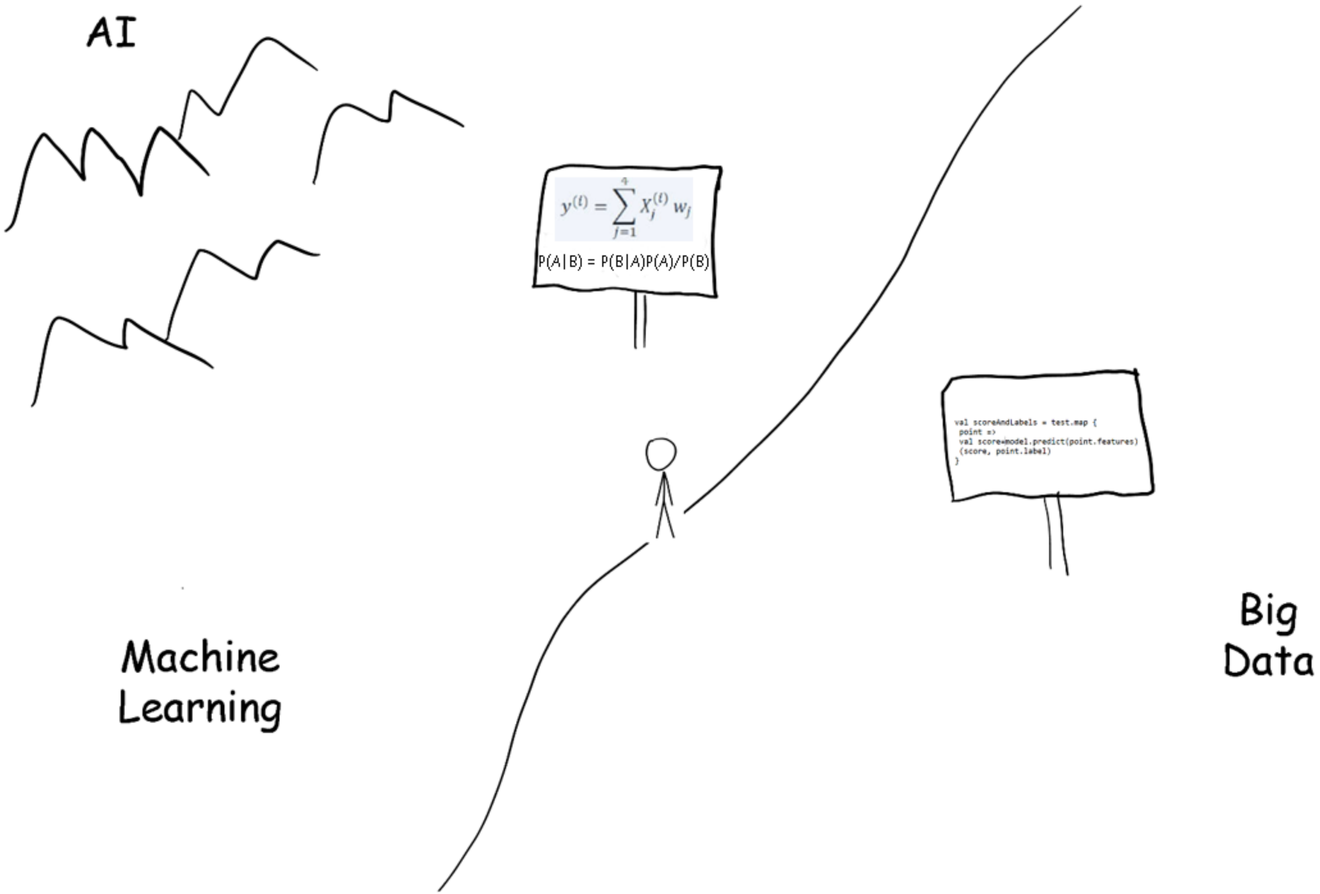
ADAPT Module

Intro to Data Science with Pandas and SQL

John Urbanic
Pittsburgh Supercomputing Center

The landscape today.



As the Data Scientist wanders across the ill-defined boundary between Data Science and Machine Learning, in search of the fabled land of Artificial Intelligence, they find that the language changes from programming to a creole of linear algebra and probablity and statistics.

Data Science Today

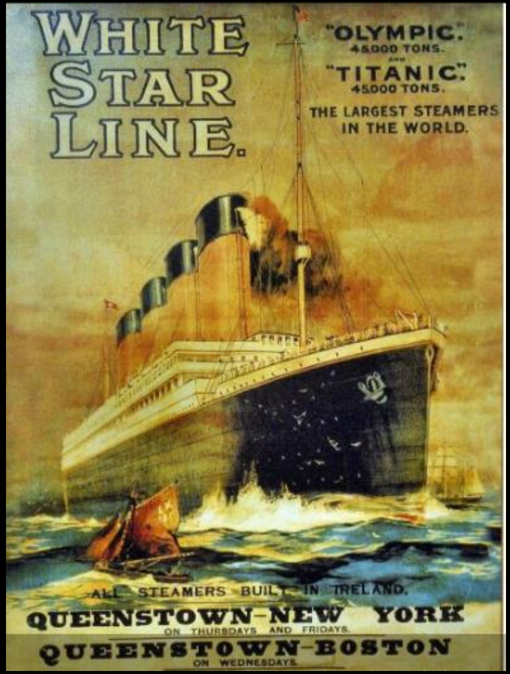
- Basic Data
  - Pandas
- "Serious" Data Science
  - SQL
- Big Data
  - Spark

Machine Learning / AI

Pandas

- Pandas has become the standard Python way to input, manipulate and write basic data.
- It also integrates well with other tools, like visualizing with Matplotlib.
- It has limitations, which is why SQL and big data techniques are essential for many tasks, but for quick-and-dirty, or limited applications it is very efficient.
- In many Python environments, it is there by default. If not, it is easy to add. In this course, if you start a python shell, it will be there.

Our First Dataset



We will begin our exploration of Pandas using a well known dataset drawn from the infamous Titanic disaster.

It has a variety of data on each of 891 passengers.

Amongst the typical demographic data is included their survival. It enables an interesting, if somewhat morbid, analysis to determine the foremost factors in survival. Women and children first? Or, save the rich?

Getting Started with Pandas

```
import pandas as pd

titanic = pd.read_csv("titanic.csv")

titanic
```

This "pd" is *very* standard

Smart, understands "csv"

```
     PassengerId  Survived  Pclass   ...     Fare Cabin  Embarked
0              1         0       3   ...    .2500   NaN         S
4              5         0       3   ...    .0500   NaN         S
5              6         0       3   ...    .4583   NaN         Q
6              7         0       1   ...    .8625   E46         S
7              8         0       3   ...    .0750   NaN         S
..           ...       ...     ...   ...      ...   ...       ...
883          884         0       2   ...    .5000   NaN         S
884          885         0       3   ...    .0500   NaN         S
886          887         0       2   ...    .0000   NaN         S
889          890         1       1   ...    .0000  C148         C
890          891         0       3   ...    .7500   NaN         Q

[577 rows x 12 columns]
```

Format Type	Data Description	Reader	Writer
text	CSV	*read_csv*	*to_csv*
text	Fixed-Width Text File	*read_fwf*	
text	JSON	*read_json*	*to_json*
text	HTML	*read_html*	*to_html*
text	LaTeX		*Styler.to_latex*
text	XML	*read_xml*	*to_xml*
text	Local clipboard	*read_clipboard*	*to_clipboard*
binary	MS Excel	*read_excel*	*to_excel*
binary	OpenDocument	*read_excel*	
binary	HDF5 Format	*read_hdf*	*to_hdf*
binary	Feather Format	*read_feather*	*to_feather*
binary	Parquet Format	*read_parquet*	*to_parquet*
binary	ORC Format	*read_orc*	
binary	Stata	*read_stata*	*to_stata*
binary	SAS	*read_sas*	
binary	SPSS	*read_spss*	
binary	Python Pickle Format	*read_pickle*	*to_pickle*
SQL	SQL	*read_sql*	*to_sql*
SQL	Google BigQuery	*read_gbq*	*to_gbq*

Survived	Survival (0 = No; 1 = Yes)
Pclass	Passenger Class (1 = 1st; 2 = 2nd; 3 = 3rd)
Name	Name
Sex	
Age	
SibSp	Number of Siblings/Spouses Aboard
Parch	Number of Parents/Children Aboard
Ticket	Ticket Number
Fare	Fare (British pound)
Cabin	Cabin number
Embarked	Port of Embarkation (C = Cherbourg; Q = Queenstown; S = Southampton)

DataFrame Queries

```
titanic["Name"]
```

```
0                              Braund, Mr. Owen Harris
1          Cumings, Mrs. John Bradley (Florence Briggs Th...
2                               Heikkinen, Miss. Laina
3               Futrelle, Mrs. Jacques Heath (Lily May Peel)
4                             Allen, Mr. William Henry
                        ...
886                              Montvila, Rev. Juozas
887                       Graham, Miss. Margaret Edith
888           Johnston, Miss. Catherine Helen "Carrie"
889                              Behr, Mr. Karl Howell
890                                Dooley, Mr. Patrick
```

DataFrame Queries

```
titanic[["Age","Sex"]]

      Age     Sex
0    22.0    male
1    38.0  female
2    26.0  female
3    35.0  female
4    35.0    male
..    ...     ...
886  27.0    male
887  19.0  female
888   NaN  female
889  26.0    male
890  32.0    male
```

DataFrame Conditional Queries

```
titanic[titanic["Age"]>30]
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S
6	7	0	1	McCarthy, Mr. Timothy J	male	54.0	0	0	17463	51.8625	E46	S
11	12	1	1	Bonnell, Miss. Elizabeth	female	58.0	0	0	113783	26.5500	C103	S
..	...	...	...	...	...	...	...	...	...	...	...	...
873	874	0	3	Vander Cruyssen, Mr. Victor	male	47.0	0	0	345765	9.0000	NaN	S
879	880	1	1	Potter, Mrs. Thomas Jr (Lily Alexenia Wilson)	female	56.0	0	1	11767	83.1583	C50	C
881	882	0	3	Markun, Mr. Johann	male	33.0	0	0	349257	7.8958	NaN	S
885	886	0	3	Rice, Mrs. William (Margaret Norton)	female	39.0	0	5	382652	29.1250	NaN	Q
890	891	0	3	Dooley, Mr. Patrick	male	32.0	0	0	370376	7.7500	NaN	Q

DataFrame Sorting

```
titanic.sort_values(by="Age")[["Name","Age"]]
```

```
                                        Name   Age
803          Thomas, Master. Assad Alexander  0.42
755                Hamalainen, Master. Viljo  0.67
644                   Baclini, Miss. Eugenie  0.75
469            Baclini, Miss. Helene Barbara  0.75
78             Caldwell, Master. Alden Gates  0.83
..                                       ...   ...
859                         Razi, Mr. Raihed   NaN
863        Sage, Miss. Dorothy Edith "Dolly"   NaN
868              van Melkebeke, Mr. Philemon   NaN
878                       Laleff, Mr. Kristo   NaN
888 Johnston, Miss. Catherine Helen "Carrie"   NaN
```

```
titanic.sort_values(by="Age")[["Name","Age"]][0:10]
```

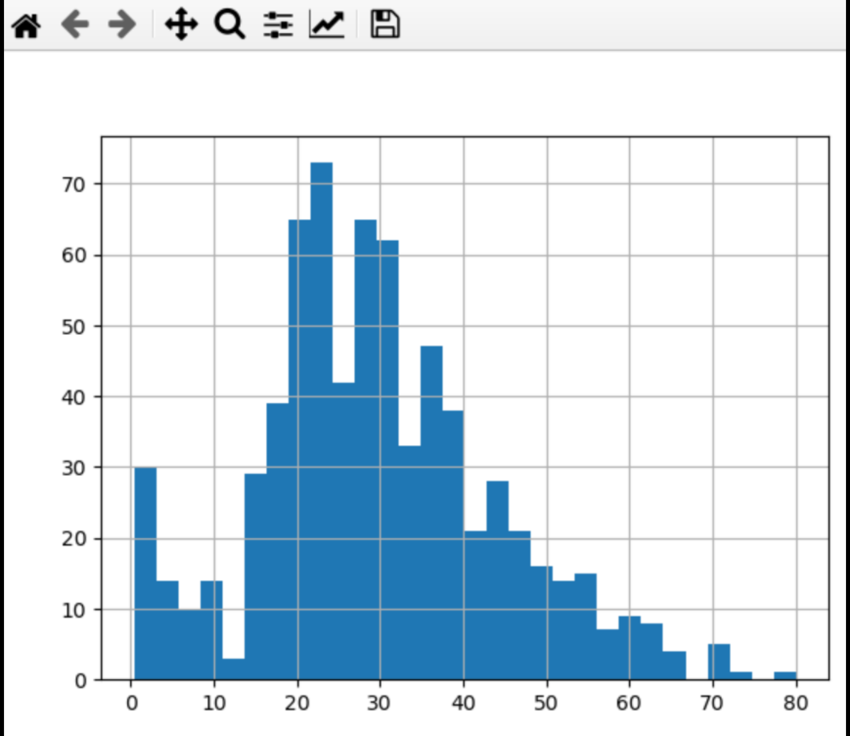
```
                                        Name   Age
803          Thomas, Master. Assad Alexander  0.42
755                Hamalainen, Master. Viljo  0.67
644                   Baclini, Miss. Eugenie  0.75
469            Baclini, Miss. Helene Barbara  0.75
78             Caldwell, Master. Alden Gates  0.83
..                                       ...   ...
859                         Razi, Mr. Raihed   NaN
863        Sage, Miss. Dorothy Edith "Dolly"   NaN
868              van Melkebeke, Mr. Philemon   NaN
878                       Laleff, Mr. Kristo   NaN
888 Johnston, Miss. Catherine Helen "Carrie"   NaN
```

If you like pictures (matplotlib)

```
import matplotlib.pyplot as plt

titanic["Age"].hist(bins=30)

plt.show()
```



This assumes you have an
X server running on your laptop.

Which we do.

Assignment: Can we find a significant survival variable?

Can you find a significant factor in the data which could be used to predict survival rates?

I will suggest you focus on one variable at a time.

Note that there are many possible answers. Going from a simple hypothesis ("Maybe people from Cherbourg are unlucky?") to a more complex formula incorporating multiple variables - with the goal of a more accurate prediction - is the path of data analysis. This is our first step on that journey.

- We are going to use a Virtual Machine for this Assignment. It is called adapt.psc.edu and you can *ssh* there.
- Copy the titanic dataset (using the cp command) from ~datasets/Titanic/titanic.csv to your own directory.
- Start a python shell.
- Find a meaningful factor and submit your script and results.

PITTSBURGH SUPERCOMPUTING CENTER

Solution Review:

Titanic with Pandas

Getting Started with Titanic

```
import pandas as pd
titanic = pd.read_csv("titanic.csv")
```

```
males = titanic[titanic["Sex"]=="male"]
```

```
     PassengerId  Survived  Pclass                           Name   Sex   Age  SibSp  Parch            Ticket     Fare Cabin Embarked
0              1         0       3        Braund, Mr. Owen Harris  male  22.0      1      0         A/5 21171   7.2500   NaN        S
4              5         0       3       Allen, Mr. William Henry  male  35.0      0      0            373450   8.0500   NaN        S
5              6         0       3               Moran, Mr. James  male   NaN      0      0            330877   8.4583   NaN        Q
6              7         0       1        McCarthy, Mr. Timothy J  male  54.0      0      0             17463  51.8625   E46        S
7              8         0       3 Palsson, Master. Gosta Leonard  male   2.0      3      1            349909  21.0750   NaN        S
..           ...       ...     ...                            ...   ...   ...    ...    ...               ...      ...   ...      ...
883          884         0       2  Banfield, Mr. Frederick James  male  28.0      0      0  C.A./SOTON 34068  10.5000   NaN        S
884          885         0       3         Sutehall, Mr. Henry Jr  male  25.0      0      0   SOTON/OQ 392076   7.0500   NaN        S
886          887         0       2          Montvila, Rev. Juozas  male  27.0      0      0            211536  13.0000   NaN        S
889          890         1       1          Behr, Mr. Karl Howell  male  26.0      0      0            111369  30.0000  C148        C
890          891         0       3            Dooley, Mr. Patrick  male  32.0      0      0            370376   7.7500   NaN        Q

[577 rows x 12 columns]
```

```
males.shape
```

```
(577, 12)
```

```
males[males["Survived"]==1].shape
```

```
(109, 12)
```

```
109/577
```

```
0.18890814558058924
```

19% Survival Rate for Males

How did the women fare?

```
titanic[titanic["Sex"]=="female"].shape
```

(314, 12)

```
titanic[ (titanic["Sex"]=="female") & (titanic["Survived"]==1) ].shape
```

(233, 12)

```
233/314
```

0.7420382165605095

74% Survival Rate for Females

Hypothesis confirmed: chivalry not dead.

But Jack Dawson is.

Women and children first!?

```
men = titanic[ (titanic["Sex"]=="male") & (titanic["Age"]>15) ]

men.shape

(413, 12)


men[ men["Survived"]==1 ].shape

(72, 12)


72/413

0.17433414043583534233/314
```

17% Survival Rate for Men

NaNs are everywhere!

```
women_and_children = titanic[ (titanic["Sex"]=="female") | (titanic["Age"]<16) ]
women_and_children.shape

(354, 12)

#Seems like some people are missing...



titanic[titanic["Age"].isna()]

     PassengerId  Survived  Pclass                                     Name     Sex  Age  SibSp  Parch      Ticket     Fare Cabin Embarked
5              6         0       3                         Moran, Mr. James    male  NaN      0      0      330877   8.4583   NaN        Q
17            18         1       2             Williams, Mr. Charles Eugene    male  NaN      0      0      244373  13.0000   NaN        S
19            20         1       3                  Masselmani, Mrs. Fatima  female  NaN      0      0        2649   7.2250   NaN        C
26            27         0       3                  Emir, Mr. Farred Chehab    male  NaN      0      0        2631   7.2250   NaN        C
28            29         1       3            O'Dwyer, Miss. Ellen "Nellie"  female  NaN      0      0      330959   7.8792   NaN        Q
..           ...       ...     ...                                      ...     ...  ...    ...    ...         ...      ...   ...      ...
859          860         0       3                         Razi, Mr. Raihed    male  NaN      0      0        2629   7.2292   NaN        C
863          864         0       3        Sage, Miss. Dorothy Edith "Dolly"  female  NaN      8      2    CA. 2343  69.5500   NaN        S
868          869         0       3              van Melkebeke, Mr. Philemon    male  NaN      0      0      345777   9.5000   NaN        S
878          879         0       3                       Laleff, Mr. Kristo    male  NaN      0      0      349217   7.8958   NaN        S
888          889         0       3  Johnston, Miss. Catherine Helen "Carrie"  female  NaN      1      2  W./C. 6607  23.4500   NaN        S

[177 rows x 12 columns]


413+354+177

944
```

This is bigger than the total passenger list (891). But makes sense as we have double counted some females with Age=NaN in our logic.

Women and children first!

```
women_and_children[ women_and_children["Survived"]==1 ].shape
```

```
(254, 12)
```

```
254/354
```

```
0.7175141242937854
```

72% Survival Rate for Women & Children

How did Thurston Howell III make out?

Another obvious question we might ask is how did the wealthier, 1st class, passengers do versus the underclasses?

We could continue with our basic tools and separate out the various passenger classes, and perform some math to get at an answer.

However, we are now starting to ask questions that can utilize more sophisticated tools like:

- Joins (called Merges in Pandas)
- Grouping
- Pivot tables

Pandas has these capabilities. However, more complex data manipulation like this can often benefit from the more powerful capabilities of a Structured Query Language (SQL) database. Certainly at scale.

So we will preview the power of these operations with one last look at this problem, and then we will move on to SQL.

After you have learned SQL, you will easily be able to employ these operations in Pandas when you wish.

Grouping

Grouping typically performs 3 steps:

- Splits the data into groups base on some criteria: Pclass
- Applies a function to each group separately: Survival Rate
- Combines the results into a new table

That is one way to get directly at our answer. It becomes this simple:

```
titanic[['Pclass', 'Survived']].groupby('Pclass').mean()

Pclass  Survived
1       0.629630
2       0.472826
3       0.242363
```

That is a pretty brutal curve.
I believe it speaks for itself.

SQL

What is a "Relational Database"?

An RDBMS (Relational DataBase Management System) organizes data into tables of columns (attributes, fields) and rows (records).

This concept has been developed and refined since 1970, and is a mature concept at this point.

Most RDMBSs use SQL as their query language. This has become an ISO standard (with many deviations).

What Is MySQL?

MySQL is an open source RDBMS originating in 1995. It has spun off forks, and it has open source peers (most notably PostgreSQL) and commercial alternatives (Oracle and MS SQL Server). These each have their own deviations from the ISO standard, as well as significant performance differences.

MySQL operates as a server, with clients that connect from wherever, and may be calling from many different languages: from JavaScript in some web page, or Java on the backend, or within a Python program. We will be using a dedicated, if basic, MySQL client.

Starting MySQL

We will use a MySQL client installed on the VM along with our database. To start it you need only log on to adapt.psc.edu and type:

```
[urbanic@msdas]$ mysql urbanic
mysql> SHOW DATABASES;

+--------------------+
| Database           |
+--------------------+
| performance_schema |
| urbanic            |
+--------------------+
2 rows in set (0.01 sec)
```

This shows us the available databases. By starting mysql with the command "mysql urbanic" I have loaded my own personal database already. Make sure to use your own username to start mysql, not "urbanic". We could also use the command USE urbanic to select this database at any time.

Note that all SQL commands end with a ";". Case matters, but SQL keywords can be upper or lower case. I will use upper case for them as that is a common convention and makes it clear what they are as you are learning.

Also note that SQL code formatting *varies wildly* and is inconsistent. This talk will adhere to that tradition. Although no ones seems to care, there is a supposed standard, and the best summary of it I can find is here:

https://www.isbe.net/Documents/SQL_server_standards.pdf

Showing Our Tables

The structure, or schema, is the most important characteristic of any database. We can get a top level view by first listing the tables.

```
mysql> SHOW TABLES;
```

```
+-----------------+
|Tables_in_urbanic|
+-----------------+
| Customer        |
| Line            |
| OrderDetail     |
| Orders          |
| Product         |
| Vendor          |
| Zips            |
+-----------------+
7 rows in set (0.00 sec)
```

From here on out I will drop the *mysql>* prompt from our examples. We are always working in the SQL client shell.

Showing Table Fields

And each table has fields, or columns. We can list them as so.

```
SHOW COLUMNS FROM Orders;

+--------------+-------------+------+-----+---------+-------+
| Field        | Type        | Null | Key | Default | Extra |
+--------------+-------------+------+-----+---------+-------+
| OrderId      | int         | NO   |     | 0       |       |
| Date         | date        | NO   |     | NULL    |       |
| DateRequired | date        | NO   |     | NULL    |       |
| DateShipped  | date        | YES  |     | NULL    |       |
| Status       | varchar(15) | NO   |     | NULL    |       |
| Comments     | text        | YES  |     | NULL    |       |
| CustomerId   | int         | NO   |     | NULL    |       |
+--------------+-------------+------+-----+---------+-------+
7 rows in set (0.00 sec)
```

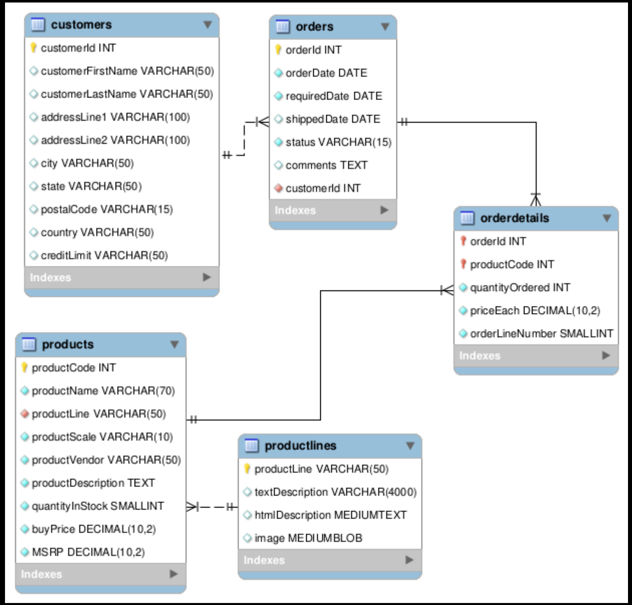
Each field has a type, and some have a size. There are 30+ types, but they are mostly obvious variations of strings, numbers and dates. There are some fancy Spatial, JSON and binary blob types as well. You can find a full list at:

https://dev.mysql.com/doc/refman/8.0/en/data-types.html

There are some other features attached to fields that we will get to later.

MySQL Workbench

MySQL Workbench is a wonderful tool for working with MySQL databases. If we were going to work deeply with SQL, we should surely involve this more in our work. However, in keeping with our theme of minimal distractions while we investigate the core concepts, we will only use it to create nice schematics.



These are most of our current tables. We will see what some of those little icons and arrows mean later.

SELECT

The SELECT command is our most useful command in manipulating data, and we will look at some of the common variations.

```
SELECT * FROM Customer;
+------------+-----------+----------+-------------------------------+--------------+-----------------+-------+------------+---------+-------------+
| CustomerId | FirstName | LastName | AddressLine1                  | AddressLine2 | City            | State | PostalCode | Country | CreditLimit |
+------------+-----------+----------+-------------------------------+--------------+-----------------+-------+------------+---------+-------------+
|          1 | Mary      | Yates    | 1414 East Anderson Street     | #317         | Savannah        | GA    | 31404      | USA     | 100         |
|          2 | James     | Parker   | 29 Lucian Street              |              | Manchester      | CT    | 06040      | USA     | 100         |
|          3 | Kim       | Bond     | 1421 Floral Street Northwest  |              | Washington      | DC    | 20012      | USA     | 100         |
|          4 | Thomas    | Broadnax | 1915 Southeast 29th Street    |              | Oklahoma City   | OK    | 73129      | USA     | 100         |
|          5 | Stephen   | Williams | 9805 South Youngs Lane        |              | Oklahoma City   | OK    | 73159      | USA     | 100         |
|          6 | Mark      | Hinton   | 8642 Yule Street              |              | Arvada          | CO    | 80007      | USA     | 100         |
|          7 | Deborah   | Lloyd    | 5244 West Port Au Prince Lane |              | Glendale        | AZ    | 85306      | USA     | 100         |
|          8 | Linda     | Barnes   | 3377 Sandstone Court          |              | Pleasanton      | CA    | 94588      | USA     | 100         |
...
Whoops, too many!
```

```
SELECT * FROM Customer LIMIT 5;
+------------+-----------+----------+-------------------------------+--------------+---------------+-------+------------+---------+-------------+
| CustomerId | FirstName | LastName | AddressLine1                  | AddressLine2 | City          | State | PostalCode | Country | CreditLimit |
+------------+-----------+----------+-------------------------------+--------------+---------------+-------+------------+---------+-------------+
|          1 | Mary      | Yates    | 1414 East Anderson Street     | #317         | Savannah      | GA    | 31404      | USA     | 100         |
|          2 | James     | Parker   | 29 Lucian Street              |              | Manchester    | CT    | 06040      | USA     | 100         |
|          3 | Kim       | Bond     | 1421 Floral Street Northwest  |              | Washington    | DC    | 20012      | USA     | 100         |
|          4 | Thomas    | Broadnax | 1915 Southeast 29th Street    |              | Oklahoma City | OK    | 73129      | USA     | 100         |
|          5 | Stephen   | Williams | 9805 South Youngs Lane        |              | Oklahoma City | OK    | 73159      | USA     | 100         |
+------------+-----------+----------+-------------------------------+--------------+---------------+-------+------------+---------+-------------+
5 rows in set (0.00 sec)
```

Note how * is our "wildcard" for all the fields.

SELECTING Fields

We can select only the fields of interest

```
SELECT FirstName, LastName FROM Customer;
+-----------+----------+
| FirstName | LastName |
+-----------+----------+
| Mary      | Yates    |
| James     | Parker   |
| Kim       | Bond     |
| Thomas    | Broadnax |
| Stephen   | Williams |
| Mark      | Hinton   |
...
...
```

And we can sort them

```
SELECT FirstName, LastName FROM Customer ORDER BY LastName;
+-----------+----------+
| FirstName | LastName |
+-----------+----------+
| Michael   | Aaberg   |
| Denver    | Aaberg   |
| Brenda    | Aaberg   |
| Mabel     | Aaberg   |
| Gary      | Aaberg   |
| Ellen     | Aaberg   |
| Frankie   | Aaberg   |
| Edward    | Aaberg   |
...
...
```

SELECTING Rows

We can select specific rows with the WHERE command.

```
SELECT * FROM Customer WHERE CustomerId = 1;
```

CustomerId	FirstName	LastName	AddressLine1	AddressLine2	City	State	PostalCode	Country	CreditLimit
1	Mary	Yates	1414 East Anderson Street	#317	Savannah	GA	31404	USA	100

1 row in set (0.00 sec)

```
SELECT * FROM Orders WHERE CustomerId = 1;
```

OrderId	Date	DateRequired	DateShipped	Status	Comments	CustomerId
390977	2007-08-09	2007-08-29	2007-09-04	OK		1
396900	2007-10-15	2007-10-21	2007-10-29	OK		1
472220	2010-02-21	2010-03-02	2010-03-10	OK		1
486581	2010-08-04	2010-08-10	2010-08-10	OK		1
487083	2010-08-10	2010-08-12	2010-08-16	OK		1
513816	2011-06-14	2011-06-22	2011-06-14	OK		1
546268	2012-06-20	2012-06-23	2012-06-28	OK		1
585992	2013-09-16	2013-09-29	2013-10-12	OK		1

8 rows in set (0.01 sec)

AGGREGATE FUNCTIONS

There are aggregate functions that we can apply to a column of data. For example, we could find the average retail price of all of our products

```
SELECT AVG(RetailPrice) FROM Product;
+-------------------+
| avg(RetailPrice) |
+-------------------+
|        90.173951 |
+-------------------+
1 row in set (0.02 sec)
```

The most common aggregate function is COUNT(), frequently used to count the number of rows in a table. MAX(), MIN(), SUM(), AVG() are others that you will see.

```
SELECT COUNT(*) FROM Product;
+----------+
| count(*) |
+----------+
|    20592 |
+----------+
```

Given that * is used in SELECT statements to select all the columns, a normal person might think that COUNT(*) is asking to somehow count multiple columns, but it is really just allowing SQL to pick whatever column it thinks is quickest to use to count the total number of rows in the table. Get used to this common/weird idiom.

GROUPING

Grouping is a very useful tool in data analysis. And we have a particular meaning for the word "grouping" in data science. It means a rearrangement of a data table such that one of the columns becomes the rows.

After this rearrangement, we usually have to decide which of the other columns we want to keep or combine.

Original Table

A	B	C	D
AO	B0	C0	D0
A1	B1	C1	D1
A2	B0	C2	D1
A3	B3	C3	D3
A4	B0	C4	D0
A5	B2	C5	D1
A6	B3	C6	D3
A7	B1	C7	D0
A8	B0	C8	D1
A9	B3	C9	D0

Grouped on B, Average of C

B	C(avg)
BO	AVG(C0,C2,C4,C8)
B1	AVG(C1,C7)
B2	AVG(C5)
B3	AVG(C3,C6,C9)

An example might be where we are logging pollution alarms, and the table is

Time	Station	Level	Supervisor
2023-4-2-11:23	Hampton	11.3	Smith
2023-5-2-12:33	Landsdale	0.42	Li
...	...	...	...

If we want an quick insight into where any serious problems are, we might want to find the average at each station.

We don't care about the timestamps, and we probably don't want to bring the supervisor data along.

We do exactly what we just did at the left: GROUP BY Station and AVG(Level).

GROUP BY and AS

As we build more complex queries, we will find it very useful, and often necessary, to alias a column or table with *AS*. The alias will only exist for the duration of the query. GROUP BY will often require this.

```
SELECT CustomerId, COUNT(*) AS NumOrders
FROM Orders
GROUP BY CustomerId
ORDER BY NumOrders DESC
LIMIT 10;
```

```
+------------+-----------+
| CustomerId | NumOrders |
+------------+-----------+
|      25806 |        25 |
|      94364 |        25 |
|      96968 |        24 |
|      33646 |        24 |
|      57572 |        23 |
|      89204 |        23 |
|      27518 |        23 |
|      29451 |        23 |
|      26682 |        23 |
|      36709 |        22 |
+------------+-----------+
10 rows in set (1.72 sec)
```

GROUP BY will group rows that have the same value (CustomerID here) into summary rows, which are then used with aggregate functions (*COUNT(), MAX(), MIN(), SUM(), AVG()*) to reduce the results.

These aggregate functions reduce the data on a selected column. Here, COUNT gives us the number in each group.

We need NumOrders to capture that value for subsequent use in the ORDER BY.

Note the grouping can be done hierarchically. You might group your data first by towns, and then zip codes within.

ORDER OF OPERATIONS

One counter intuitive, but central notion, to SQL is that the listed order of the specified commands does not correspond to the order in which they are executed. There is a mandated order to the evaluation of the clauses.

1. FROM and JOINs

The FROM clause, and subsequent JOINs are first executed to determine the set of data that is being queried. This includes subqueries in this clause, and can cause temporary tables to be created.

2. WHERE

Then any WHERE constraints are applied to the individual rows, and rows that do not satisfy the constraint are discarded. Each of the constraints can only access columns directly from the tables requested in the FROM clause. Aliases in the SELECT part of the query are not accessible since they may include expressions dependent on parts of the query that have not yet executed.

3. GROUP BY

Remaining rows after the WHERE constraints are applied are then grouped based on common values in the column specified in the GROUP BY clause. As a result of the grouping, there will only be as many rows as there are unique values in that column. This means that you should only use this when you have aggregate functions in your query.

4. HAVING

If the query has a GROUP BY clause, then any constraints of a HAVING clause are applied to the grouped rows. Like the WHERE clause, aliases may also not be accessible from this step.

5. SELECT

Any expressions in the SELECT part of the query are finally computed.

6. DISTINCT

Of the remaining rows, rows with duplicate values in the column marked as DISTINCT will be discarded.

7. ORDER BY

If an ORDER BY is specified, the rows are then sorted by the specified data in either ascending or descending order. Since all the expressions in the SELECT part of the query have been computed, you can reference aliases at this point.

8. LIMIT / OFFSET

Last, the rows that fall outside the range specified by the LIMIT and OFFSET are discarded.

ORDER OF OPERATIONS HERE

Here is the order of operations on our previous group example.

```
SELECT CustomerId, COUNT(*) AS NumOrders
FROM Orders
GROUP BY CustomerId
ORDER BY NumOrders DESC
LIMIT 10;

+------------+-----------+
| CustomerId | NumOrders |
+------------+-----------+
|      25806 |        25 |
|      94364 |        25 |
|      96968 |        24 |
|      33646 |        24 |
|      57572 |        23 |
|      89204 |        23 |
|      27518 |        23 |
|      29451 |        23 |
|      26682 |        23 |
|      36709 |        22 |
+------------+-----------+
10 rows in set (1.72 sec)
```

First, we determine our data set. Here it is trivial as we have only Orders.

Then we GROUP the Orders on CustomerID.

We evaluate the SELECT. This means we need an aggregate function to apply to each sub-group, which is to COUNT the rows of each sub-group. We alias this count as Numorders, because we will need to use it in the ORDER clause later.

Last we use the SELECTED values as our output fields, ORDERED and up to a LIMIT of 10.

If this seems counter to how you have been conditioned to think from normal programming, you are not alone. All I can say is that you will have to get used to this "inside-out" thinking if you want to get truly comfortable with SQL. Fortunately there are a limited number of idioms (patterns) to deal with and you will soon get an intuitive understanding of the order of evaluation.

SELECT SUBQUERIES

We will often wish to feed one result (in the form of a table) into another query. These subqueries are created by nesting selects within each other.

Let's say we wish to create a list of all customers with more than 20 orders.

```
SELECT CustomerId, COUNT(*) AS NumOrders
FROM Orders
GROUP BY CustomerId
HAVING NumOrders > 20;
```

```
+------------+-----------+
| CustomerId | NumOrders |
+------------+-----------+
|       1885 |        21 |
|       2311 |        21 |
|       2344 |        21 |
|       2364 |        21 |
...
...
|      88414 |        22 |
|      89204 |        23 |
|      91017 |        21 |
|      94364 |        25 |
|      96968 |        24 |
|      99255 |        22 |
|     100068 |        21 |
+------------+-----------+
51 rows in set (1.03 sec)
```

Note that when filtering aggregated results we must use *HAVING* instead of *WHERE*.

SELECT SUBQUERIES

Now we can treat that query as a table itself. Here we just apply the count(*) to it. Next we will start connecting these together.

```
SELECT COUNT(*)
FROM (SELECT CustomerId, COUNT(*) AS NumOrders
      FROM Orders
      GROUP BY CustomerId
      HAVING NumOrders > 20) AS TopOrders;
```

```
+----------+
| COUNT(*) |
+----------+
|       51 |
+----------+
1 row in set (1.04 sec)
```

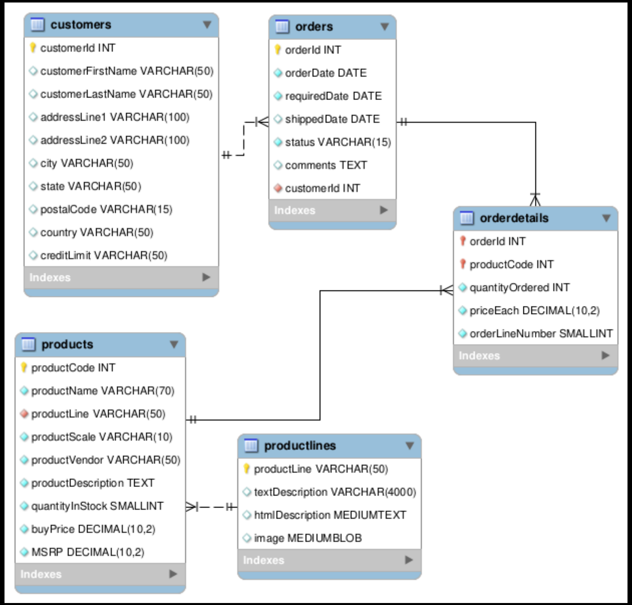
SQL insists that every derived table have a name (alias). So, we must name our subquery before we can use it, even for something as trivial as this. Here our alias is a new table.

SELECT SUBQUERIES

This might be better with more detailed customer information included. But that isn't in our Orders table. Some obviously useful info can be found in our Customers table.

```
SHOW COLUMNS FROM Customer;

+--------------+--------------+------+-----+---------+----------------+
| Field        | Type         | Null | Key | Default | Extra          |
+--------------+--------------+------+-----+---------+----------------+
| CustomerId   | int          | NO   | PRI | NULL    | auto_increment |
| FirstName    | varchar(50)  | YES  |     | NULL    |                |
| LastName     | varchar(50)  | YES  |     | NULL    |                |
| AddressLine1 | varchar(100) | YES  |     | NULL    |                |
| AddressLine2 | varchar(100) | YES  |     | NULL    |                |
| City         | varchar(50)  | YES  |     | NULL    |                |
| State        | varchar(50)  | YES  |     | NULL    |                |
| PostalCode   | varchar(15)  | YES  |     | NULL    |                |
| Country      | varchar(50)  | YES  |     | NULL    |                |
| CreditLimit  | varchar(50)  | YES  |     | NULL    |                |
+--------------+--------------+------+-----+---------+----------------+
```



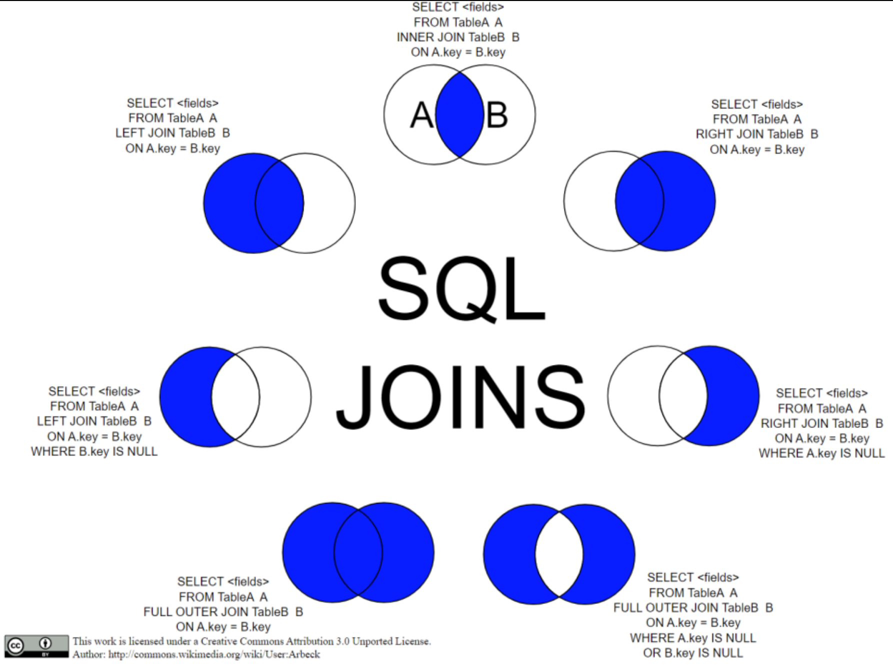
Combining Table Data

This is where the relational part of our RDBMS comes in. We want to combine data from different tables.

Our most powerful tool here will be *joins*.

There are a variety of these, and they have a logical relationship between them that is often summarized by their Venn Diagrams.

However, a few examples are generally enough to get the point across, and then this diagram will make total sense, and you won't have to memorize anything.



Inner Join

This is the default "join", and most common. It is used to collect only items with matching keys from both tables. The keys are specified with the ON clause and could be combinations of columns.

```
SELECT left_table.B, right_table.F
FROM left_table
JOIN right_table
  ON   left_table.A = right_table.E;
```

Left Table

A	B	C	D
K0	B0	C0	D0
K1	B1	C1	D1
K2	B2	C2	D2
K3	B3	C3	D3

Right Table

E	F	G	H
K1	F0	G0	H0
K1	F1	G1	H1
K0	F2	G2	H2
K6	F3	G3	H3

Result

B	F
B0	F2
B1	F0
B1	F1

The actual row order in the result could vary, unless we added an ORDER BY clause.

Implementation notes: In Pandas, these are referred to as merges.

Inner Join Example

Note that the table name is usually inferred from the FROM clause but in this JOIN columns must be disambiguated as there are multiple CustomerIDs, one in each table.

```
SELECT Customer.*, TopOrders.NumOrders FROM
(SELECT CustomerId, COUNT(*) AS NumOrders
    FROM Orders
    GROUP BY CustomerId
    HAVING NumOrders > 20) AS TopOrders
JOIN Customer
    ON TopOrders.CustomerId = Customer.CustomerId;
```

CustomerId	FirstName	LastName	AddressLine1	AddressLine2	City	State	PostalCode	Country	CreditLimit	NumOrders
1885	Karen	Roberge	44 Downey Drive		Manchester	CT	06040	USA	100	21
2311	Daniel	Jurado	709 Mildred Street		Montgomery	AL	36104	USA	100	21
2344	Lois	Hoskins	5631 West Colter Street	#2129	Glendale	AZ	85301	USA	100	21
2364	Arthur	Sosa	3145 19th Avenue Court		Greeley	CO	80631	USA	100	21
8049	Frances	Johnson	83 Oakdale Road		Newton	MA	02459	USA	100	22
9584	Claudia	Price	1452 55th Avenue	B	Oakland	CA	94621	USA	100	21
...										
...										
13803	Teresa	Pierce	154 Boca Lagoon Drive		Panama City Beach	FL	32408	USA	100	21
17740	Joseph	Grisson	8642 Yule Street		Arvada	CO	80007	USA	100	21
18420	Betty	Brown	1822 Pine Grove Court		Severn	MD	21144	USA	100	21
20647	Robert	Patton	1995 Nolensville Pike		Nashville	TN	37211	USA	100	21
25806	Kyle	Hosmer	95 Briarwood Drive		Manchester	CT	06040	USA	100	25
26682	Jana	Mapes	718 Newhall Drive		Nashville	TN	37206	USA	100	23
27518	Jason	Johnson	824 Main Street	D	Manchester	CT	06040	USA	100	23
27604	Matt	Mackey	1600 20th Street Northwest		Washington	DC	20009	USA	100	22
27982	Linda	Parks	378 Bonny Street		Grand Junction	CO	81501	USA	100	22
96968	Ellis	Chaddock	509 Sea Breeze Drive		Panama City Beach	FL	32413	USA	100	24
99255	Elaine	Soto	721 Vermont 5A		Westmore	VT	05860	USA	100	22
100068	Glenna	Lloyd	6424 Simms Street	#71	Arvada	CO	80004	USA	100	21

51 rows in set (1.06 sec)

Inner Join Example

```
Let's break this down, one subquery at a time.


SELECT Customer.*, TopOrders.NumOrders FROM
(SELECT CustomerId, COUNT(*) AS NumOrders
    FROM Orders
    GROUP BY CustomerId
    HAVING NumOrders > 20) AS TopOrders
JOIN Customer ON TopOrders.CustomerId = Customer.CustomerId;


+------------+-----------+
| CustomerId | NumOrders |
+------------+-----------+
|       1885 |        21 |
|       2311 |        21 |
|       2344 |        21 |
|       2364 |        21 |
|       8049 |        22 |
|       9584 |        21 |
|      13803 |        21 |
...
...
```

Inner Join Example

Let's break this down, one subquery at a time.

```
SELECT Customer.*, TopOrders.NumOrders FROM
(SELECT CustomerId, COUNT(*) AS NumOrders
    FROM Orders
    GROUP BY CustomerId
    HAVING NumOrders > 20) AS TopOrders
JOIN Customer ON TopOrders.CustomerId = Customer.CustomerId;
```

```
+------------+-----------+
| CustomerId | NumOrders |
+------------+-----------+
|       1885 |        21 |
|       2311 |        21 |
|       2344 |        21 |
|       2364 |        21 |
|       8049 |        22 |
|       9584 |        21 |
|      13803 |        21 |
...
...
```

```
+------------+------------+-----------+--------------------------------------+--------------+------------------+-------+------------+---------+-------------+
| CustomerId | FirstName  | LastName  | AddressLine1                         | AddressLine2 | City             | State | PostalCode | Country | CreditLimit |
+------------+------------+-----------+--------------------------------------+--------------+------------------+-------+------------+---------+-------------+
|          1 | Mary       | Yates     | 1414 East Anderson Street            | #317         | Savannah         | GA    | 31404      | USA     | 100         |
|          2 | James      | Parker    | 29 Lucian Street                     |              | Manchester       | CT    | 06040      | USA     | 100         |
|          3 | Kim        | Bond      | 1421 Floral Street Northwest         |              | Washington       | DC    | 20012      | USA     | 100         |
|          4 | Thomas     | Broadnax  | 1915 Southeast 29th Street           |              | Oklahoma City    | OK    | 73129      | USA     | 100         |
|          5 | Stephen    | Williams  | 9805 South Youngs Lane               |              | Oklahoma City    | OK    | 73159      | USA     | 100         |
|          6 | Mark       | Hinton    | 8642 Yule Street                     |              | Arvada           | CO    | 80007      | USA     | 100         |
|          7 | Deborah    | Lloyd     | 5244 West Port Au Prince Lane        |              | Glendale         | AZ    | 85306      | USA     | 100         |
|          8 | Linda      | Barnes    | 3377 Sandstone Court                 |              | Pleasanton       | CA    | 94588      | USA     | 100         |
|          9 | Donald     | Zawacki   | 4709 North Willow Avenue             |              | Bethany          | OK    | 73008      | USA     | 100         |
|         10 | Rene       | Spencer   | 1797 Pasatiempo Drive                |              | Chico            | CA    | 95928      | USA     | 100         |
...
```

Inner Join Example

Note that each subquery follows our official Order of Operations as we work our way to the topmost query.

```sql
SELECT Customer.*, TopOrders.NumOrders FROM
(SELECT CustomerId, COUNT(*) AS NumOrders
    FROM Orders
    GROUP BY CustomerId
    HAVING NumOrders > 20) AS TopOrders
JOIN Customer
    ON TopOrders.CustomerId = Customer.CustomerId;
```

CustomerId	FirstName	LastName	AddressLine1	AddressLine2	City	State	PostalCode	Country	CreditLimit	NumOrders
1885	Karen	Roberge	44 Downey Drive		Manchester	CT	06040	USA	100	21
2311	Daniel	Jurado	709 Mildred Street		Montgomery	AL	36104	USA	100	21
2344	Lois	Hoskins	5631 West Colter Street	#2129	Glendale	AZ	85301	USA	100	21
2364	Arthur	Sosa	3145 19th Avenue Court		Greeley	CO	80631	USA	100	21
8049	Frances	Johnson	83 Oakdale Road		Newton	MA	02459	USA	100	22
9584	Claudia	Price	1452 55th Avenue	B	Oakland	CA	94621	USA	100	21
...										
...										
13803	Teresa	Pierce	154 Boca Lagoon Drive		Panama City Beach	FL	32408	USA	100	21
17740	Joseph	Grisson	8642 Yule Street		Arvada	CO	80007	USA	100	21
18420	Betty	Brown	1822 Pine Grove Court		Severn	MD	21144	USA	100	21
20647	Robert	Patton	1995 Nolensville Pike		Nashville	TN	37211	USA	100	21
25806	Kyle	Hosmer	95 Briarwood Drive		Manchester	CT	06040	USA	100	25
26682	Jana	Mapes	718 Newhall Drive		Nashville	TN	37206	USA	100	23
27518	Jason	Johnson	824 Main Street	D	Manchester	CT	06040	USA	100	23
27604	Matt	Mackey	1600 20th Street Northwest		Washington	DC	20009	USA	100	22
27982	Linda	Parks	378 Bonny Street		Grand Junction	CO	81501	USA	100	22
96968	Ellis	Chaddock	509 Sea Breeze Drive		Panama City Beach	FL	32413	USA	100	24
99255	Elaine	Soto	721 Vermont 5A		Westmore	VT	05860	USA	100	22
100068	Glenna	Lloyd	6424 Simms Street	#71	Arvada	CO	80004	USA	100	21

51 rows in set (1.06 sec)

ORDER OF OPERATIONS (Again)

1. FROM and JOINs
2. WHERE
3. GROUP BY
4. HAVING
5. SELECT
6. DISTINCT
7. ORDER BY
8. LIMIT / OFFSET

Inner Join Example

Note that the table name is usually inferred from the FROM clause but in a JOIN columns must be disambiguated as there are multiple CustomerIDs.

```
SELECT Customer.*, TopOrders.NumOrders FROM
(SELECT CustomerId, COUNT(*) AS NumOrders
    FROM Orders
    GROUP BY CustomerId
    HAVING NumOrders > 20) AS TopOrders
JOIN Customer
    ON TopOrders.CustomerId = Customer.CustomerId;
```

CustomerId	FirstName	LastName	AddressLine1	AddressLine2	City	State	PostalCode	Country	CreditLimit	NumOrders
1885	Karen	Roberge	44 Downey Drive		Manchester	CT	06040	USA	100	21
2311	Daniel	Jurado	709 Mildred Street		Montgomery	AL	36104	USA	100	21
2344	Lois	Hoskins	5631 West Colter Street	#2129	Glendale	AZ	85301	USA	100	21
2364	Arthur	Sosa	3145 19th Avenue Court		Greeley	CO	80631	USA	100	21
8049	Frances	Johnson	83 Oakdale Road		Newton	MA	02459	USA	100	22
9584	Claudia	Price	1452 55th Avenue	B	Oakland	CA	94621	USA	100	21
...										
...										
13803	Teresa	Pierce	154 Boca Lagoon Drive		Panama City Beach	FL	32408	USA	100	21
17740	Joseph	Grisson	8642 Yule Street		Arvada	CO	80007	USA	100	21
18420	Betty	Brown	1822 Pine Grove Court		Severn	MD	21144	USA	100	21
20647	Robert	Patton	1995 Nolensville Pike		Nashville	TN	37211	USA	100	21
25806	Kyle	Hosmer	95 Briarwood Drive		Manchester	CT	06040	USA	100	25
26682	Jana	Mapes	718 Newhall Drive		Nashville	TN	37206	USA	100	23
27518	Jason	Johnson	824 Main Street	D	Manchester	CT	06040	USA	100	23
27604	Matt	Mackey	1600 20th Street Northwest		Washington	DC	20009	USA	100	22
27982	Linda	Parks	378 Bonny Street		Grand Junction	CO	81501	USA	100	22
96968	Ellis	Chaddock	509 Sea Breeze Drive		Panama City Beach	FL	32413	USA	100	24
99255	Elaine	Soto	721 Vermont 5A		Westmore	VT	05860	USA	100	22
100068	Glenna	Lloyd	6424 Simms Street	#71	Arvada	CO	80004	USA	100	21

51 rows in set (1.06 sec)

Inner Join Example

Let's break this down, one subquery at a time.

```
SELECT Customer.*, TopOrders.NumOrders FROM
(SELECT CustomerId, COUNT(*) AS NumOrders
    FROM Orders
    GROUP BY CustomerId
    HAVING NumOrders > 20) AS TopOrders
JOIN Customer ON TopOrders.CustomerId = Customer.CustomerId;
```

```
+------------+-----------+
| CustomerId | NumOrders |
+------------+-----------+
|       1885 |        21 |
|       2311 |        21 |
|       2344 |        21 |
|       2364 |        21 |
|       8049 |        22 |
|       9584 |        21 |
|      13803 |        21 |
...
...
```

Inner Join Example

Let's break this down, one subquery at a time.

```
SELECT Customer.*, TopOrders.NumOrders FROM
(SELECT CustomerId, COUNT(*) AS NumOrders
    FROM Orders
    GROUP BY CustomerId
    HAVING NumOrders > 20) AS TopOrders
JOIN Customer ON TopOrders.CustomerId = Customer.CustomerId;
```

```
+------------+-----------+
| CustomerId | NumOrders |
+------------+-----------+
|       1885 |        21 |
|       2311 |        21 |
|       2344 |        21 |
|       2364 |        21 |
|       8049 |        22 |
|       9584 |        21 |
|      13803 |        21 |
...
...
```

```
+------------+-------------+-----------+--------------------------------------+--------------+-----------------+-------+------------+----------+-------------+
| CustomerId | FirstName   | LastName  | AddressLine1                         | AddressLine2 | City            | State | PostalCode | Country  | CreditLimit |
+------------+-------------+-----------+--------------------------------------+--------------+-----------------+-------+------------+----------+-------------+
|          1 | Mary        | Yates     | 1414 East Anderson Street            | #317         | Savannah        | GA    | 31404      | USA      | 100         |
|          2 | James       | Parker    | 29 Lucian Street                     |              | Manchester      | CT    | 06040      | USA      | 100         |
|          3 | Kim         | Bond      | 1421 Floral Street Northwest         |              | Washington      | DC    | 20012      | USA      | 100         |
|          4 | Thomas      | Broadnax  | 1915 Southeast 29th Street           |              | Oklahoma City   | OK    | 73129      | USA      | 100         |
|          5 | Stephen     | Williams  | 9805 South Youngs Lane               |              | Oklahoma City   | OK    | 73159      | USA      | 100         |
|          6 | Mark        | Hinton    | 8642 Yule Street                     |              | Arvada          | CO    | 80007      | USA      | 100         |
|          7 | Deborah     | Lloyd     | 5244 West Port Au Prince Lane        |              | Glendale        | AZ    | 85306      | USA      | 100         |
|          8 | Linda       | Barnes    | 3377 Sandstone Court                 |              | Pleasanton      | CA    | 94588      | USA      | 100         |
|          9 | Donald      | Zawacki   | 4709 North Willow Avenue             |              | Bethany         | OK    | 73008      | USA      | 100         |
|         10 | Rene        | Spencer   | 1797 Pasatiempo Drive                |              | Chico           | CA    | 95928      | USA      | 100         |
...
```

Inner Join Example

Note that the table name is usually inferred from the FROM clause but in a JOIN columns must be disambiguated as there are multiple CustomerIDs.

```
SELECT Customer.*, TopOrders.NumOrders FROM
(SELECT CustomerId, COUNT(*) AS NumOrders
    FROM Orders
    GROUP BY CustomerId
    HAVING NumOrders > 20) AS TopOrders
JOIN Customer
    ON TopOrders.CustomerId = Customer.CustomerId;
```

CustomerId	FirstName	LastName	AddressLine1	AddressLine2	City	State	PostalCode	Country	CreditLimit	NumOrders
1885	Karen	Roberge	44 Downey Drive		Manchester	CT	06040	USA	100	21
2311	Daniel	Jurado	709 Mildred Street		Montgomery	AL	36104	USA	100	21
2344	Lois	Hoskins	5631 West Colter Street	#2129	Glendale	AZ	85301	USA	100	21
2364	Arthur	Sosa	3145 19th Avenue Court		Greeley	CO	80631	USA	100	21
8049	Frances	Johnson	83 Oakdale Road		Newton	MA	02459	USA	100	22
9584	Claudia	Price	1452 55th Avenue	B	Oakland	CA	94621	USA	100	21
...										
...										
13803	Teresa	Pierce	154 Boca Lagoon Drive		Panama City Beach	FL	32408	USA	100	21
17740	Joseph	Grisson	8642 Yule Street		Arvada	CO	80007	USA	100	21
18420	Betty	Brown	1822 Pine Grove Court		Severn	MD	21144	USA	100	21
20647	Robert	Patton	1995 Nolensville Pike		Nashville	TN	37211	USA	100	21
25806	Kyle	Hosmer	95 Briarwood Drive		Manchester	CT	06040	USA	100	25
26682	Jana	Mapes	718 Newhall Drive		Nashville	TN	37206	USA	100	23
27518	Jason	Johnson	824 Main Street	D	Manchester	CT	06040	USA	100	23
27604	Matt	Mackey	1600 20th Street Northwest		Washington	DC	20009	USA	100	22
27982	Linda	Parks	378 Bonny Street		Grand Junction	CO	81501	USA	100	22
96968	Ellis	Chaddock	509 Sea Breeze Drive		Panama City Beach	FL	32413	USA	100	24
99255	Elaine	Soto	721 Vermont 5A		Westmore	VT	05860	USA	100	22
100068	Glenna	Lloyd	6424 Simms Street	#71	Arvada	CO	80004	USA	100	21

51 rows in set (1.06 sec)

Views

As our queries, and subqueries, get more complex it becomes cumbersome and inefficient to keep recreating them. A VIEW will give us the ability to capture these as temporary tables.

```
CREATE VIEW TopCustomers AS
SELECT Customer.* FROM (SELECT CustomerId, COUNT(*) AS NumOrders
FROM Orders
  GROUP BY CustomerId
  HAVING NumOrders > 20) AS TopOrders
JOIN Customer
  ON TopOrders.CustomerId = Customer.CustomerId;
```

```
SHOW TABLES;
```

```
+-------------------+
| Tables_in_urbanic |
+-------------------+
| Customer          |
| Line              |
| OrderDetail       |
| Orders            |
| Product           |
| TopCustomers      |
| Vendor            |
+-------------------+
7 rows in set (0.00 sec)
```

A Useful View

Marketing has asked us to identify all the leather products customers have ordered.

```
CREATE VIEW LeatherOrders AS
SELECT Orders.*, orderprod.Fabric, orderprod.ProductId, orderprod.Item, orderprod.Color
FROM Orders
INNER JOIN (SELECT OrderDetail.OrderId, Product.Fabric, Product.ProductId, Product.Item, Product.Color
            FROM OrderDetail
            INNER JOIN Product
            ON OrderDetail.ProductId = Product.ProductId) AS orderprod
ON Orders.OrderId = orderprod.OrderId
WHERE orderprod.Fabric
LIKE 'Leather%'
ORDER BY Orders.OrderId;
```

OrderId	Date	DateRequired	DateShipped	Status	Comments	CustomerId	Fabric	ProductId	Item	Color
14	1995-05-23	1995-06-11	1995-06-10	OK		28781	Leather	21876	Slacks	White
18	1995-05-23	1995-05-29	1995-05-29	OK		57443	Leather	27031	Blazer	Silver
20	1995-05-23	1995-05-23	1995-05-26	OK		73528	Leather	22949	Dungarees	Cyan
22	1995-05-23	1995-05-30	1995-06-10	OK		30334	Leather	39039	Fedora	Pink
34	1995-05-23	1995-06-04	1995-06-04	OK		13149	Leather	36187	Swimsuit	Magenta
40	1995-05-23	1995-06-07	1995-06-04	OK		66536	Leather	39541	Scarf	Orange
60	1995-05-23	1995-06-04	1995-06-04	OK		71618	Leather	22458	Pants	Plum
65	1995-05-23	1995-06-02	1995-06-03	OK		62043	Leather	40680	Jacket	Navy
69	1995-05-23	1995-06-02	1995-05-23	OK		46767	Leather	40748	Jacket	Bl

...
...
...

We can match strings with the LIKE keyword and the '%' symbol works as a wildcard.

More Joins

We wish to include all of our top customers in a possible "Leather Sale" promotion even if they don't have a leather order.

LEFT JOIN will include all elements from the left table and matching ones from the right table. Unmatched values will be shown as NULL.

```
SELECT TopCustomers.customerID,LeatherOrders.OrderId,LeatherOrders.Fabric
FROM TopCustomers
LEFT JOIN LeatherOrders
   ON TopCustomers.CustomerId = LeatherOrders.CustomerId;
```

```
+------------+---------+---------+
| CustomerId | OrderId | Fabric  |
+------------+---------+---------+
|       1885 |   81257 | Leather |
|       1885 |  238207 | Leather |
|       2364 |  359837 | Leather |
|       2364 |  488584 | Leather |
...
...
|      39578 |  414186 | Leather |
|      39578 |  650871 | Leather |
|      39578 |  705664 | Leather |
|      41072 |    NULL | NULL    |
|      47511 |  529563 | Leather |
|      47511 |  676164 | Leather |
|      50446 |  598894 | Leather |
|      50446 |  821076 | Leather |
|      51859 |  660109 | Leather |
|      52328 |  160968 | Leather |
|      52328 |  750126 | Leather |
+------------+---------+---------+
```

We can match strings with the LIKE keyword and the '%' symbol works as a wildcard.

Left Join

LEFT JOIN will include all elements from the left table and matching ones from the right table. Unmatched values will be shown as NULL.

```
SELECT left_table.B, right_table.F
FROM left_table
LEFT JOIN right_table
  ON   left_table.A = right_table.E;
```

Left Table

A	B	C	D
K0	B0	C0	D0
K1	B1	C1	D1
K2	B2	C2	D2
K3	B3	C3	D3

Right Table

E	F	G	H
K1	F0	G0	H0
K1	F1	G1	H1
K0	F2	G2	H2
K6	F3	G3	H3

Result

B	F
B0	F2
B1	F0
B1	F1
B2	NULL
B3	NULL

The actual row order in the result could vary, unless we added an ORDER BY clause.

Implementation notes: In Pandas, these are referred to as merges.

Right Join

As you might expect by now, RIGHT JOIN will include all elements from the right table and matching ones from the left table. Unmatched values will be shown as NULL.

```
SELECT left_table.B, right_table.F
FROM left_table
RIGHT JOIN right_table
  ON   left_table.A = right_table.E;
```

Left Table

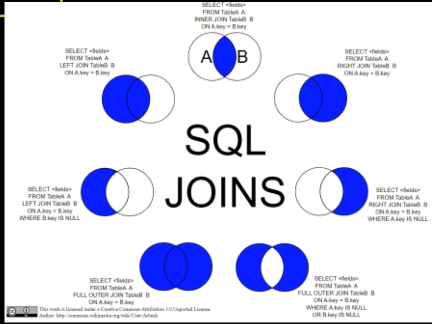
A	B	C	D
K0	B0	C0	D0
K1	B1	C1	D1
K2	B2	C2	D2
K3	B3	C3	D3

Right Table

E	F	G	H
K1	F0	G0	H0
K1	F1	G1	H1
K0	F2	G2	H2
K6	F3	G3	H3

Result

B	F
B1	F0
B1	F1
B0	F2
NULL	F3



The actual row order in the result could vary, unless we added an ORDER BY clause.

Implementation notes: In Pandas, these are referred to as merges.

Another Interesting Join

We'd also like to know which products haven't been of interest to top customers. We'll do a RIGHT JOIN to find all the leather products with no orders by this group. We also use the keyword *IN* in this query to filter down to only TopCustomer orders. Other comparison operators we will see are *NOT IN, BETWEEN, IS, IS NOT, IS NOT NULL*.

```
SELECT topLeatherOrders.OrderId, topLeatherOrders.CustomerId, LeatherProduct.Color, LeatherProduct.Fabric,
       LeatherProduct.Item, LeatherProduct.ProductId
FROM (SELECT *
      FROM LeatherOrders
         WHERE LeatherOrders.CustomerId IN (SELECT CustomerId FROM TopCustomers)) AS topLeatherOrders
      RIGHT JOIN (SELECT ProductId, Fabric, Color, Item FROM Product WHERE Fabric LIKE 'Leather%') AS LeatherProduct
         ON topLeatherOrders.ProductId = LeatherProduct.ProductId
LIMIT 10;
```

OrderId	CustomerId	Color	Fabric	Item	ProductId
NULL	NULL	Red	Leather	Skirt	20659
NULL	NULL	Navy	Leather	Skirt	20660
NULL	NULL	Cyan	Leather	Skirt	20661
NULL	NULL	Black	Leather	Skirt	20662
NULL	NULL	Brown	Leather	Skirt	20663
234297	17740	Ocher	Leather	Skirt	20664
NULL	NULL	Orange	Leather	Skirt	20665
NULL	NULL	White	Leather	Skirt	20666
NULL	NULL	Green	Leather	Skirt	20667
NULL	NULL	Puce	Leather	Skirt	20668

10 rows in set (1.14 sec)

One last refinement

The NULL entries in the left table are what we're after here, so we can add one more clause. NULL values require the IS keyword since a value can not be equal to NULL. IS tests for values that are either TRUE, FALSE or NULL.

```
SELECT OrderId, CustomerId, LeatherProduct.Color, LeatherProduct.Fabric, LeatherProduct.Item,
       LeatherProduct.ProductId
FROM (SELECT *
      FROM LeatherOrders
      WHERE LeatherOrders.CustomerId IN (SELECT CustomerId FROM TopCustomers)) AS topLeatherOrders
      RIGHT JOIN (SELECT ProductId,Fabric,Color,Item FROM Product WHERE Fabric LIKE 'Leather%') AS LeatherProduct
          ON topLeatherOrders.ProductId = LeatherProduct.ProductId
      WHERE OrderId IS NULL
LIMIT 10;
```

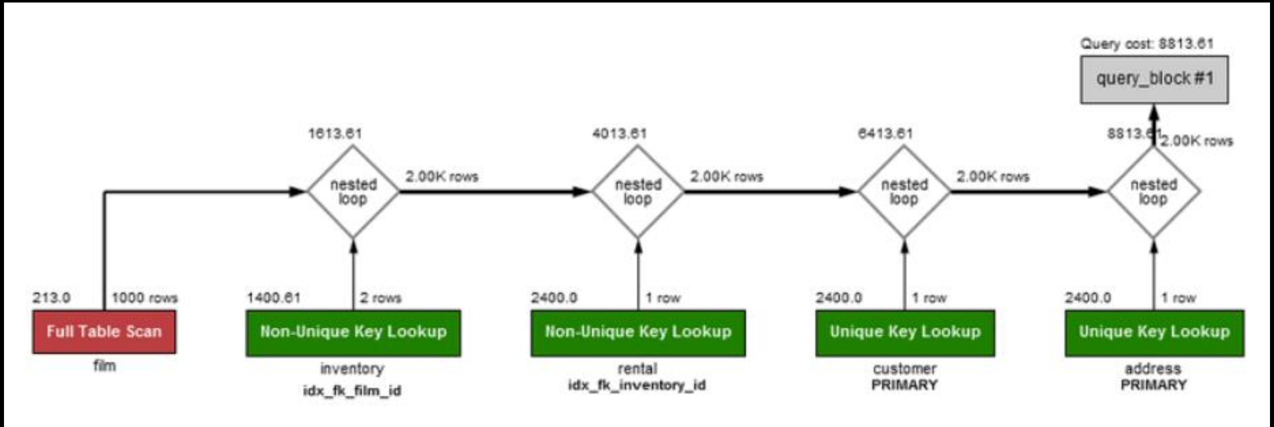
OrderId	CustomerId	Color	Fabric	Item	ProductId
NULL	NULL	Red	Leather	Skirt	20659
NULL	NULL	Navy	Leather	Skirt	20660
NULL	NULL	Cyan	Leather	Skirt	20661
NULL	NULL	Black	Leather	Skirt	20662
NULL	NULL	Brown	Leather	Skirt	20663
NULL	NULL	Orange	Leather	Skirt	20665
NULL	NULL	White	Leather	Skirt	20666
NULL	NULL	Green	Leather	Skirt	20667
NULL	NULL	Puce	Leather	Skirt	20668
NULL	NULL	Pink	Leather	Skirt	20669

Joins are Loops

You notice how we loop through the keys as we manually create our joins. This is what our database must do as well. Nested joins turn into nested loops. Here is a typical query from a classic film rental database.

```
SELECT CONCAT(customer.last_name, ', ', customer.first_name)
  AS customer, address.phone, film.title FROM rental
INNER JOIN customer ON rental.customer_id = customer.customer_id
INNER JOIN address ON customer.address_id = address.address_id
INNER JOIN inventory ON rental.inventory_id = inventory.inventory_id
INNER JOIN film ON inventory.film_id = film.film_id
WHERE rental.return_date IS NULL
AND rental_date + INTERVAL film.rental_duration DAY < CURRENT_DATE()
LIMIT 5;
```

This is how MySQL Workbench explains the operations.



Keys

If we are trying to quickly equate things from two tables, you might imagine that the organization of those tables might have a major effect on performance. Indeed, the correct selection of keys for each table is the most important consideration.

There are a variety of key types. Two are very important.

Primary Key

A column (*or possibly combination of columns!*) with unique values.

Foreign Key

A column whose values point to a Primary Key in a different table.

There are other terms for keys that are less important to know. *Candidate Keys* are any keys that could be a Primary Key. A *Unique Key* could have a single NULL value (which is not allowed for a Primary Key). A *Composite Key* is a key created from multiple columns, etc.

Keys

Primary Keys are very important as the database can use that as an index to allow us to quickly find a record. This is usually via a good *hashing algorithm*.

When we are doing a join, this allows us to quickly find any two items we are wishing to compare.

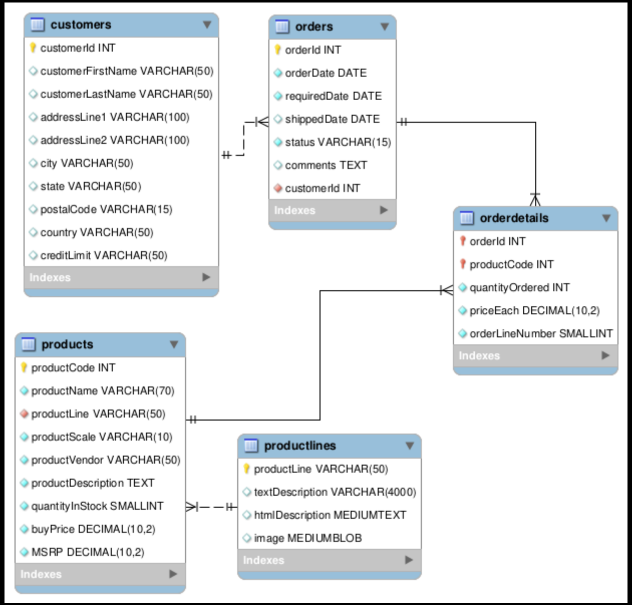
This is why we really prefer our joins to use the assigned table keys if possible.

Keys can also aid greatly in ensuring data integrity.

If it is the case that every record should be unique (order #s, for example), then using that as the primary key will enforce that condition.

A necessary relationship between data in different tables can be enforced with foreign keys. If an Order table uses a customer ID as a foreign key, they will ensure that a matching customer exists in a Customer data table.

A common default Primary Key is simply an integer that might be auto-incremented as each new record is added. In *Pandas* we always have a row number.



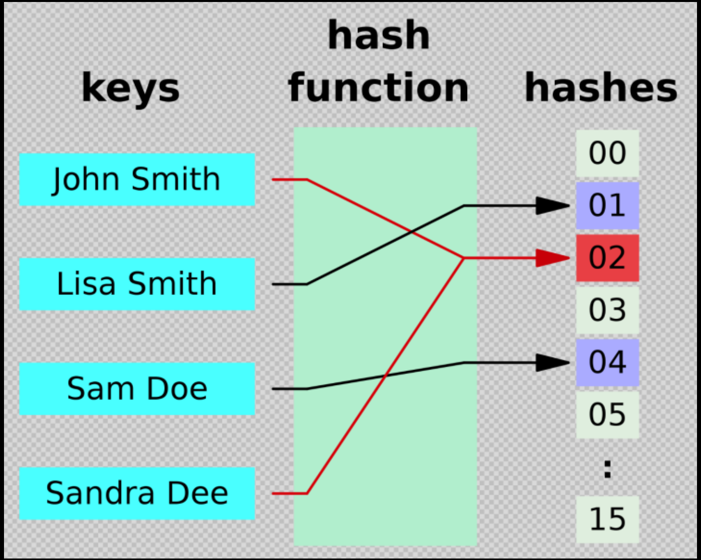
Hashing

You won't get very far in data science without hearing about how hashing is used to organize important data. It is by far the most common way to index any substantial RDBMS table.

In this context, a hashing algorithm's job is to take a key and use it to generate an index into the data storage.

From the mathematical perspective, it takes some string - of possibly arbitrary length - and generates a fixed size number. In general this means that it can't guarantee the uniqueness of that number, but you hope it does a good job of distributing the indices around. And, you hope it is fast.

A *collision* can occur, and we have various schemes to cope with that.



Without hashes, looking for "John Smith" requires us to either dig through all the stored data, or sort our data based upon the keys. This latter sounds reasonable (and sometimes is), but doesn't work so well if we are frequently adding or deleting data.

In our case, picture how important this is as our joins are looping over our keys and have to retrieve each key's associated values as quickly as possible.

Let's get creative.

So far we have just been analyzing our data. We haven't been creating, or even modifying it. Let do that.

```
CREATE DATABASE clothing;

CREATE TABLE vendors (
  vendorId int NOT NULL AUTO_INCREMENT, vendorName varchar(100) DEFAULT NULL, addressLine1 varchar(100) DEFAULT NULL,
  addressLine2 varchar(100) DEFAULT NULL, city varchar(50) DEFAULT NULL, state varchar(50) DEFAULT NULL,
  postalCode varchar(15) DEFAULT NULL, country varchar(50) DEFAULT NULL,
PRIMARY KEY (vendorId)
);
```

```
show columns from vendors;
+--------------+--------------+------+-----+---------+----------------+
| Field        | Type         | Null | Key | Default | Extra          |
+--------------+--------------+------+-----+---------+----------------+
| vendorId     | int          | NO   | PRI | NULL    | auto_increment |
| vendorName   | varchar(100) | YES  |     | NULL    |                |
| addressLine1 | varchar(100) | YES  |     | NULL    |                |
| addressLine2 | varchar(100) | YES  |     | NULL    |                |
| city         | varchar(50)  | YES  |     | NULL    |                |
| state        | varchar(50)  | YES  |     | NULL    |                |
| postalCode   | varchar(15)  | YES  |     | NULL    |                |
| country      | varchar(50)  | YES  |     | NULL    |                |
+--------------+--------------+------+-----+---------+----------------+
```

Altering Existing Tables

```
ALTER TABLE vendors ADD COLUMN comment VARCHAR(200);
```

```
show columns from vendors;
```

Field	Type	Null	Key	Default	Extra
vendorId	int	NO	PRI	NULL	auto_increment
vendorName	varchar(100)	YES		NULL	
addressLine1	varchar(100)	YES		NULL	
addressLine2	varchar(100)	YES		NULL	
city	varchar(50)	YES		NULL	
state	varchar(50)	YES		NULL	
postalCode	varchar(15)	YES		NULL	
country	varchar(50)	YES		NULL	
comment	varchar(200)	YES		NULL	

Inserting Data

```
INSERT INTO vendors(vendorName,addressLine1,addressLine2,city,state,postalCode,country,comment) VALUES ('ThreadBlasters','123
Imaginary Place',NULL,'Sampletown','PA','15217','USA',NULL);
```

```
SELECT * FROM vendors;
```

```
+----------+----------------+---------------------+--------------+------------+-------+------------+---------+---------+
| vendorId | vendorName     | addressLine1        | addressLine2 | city       | state | postalCode | country | comment |
+----------+----------------+---------------------+--------------+------------+-------+------------+---------+---------+
|        1 | ThreadBlasters | 123 Imaginary Place | NULL         | Sampletown | PA    | 15217      | USA     | NULL    |
+----------+----------------+---------------------+--------------+------------+-------+------------+---------+---------+
```

Updating Data

```
UPDATE vendors SET vendorName = 'ThreadBlazers' WHERE vendorId = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0



SELECT * FROM vendors;
+----------+---------------+---------------------+--------------+------------+-------+------------+---------+---------+
| vendorId | vendorName    | addressLine1        | addressLine2 | city       | state | postalCode | country | comment |
+----------+---------------+---------------------+--------------+------------+-------+------------+---------+---------+
|        1 | ThreadBlazers | 123 Imaginary Place | NULL         | Sampletown | PA    | 15217      | USA     | NULL    |
+----------+---------------+---------------------+--------------+------------+-------+------------+---------+---------+
1 row in set (0.00 sec)
```

Deleting Data

```
DELETE FROM vendors WHERE city = 'Sampletown';Query OK, 1 row affected (0.00 sec)
Query OK, 1 row affected (0.00 sec)
```

```
SELECT * FROM vendors;
Empty set (0.00 sec)
```

Deleting Tables

```
DROP TABLE vendors;
Query OK, 0 rows affected (0.02 sec)
```

```
SELECT * FROM vendors;
ERROR 1146 (42S02): Table ‘clothing.vendors' doesn't exist
```

SQL Injection Attacks

Consider a typical website, which asks the user to enter their username. It then constructs a string to use in querying the database for that user's info:

```
var statement = "SELECT * FROM users WHERE name = '" + userName + "'";
```

This seems reasonable. However, what if a nefarious user enters this as their username:

```
a';DROP TABLE users; SELECT * FROM userinfo WHERE 't' = 't
```

Then the SQL command that gets constructed is

```
SELECT * FROM users WHERE name = 'a';DROP TABLE users; SELECT * FROM userinfo WHERE 't' = 't';
```

And we have not only exposed all user data, but also deleted our users table.

Good practices can help to mitigate this and sanitize the inputs. Be aware.

Normalization

Any serious, semester-long, SQL course will spend some time talking about Normalization. This is the very formal process of eliminating redundant data and ensuring consistency. If you are a DB admin working for a bank, you should probably know what this is.

Otherwise, common sense and an understanding of what you are asking of the database is usually sufficient, especially for scientific data bases.

And it is important to note that databases are frequently *denormalized* to help with performance. A redundant field may avoid an expensive join.

Constraint (informal description in parentheses)	UNF (1970)	1NF (1970)	2NF (1971)	3NF (1971)	EKNF (1982)	BCNF (1974)	4NF (1977)	ETNF (2012)	5NF (1979)	DKNF (1981)	6NF (2003)
Unique rows (no duplicate records)[4]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Scalar columns (columns cannot contain relations or composite values)[5]	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Every non-prime attribute has a full functional dependency on a candidate key (attributes depend on the *whole* of every key)[5]	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
Every non-trivial functional dependency either begins with a superkey or ends with a prime attribute (attributes depend *only* on candidate keys)[5]	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓
Every non-trivial functional dependency either begins with a superkey or ends with an elementary prime attribute (a stricter form of 3NF)	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	—
Every non-trivial functional dependency begins with a superkey (a stricter form of 3NF)	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	—
Every non-trivial multivalued dependency begins with a superkey	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	—
Every join dependency has a superkey component[8]	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	—
Every join dependency has only superkey components	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	—
Every constraint is a consequence of domain constraints and key constraints	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
Every join dependency is trivial	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓

ACID

ACID is a set of properties of database transactions intended to guarantee data validity despite errors, power failures, and other mishaps. In the context of databases, a sequence of database operations that satisfies the ACID properties (which can be perceived as a single logical operation on the data) is called a transaction. For example, a transfer of funds from one bank account to another, even involving multiple changes such as debiting one account and crediting another, is a single transaction.

Atomicity

An atomic system must guarantee atomicity in each and every situation, including power failures, errors, and crashes. A guarantee of atomicity prevents updates to the database from occurring only partially, which can cause greater problems than rejecting the whole series outright.

Consistency

Consistency ensures that a transaction can only bring the database from one consistent state to another, preserving database invariants: any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof. This prevents database corruption by an illegal transaction.

Isolation

Isolation ensures that concurrent execution of transactions leaves the database in the same state that would have been obtained if the transactions were executed sequentially.

Durability

Durability guarantees that once a transaction has been committed, it will remain committed even in the case of a system failure (e.g., power outage or crash). This usually means that completed transactions (or their effects) are recorded in non-volatile memory.

Triggers

Triggers allow us to dynamically enforce conditions and check integrity whenever certain actions occur. We'll just discuss this simple example, but they can involve some complex behavior and take advantage of some of the more dynamic programming capabilities of SQL like variables and control flow.

```
CREATE TRIGGER upd_check BEFORE UPDATE ON account
FOR EACH ROW
BEGIN
   IF NEW.amount < 0 THEN
         SET NEW.amount = 0;
      ELSEIF NEW.amount > 100 THEN
         SET NEW.amount = 100;
      END IF;
END;
```

Procedures

Since we have strayed into the programmatic capabilities of SQL, I must at least mention stored procedures.

The simplest ones look like

```
CREATE PROCEDURE SelectAllCustomers
AS
SELECT * FROM Customers
GO;

EXEC SelectAllCustomers;
```

But, what is the point of that? In reality, these are used to capture serious business logic and have features like:

- Parameters
- Variables
- Conditional Statements: IF, CASE
- Loops: LOOP, WHILE, REPEAT

None of this should intimidate any basic programmer, but we aren't going to dive into the details here.

Connectors

MySQL is a server, and Connectors allow clients using other languages to connect to the databases. This is extremely common in web applications, which are written in their own native languages (Javascript or maybe a Java backend). Here is a query to our database from Python:

```
import datetime
import mysql.connector

cnx = mysql.connector.connect(user=‘adaptuser', database=‘clothing')
cursor = cnx.cursor()

query = ("SELECT orderId, orderDate FROM orders "
         "WHERE orderDate BETWEEN %s AND %s")

sale_start = datetime.date(1999, 1, 1)
sale_end = datetime.date(1999, 12, 31)

cursor.execute(query, (sale_start, sale_end))

for (orderId, orderDate) in cursor:
  print(“Order {} placed on {:%d %b %Y}".format(
    orderId, orderDate))

cursor.close()
cnx.close()
```

Input and Output

There are many ways to get data into and out of a database. The most direct is from a text file, although formats such as XML are supported too. *LOAD DATA* will do this and has many options. The most basic looks like this

```
LOAD DATA LOCAL INFILE '/path/product.txt' INTO TABLE product;
```

Likewise, output can be done with a *SELECT* statement using *INTO OUTFILE*

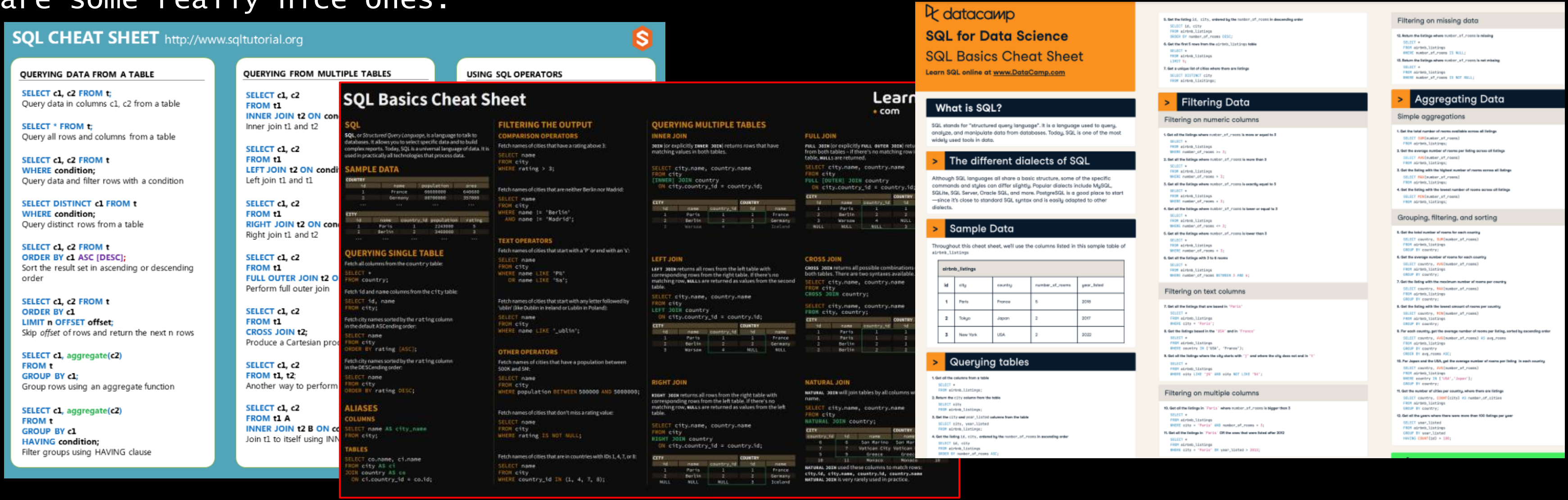
```
SELECT * FROM orders WHERE orderDate < '1997-01-01' LIMIT 100 INTO OUTFILE 'MySQLHomework.txt';
```

Documentation

There is of course much we haven't covered and, even considering the topics we have, you want to have some documentation as a guide. The best place to go is

https://dev.mysql.com/doc/refman/8.1/en/

And, I like to have a "cheat sheet" by my side every time I have to revisit this subject. There are some really nice ones:



Just the first pages shown. Go to the URLs to get the full ones.

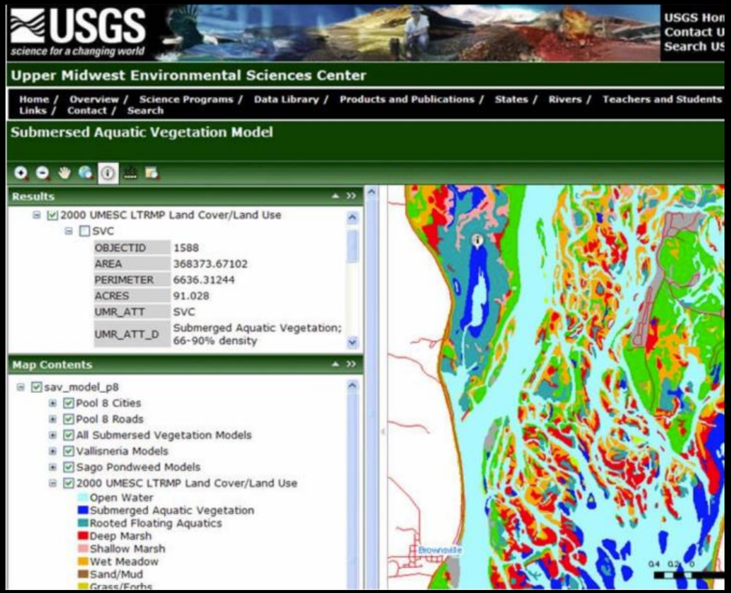
A little GIS

MySQL includes functions that allow us to measure geographic distance. This is a lightweight introduction to the important data science domain of Geographic Information Systems (GIS).

We are going to draw upon a Zips table that you have already have in your database. Amongst the other fields, you can find the geographic center of each zip code (in the US these are the same as Postal Codes). That will be sufficient location resolution for our next task.

We can find our Customer's (rough) locations from this:

```
SELECT CustomerID, Lng, Lat, PostalCode
FROM Customer
JOIN Zips
  ON Customer.PostalCode = Zips.zip;
```



Or we can find the coordinates for one particular Cambridge, MA zip code, 02139:

```
SELECT Lng, Lat FROM Zips WHERE Zip = 02139;
```

ST_DISTANCE_SPHERE

Let's find out how far away from Cambridge customer number 1 is. *ST_DISTANCE_SPHERE()* takes coordinates in the order longitude then latitude and returns a distance in meters:

```
SELECT CustomerId, Lng, Lat,  Zip,
      ST_DISTANCE_SPHERE(POINT(Lng,Lat), POINT(-71.10253,42.36224)) AS distance
FROM Customer JOIN Zips on PostalCode = Zip
WHERE CustomerId = 1;
```

```
+------------+-----------+----------+-------+-------------------+
| CustomerId | Lng       | Lat      | Zip   | distance          |
+------------+-----------+----------+-------+-------------------+
|          1 | -81.05367 | 32.05064 | 31404 | 1443923.3563356877 |
+------------+-----------+----------+-------+-------------------+
```

1443 kilometers. Pretty far!

Assignment

Our company is going to test a drone delivery service, and we wish to find the three best launch sites within 100 km of our Cambridge, MA headquarters (in zip code `02139`). The drones can only fly a few km from their launch site. The 100 km limit is so our service techs don't have to commute too far.

Our initial search will consider which zip codes, within that eligible radius, have the most orders. We will locate our three sites within those three zip codes.

We could probably do a more in-depth analysis and consider the relative locations of the sites, or clusters of zip codes that might be within flight range of our drones. But we will start with this basic approach.

Your final email submission must include:

- Your SQL script
- Your answer

This assignment is due Noctober 55th. We will review that day, so no late credit is possible. This is 5 points of your grade, because it shouldn't take much effort. Nevertheless, I suggest you don't wait until the last minute.

Solution Review:

Drone Delivery with SQL

My Plan

There is no one correct answer, and there are several reasonable paths to the solution (as well as countless awkward - but acceptable - ones).

My initial thought was to do the obvious, in which case I would:

1. Find the eligible zip codes
   a) Get the coordinates for all the zip codes
   b) Get the distances from Cambridge
   c) Keep only those <100km

2. Get the orders for each zipcode
   Maybe I should could have tried to process only those from step 1...
   but this approach was stupidly obvious and lazy and and we aren't
   grading on performance, today. You may have been smarter.

3. Use the first step to filter the results of the second step
   In other words, inner join the first table with the second

4. Now just sort the results and take the top 3

Find The Eligible Zip Codes

This is the obvious thing to do with the GIS distance function:

```
SELECT Zip
FROM Zips
   WHERE ST_Distance_Sphere(point(Lng,Lat),point(-71.10253,42.36224)) < 100000
```

But, we really shouldn't hard code this number in there, so I computed it properly

```
SELECT Zip
FROM Zips
   WHERE ST_Distance_Sphere( point(Lng,Lat),point((SELECT Lng FROM Zips WHERE Zip=02139),(SELECT Lat FROM Zips WHERE Zip=02139))) < 100000;
```

```
+------+
| Zip  |
+------+
| 2133 |
| 2203 |
| 2205 |
| 2872 |
| 2542 |
...
...
| 2115 |
| 2113 |
+------+
552 rows in set (0.21 sec)
```

Looks reasonable. We found 552 zip codes.

Get The Order Counts For Each Zip Code

This looks a lot like what we did in our GROUP BY example earlier. Let's do the analog here:

```
SELECT PostalCode, count(*) AS OrdersInPostalCode
FROM Orders
JOIN
Customer
  ON Orders.CustomerID=Customer.CustomerID
GROUP BY PostalCode;
```

```
+------------+--------------------+
| PostalCode | OrdersInPostalCode |
+------------+--------------------+
| 01001      |                819 |
| 01007      |                812 |
| 01010      |                868 |
| 01020      |                751 |
| 01027      |               1649 |
...
...
...
| 99518      |               7361 |
| 99567      |               4878 |
| 99577      |              14453 |
| 99603      |                809 |
| 99611      |               1530 |
| 99669      |                763 |
+------------+--------------------+
684 rows in set (19.83 sec)
```

Note that this step does take some time. We might want to consider implementing our logic so that we only perform this on the eligible zip codes.

Combine The First Two Steps

Now I have a list of eligible zip codes, and I have all the zip code order counts. Sounds like I just want to find the common items: an inner join. Something like

```
SELECT Results_I_Want_Displayed
FROM Eligible_Zips
JOIN
Orders_by_Zip
  ON Zips_as_key
```

Now we just cut and paste the right stuff from our first two queries. However, after we do that we will find that we need to alias those tables/queries so that we can properly specify the keys and results.

Building Our Query

Here is what I ended up with.

```
SELECT AllowedZips.Zip, OrdersByPostalCode.OrdersInThisPostalCode
FROM
(SELECT Zip
   FROM Zips
      WHERE ST_Distance_Sphere( point(Lng,Lat),point((SELECT Lng FROM Zips WHERE Zip=02139),(SELECT Lat FROM Zips WHERE Zip=02139))) < 100000
    ) AS AllowedZips
JOIN
(SELECT PostalCode, count(*) AS OrdersInThisPostalCode
   FROM Orders
   JOIN
   Customer
     ON Orders.CustomerID=Customer.CustomerID
   GROUP BY PostalCode) AS OrdersByPostalCode
ON
OrdersByPostalCode.PostalCode=AllowedZips.Zip;
```

```
+------+-----------------------+
| Zip  | OrdersInThisPostalCode |
+------+-----------------------+
| 1010 |                   868 |
| 1031 |                   676 |
| 1331 |                  2538 |
| 1420 |                   737 |
| 1440 |                  1529 |
...
...
...
| 2745 |                  1670 |
| 2747 |                   781 |
| 2748 |                   815 |
| 2766 |                   711 |
| 2790 |                   899 |
+------+-----------------------+
108 rows in set (21.89 sec)
```

The Two Subqueries

Let's make sure you understand the pieces. Here are our two original queries.

```
SELECT AllowedZips.Zip, OrdersByPostalCode.OrdersInThisPostalCode
FROM
(SELECT Zip
   FROM Zips
      WHERE ST_Distance_Sphere( point(Lng,Lat),point((SELECT Lng FROM Zips WHERE Zip=02139),(SELECT Lat FROM Zips WHERE..< 100000
    ) AS AllowedZips
JOIN
(SELECT PostalCode, count(*) AS OrdersInThisPostalCode
   FROM Orders
   JOIN
   Customer
     ON Orders.CustomerID=Customer.CustomerID
   GROUP BY PostalCode) AS OrdersByPostalCode
ON
OrdersByPostalCode.PostalCode=AllowedZips.Zip;
```

Results

These are our results. Also make sure you see how the aliases are used here.

```
SELECT AllowedZips.Zip, OrdersByPostalCode.OrdersInThisPostalCode
FROM
(SELECT Zip
   FROM Zips
      WHERE ST_Distance_Sphere( point(Lng,Lat),point((SELECT Lng FROM Zips WHERE Zip=02139),(SELECT Lat FROM Zips WHERE Zip=02139))) < 100000
    ) AS AllowedZips
JOIN
(SELECT PostalCode, count(*) AS OrdersInThisPostalCode
   FROM Orders
   JOIN
   Customer
     ON Orders.CustomerID=Customer.CustomerID
   GROUP BY PostalCode) AS OrdersByPostalCode
ON
OrdersByPostalCode.PostalCode=AllowedZips.Zip;
```

```
+------+-----------------------+
| Zip  | OrdersInThisPostalCode |
+------+-----------------------+
| 1010 |                   868 |
| 1031 |                   676 |
| 1331 |                  2538 |
| 1420 |                   737 |
| 1440 |                  1529 |
...
...
...
| 2745 |                  1670 |
| 2747 |                   781 |
| 2748 |                   815 |
| 2766 |                   711 |
| 2790 |                   899 |
+------+-----------------------+
```

Final Answer

All that remains is to sort and select the top 3.

```
SELECT AllowedZips.Zip, OrdersByPostalCode.OrdersInThisPostalCode
FROM
(SELECT Zip
   FROM Zips
      WHERE ST_Distance_Sphere( point(Lng,Lat),point((SELECT Lng FROM Zips WHERE Zip=02139),(SELECT Lat FROM Zips WHERE Zip=02139))) < 100000
    ) AS AllowedZips
JOIN
(SELECT PostalCode, count(*) AS OrdersInThisPostalCode
   FROM Orders
   JOIN
   Customer
     ON Orders.CustomerID=Customer.CustomerID
   GROUP BY PostalCode) AS OrdersByPostalCode
ON
OrdersByPostalCode.PostalCode=AllowedZips.Zip
ORDER BY OrdersByPostalCode.OrdersInThisPostalCode DESC
LIMIT 3;
```

```
+------+-----------------------+
| Zip  | OrdersInThisPostalCode |
+------+-----------------------+
| 2169 |                  4688 |
| 2155 |                  4127 |
| 2446 |                  2622 |
+------+-----------------------+
3 rows in set (18.13 sec)
```

Actual Variables

One thing that seems especially awkward is the way we had to refer to our fixed Cambridge Lng and Lat values. This begged for a "do it once" solution. Indeed we do have regular variables we can use for these tasks.

```
SET @HeadquartersLng = (SELECT Lng FROM Zips WHERE Zip=02139);
SET @HeadquartersLat = (SELECT Lat FROM Zips WHERE Zip=02139);
```

And our solution cleans up a bit.

```
SELECT AllowedZips.Zip, OrdersByPostalCode.OrdersInThisPostalCode
FROM
(SELECT Zip
   FROM Zips
      WHERE ST_Distance_Sphere( point(Lng,Lat),point(@HeadquartersLng,@HeadquartersLat) ) < 100000 ) AS AllowedZips
JOIN
(SELECT PostalCode, count(*) AS OrdersInThisPostalCode
   FROM Orders
   JOIN
   Customer
     ON Orders.CustomerID=Customer.CustomerID
   GROUP BY PostalCode) AS OrdersByPostalCode
ON
OrdersByPostalCode.PostalCode=AllowedZips.Zip
ORDER BY OrdersByPostalCode.OrdersInThisPostalCode DESC
LIMIT 3;
```

Why didn't I mention this capability earlier? Mostly because you might tend to start thinking like a sequential programming, using variables to move from one state to the next. The standard SQL paradigm is to build these "inside out" queries instead, and you have to understand that if you want to swim in those waters. But in this case, this is perfectly acceptable.

A Little More Efficient

As mentioned earlier, we could also be a little more thoughtful about minimizing the larger joins or groupings by eliminating ineligible zip codes early. Here is an approach using that philosophy.

```
SELECT COUNT(*) as ordersbyzip, PostalCode
FROM Orders
JOIN Customer
  ON Orders.CustomerId = Customer.CustomerId
  WHERE Customer.PostalCode IN (SELECT DISTINCT distances.PostalCode
                                FROM ( SELECT * FROM
                                       (SELECT Customer.PostalCode,ST_Distance_Sphere( point(Lng,Lat),
                                                   point((SELECT Lng FROM Zips WHERE Zip = 02139),
                                                         (SELECT Lat FROM Zips WHERE Zip = 02139 ))) AS distance
                                        FROM Customer
                                        JOIN Zips
                                          ON PostalCode = Zip) AS alldistances
                                   WHERE alldistances.distance < 100000)
                                   AS distances)
GROUP BY PostalCode
ORDER BY ordersbyzip DESC
LIMIT 3;

+-------------+------------+
| ordersbyzip | PostalCode |
+-------------+------------+
|        4688 | 02169      |
|        4127 | 02155      |
|        2622 | 02446      |
+-------------+------------+
3 rows in set (7.23 sec)
```

This is well over twice as fast. These kind of optimizations abound in the SQL database world.

An Interesting Submission

A submission from an earlier student had an interesting approach.

```
SELECT Zip, COUNT(OrderId) AS NumOrders
FROM
(SELECT CustomerId, Lng, Lat, Zip, ST_DISTANCE_SPHERE(POINT(Lng,Lat), POINT(-71.10253,42.36224)) AS distance
  FROM Customer JOIN Zips on PostalCode = Zip HAVING distance <100000) AS WithinRadius
JOIN
Orders
  ON WithinRadius.CustomerId = Orders.CustomerId
GROUP BY Zip
ORDER BY NumOrders DESC
LIMIT 3;
```

The peculiar thing here is this bit:

```
SELECT CustomerId, Lng, Lat, Zip, ST_DISTANCE_SPHERE(POINT(Lng,Lat), POINT(-71.10253,42.36224)) AS distance
FROM
Customer
JOIN Zips
  ON PostalCode = Zip
  HAVING distance <100000
```

HAVING should only be used with GROUP BY. The answer here can be found in the MySQL Reference Manual:

The SQL standard requires that HAVING must reference only columns in the GROUP BY clause or columns used in aggregate functions. However, MySQL supports an extension to this behavior…

It then goes on to explain the complications involved with this extension. Don't do this.