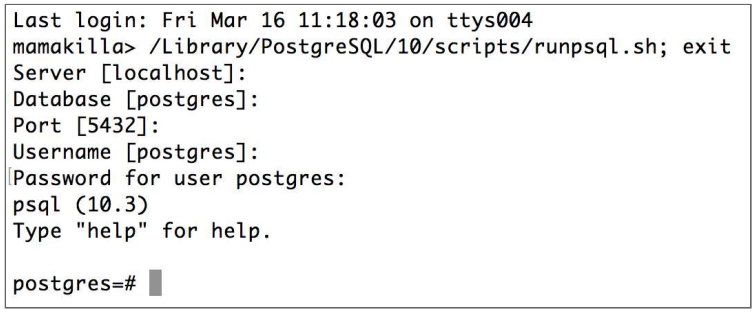
- In an RDB, the basic unit of data storage is the **table (or relation).** Rows are rows, but sometimes dubbed tuples or records, and columns are sometimes dubbed attributes. (Fields are specific cells within a given row.)

- When designing an RDB, a key goal is to eliminate redundancy by having many tables that are linked via foreign keys. For instance, if you want to construct an RDB having information about movies, actors, and studios (where, e.g., a movie is produced by a studio and stars an actor), it makes the most sense to split the information cleanly between interlinked tables.
- RDBs are commonly manipulated and queried using **SQL: Structured Query Language.**

Postgres

- There are many implementations of SQL (SEE-kwule): Oracle, MySQL, SQLite, etc.
- In this class, we will make use of PostgreSQL, or more simply Postgres. It is open source, with Mac, Windows, and Linux versions available. You should download and install it!
- Postgres will set up a server on your machine. When you install Postgres, a superuser account “postgres” will be set up, with a password specified by you. Make this a good one, i.e., not just “postgres.” (That’s because hackers like to look for lazily set up postgres accounts.)

- In this class, it's preferrable that you run SQL code within the pgAdmin4 program that may have come packaged with your version of postgres, but you may also use the SQL shell by running the SQL Shell, or psql...or by typing psql -U postgres. Note that the interface may differ for Windows;



- You should hit return to accept the defaults at each prompt, and provide the password at the end to get to the postgres prompt (postgres=#).

Tables

Tables are the data frames of SQL. But whereas Spark has much more than just data frames, life in SQL revolves around tables; as I am wont to say, "Everything in SQL is a table." As an example, you cannot print out the value of π unless it's in a table:

```
select pi();
        pi
------------------
 3.14159265358979
(1 row)
```

(Here, the value of π is being displayed in a one-row, one-column table, with column name pi. Don't worry about the select part for now; we'll cover that starting next week. Do worry about the semicolon: any command in postgres that doesn't start with a backslash needs to end with a semicolon. Leaving the semicolon out is possibly the Number 1 Coding Error of new SQL users.)

Tables

We note the following about tables:

- each table in a database represents a single entity;
- each row of a table represents one instance of that entity;
- each column of a table must have a name;
- each column of a table must have a pre-specified data type; and
- the columns in each row must satisfy any constraints specified with the table is created.

Data Types

There are many defined data types in SQL. For documentation on those defined within postgres, see the postgres data type documentation page.
For our purposes, there are two categories of data types that you will utilize the most:

- characters;
- numbers; and

We will look at each category over the next few slides.

Note that in addition to these there are date/time and boolean data types. Boolean can take on the values true (or yes or on or 1), false (or no or off or 0), and unknown.

Data Types: Characters

There are three data types used for handling strings:

- char(n): fixed-length strings with n characters, padded with blank spaces if necessary;
- varchar(n): variable-length strings that can have at most n characters, with no blank-space padding; and
- text: variable-length strings of arbitrary length, with no blank-space padding.

One would generally use char(n) only when the length of the strings is known and constant (as would be the case for, e.g., state postal code abbreviations like PA). Otherwise, varchar(n) is preferred over text just because it effectively puts a cap on the amount of memory taken up by a table.

Data Types: Integers

SQL provides a wide variety of data types for representing integers and both fixed-point and floating-point numbers.
For integers:

Data Type	Storage Size	Range
smallint	2 bytes	-2^{15} to $2^{15}-1$
integer	4 bytes	-2^{31} to $2^{31}-1$
bigint	8 bytes	-2^{63} to $2^{63}-1$

While most programmers will default to integer, for most purposes smallint is appropriate and will reduce memory use.

There is also a special category of “auto-incrementing” integers with types smallserial, serial, and bigserial. These are purely positive variations on the integer types listed above, and are used to, e.g., provide unique labels for each row of a table (i.e., “keys”). We will see how serial is used later when we insert data into tables.

Data Types: Fixed- and Floating-Point Numbers

The difference between fixed- and floating-point numbers is in how the computer stores them in memory, and is a topic beyond the scope of this class. Simply note that arithmetic with floating-point numbers can yield inexact results, but in the vast majority of cases the slight inexactness has no bearing on data analysis and interpretation.

For floating-point numbers, we have

- real: the number is stored in 4 bytes and is precise to 6 decimal places; and
- double precision: the number is stored in 8 bytes and is precise to 15 decimal places.

For fixed-point numbers, we have

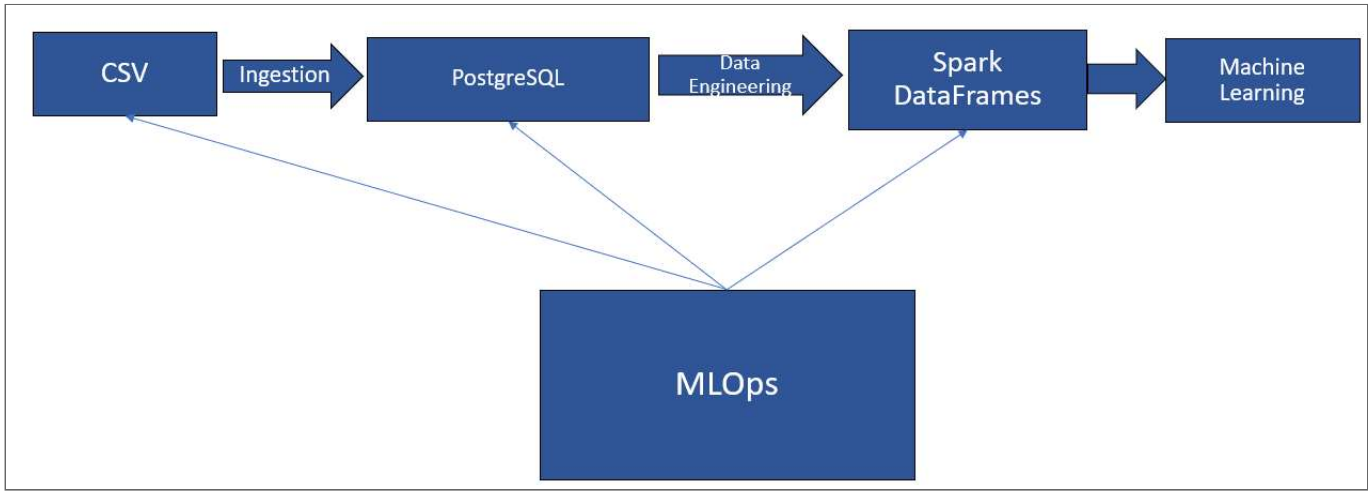
- numeric(precision,scale): the number contains at most precision digits overall, with scale digits to the right of the decimal point. For instance, 147.89 is an example of a number that can be stored as data type numeric(5,2). (Note that decimal is an alternative name for numeric.) The numeric data type is useful when we know that a particular number will not have more than a particular number of digits (e.g., rainfall in a 24-hour period requires at most two digits to the left of the decimal point) and when we want to round off to, say, the hundredths place.

Note that there is a money type within SQL which is akin to numeric but fixes the number of digits to two to the right of the decimal point (and allows for symbols like $ to be included with the numbers).

Data Types: Dates and Times

It's expected that you should have a basic understanding of Date and Times data type. Please read about it from this URL: **https://www.postgresql.org/docs/current/datatype-datetime.html**

You need SQL to execute commands on PostgreSQL. The Main SQL operations can be summarized in the term *CRUD*



But I Don't Like CRUD

No, no. CRUD is an acronym that stands for four basic table operations:

- Create (meaning for us: create and insert)
- Read (meaning select)
- Update (meaning update table and alter table)
- Delete (meaning delete from and drop table)

Creating a Table

Assuming you will need to create and populate a table to host your data.
The command is

```
CREATE TABLE < < name > >
(<< column 1 >>> << type 1 >> << constraint 1 >>, ... , << multi-column constraint(s) >>);
```

Here is a simple example:

```
CREATE TABLE products(
  product_id SERIAL,
  label TEXT,
  price MONEY,
  discount MONEY,
  inventory INTEGER
);
```

This creates a five-column table that contains a label for each product, its price, any current discount on that price, and the current number of each product available.

Note that the product_id column contains data of type **SERIAL**. This, as was stated in the last set of notes, is a data type that represents an auto-incrementing integer. Thus, when we first insert a row into this table, product_id will take on the value 1, then when we insert the next row it will take on the value 2, etc., etc. This provides a unique identifier for each row. Instead of just

```
product_id SERIAL
```

we could specify this as

```
product_id serial PRIMARY KEY,
```

to make it clear that this column has the unique identifier for each product in the table.

Adding Constraints

When creating a SQL table, we may want to add constraints. For instance, a product's price should not be less than $0, and each product should be labeled in a unique manner. Here, we recreate the table defined on the previous page.

```
CREATE TABLE products_with_constraints (
  product_id SERIAL PRIMARY KEY,
  label TEXT UNIQUE NOT NULL CHECK (char_length(label) > 0),
  price MONEY CHECK (price >= 0::MONEY),
  discount MONEY DEFAULT 0.0 CHECK (discount >= 0::MONEY),
  inventory INTEGER DEFAULT 0 CHECK (inventory >= 0),
  CHECK (price > discount)
);
```

There are many things to note here:

- There is a **UINQUE** specifier that ensures that every inserted value is unique.
- There is a **NOT NULL** specifier that indicates you have to insert a value for this variable when inserting a new row into the table.
- There is a **CHECK** specifier:

```
CHECK (<< relationship >>)
```

- There is a **DEFAULT** specifier that is implemented when you do not insert a value.
- **::** is used for casting values into types (e.g., 0::MONEY casts the integer 0 to be of type MONEY).
- You can specify checks of one column's values versus anothers (e.g., the price should be larger than the discount).

Group Excercise

Look at the NSL-KDD dataset and document 5 different constraints that can be applied if we create a database table out of this dataset.

Inserting Values

There are a few ways to insert data into a SQL table. We'll show how to do it row-by-row here, and then utilize select to create bigger tables next week.

To populate the table one row at a time:

```
INSERT INTO << name >> (<< column i >>, << column j >>,...) VALUES
(<< value i >>, << value j >>,...),
...
(<< value i >>, << value j >>,...);
```

If you leave out a column, then the data there will be missing (and can be added later with **UPDATE TABLE**) or will have a default value. Note that any column with data type SERIAL has default behavior: it will auto-increment).

```
INSERT INTO products_with_constraints (label,price,discount,inventory) VALUES
  ('kirk action figure',50,10,13),
  ('spock action figure',40,5,22);
</div> The product_id, being of type SERIAL, will take on the values 1, then 2.
```

If you leave out the (< column i >, < column j>,...) part, then postgres will simply assume that you will be entering values for all columns (in order...so the first value will go into the first column, etc.). For instance:

```
INSERT INTO products_with_constraints VALUES
  (3,'uhura action figure',150,30,3),
  (4,'khan action figure',80,5,12);
```

To look at all columns in your table, do

```
select * from products_with_constraints;
```

```
 product_id |        label         |  price  | discount | inventory
------------+----------------------+---------+----------+-----------
          1 | kirk action figure   |  $50.00 |   $10.00 |        13
          2 | spock action figure  |  $40.00 |    $5.00 |        22
          3 | uhura action figure  | $150.00 |   $30.00 |         3
          4 | khan action figure   |  $80.00 |    $5.00 |        12
(4 rows)
```

Updating Table Values

The **UPDATE** command allows us to modify values in table cells.

```
UPDATE < name >
  SET <column1> = < new value 1 > ,
  SET <column2> = < new value 2 >,
  ...
  WHERE <row condition >;
</div>
```

Think of the **WHERE < row condition >** as being like a call to the **which()** function in R: in it, you set a range of values for one of the table columns, and thereby select which rows to update.

```
UPDATE products_with_constraints
  set discount = 10::money
  where price >= 70::money;
```

(Note that when you look at an updated table, the serial data may not be displayed in numeric order, i.e., the rows may be rearranged. This is OK.)

Deleting Rows and Removing a Table Entirely

To remove one or more entries from a table:

```
DELETE FROM << name >>
 WHERE << condition >>;
```

ⓘ (Be careful! If you leave out the where clause, then all entries are deleted.)

To remove a table in its entirety:

```
DROP TABLE <<name>>;
```

To check that the table is removed, look for it in the Table list on PgAdmin4.

Select: Querying a Database

The **SELECT** command is how we query a database. It is a versatile and powerful command!
A shortened definition that highlights elements of the syntax that are important in the context of this class is:

```
SELECT
  [*]
  [<expression>]
  [FROM <expression>]
  [WHERE <condition>]
  [GROUP BY <expression>]
  [HAVING <condition>]
  [ORDER BY <expression>];
```

These **SELECT** options are useful to understand and they match what Spark and other big data processing frameworks offer to help you select, filter and group data from their tables. In this lecture, we will explore the **WHERE** option. Keep in mind the following considerations:

- You can pass functions (built-ins or customly created) into SELECT. (See below.)
- **SELECT** constructs virtual tables, and its displayed output is a table. (Everything is a table in postgres and in SQL in general.)

```
select pi();
        pi
------------------
 3.14159265358979
(1 row)
```

Where: Filtering Rows in a Table

We can select the rows to view or act on by giving logical conditions in a WHERE clause:

- SELECT product_id, label, price FROM

  products_with_constraints

  WHERE price >= 45::MONEY;

This is just an introduction to SQL. There are several things you can do in SQL that are not mentioned here. For example, you can find the maximum/minimum value in a specific column or group values in a specific column. We will discuss these details when we talk about SparkSQL