Project T Final - SQL

Justin Haug, Ryan Van de Water & David Zagaynov

November 2020

1 Introduction

As an ML Engineer you may be asked to pull data from a company-specific RDBMS or Relational Database Management System. Normally, companies use SQL to carry out these tasks as it is easy to understand and read queries. SQL is a very high-level language that allows ML engineers to access specific data from a large Relational Database Management System.

SQL stands for Structured Query Language. SQL is a ANSI and ISO standard, but different variations of SQL exist with slight differences. The most popular variations are MySQL, SQLite, PostgresSQL, and MongoDB. The most common commands between all of these languages are identical in form and function.

The data accessed by SQL calls is stored in RDBMS, with are Relational Database Management Systems. An RDBMS is composed of unordered tables. A tables is a set of data entries, most commonly represented as rows (data point) and columns (data attribute). The columns of a table are called Fields, and are named. For example, fields for a Berkeley student could be GPA, SSID, and Name.

SQL developed in the 1970s by IBM researchers Raymond Boyce and Donald Chamberlin. The first version, called SEQUEL, was designed to interact wit IBM's quasi-relational database system, called System R. In 1979, Oracle released the first commercially available version of SQL, called Oracle V2.

2 Review

2.1 Select Tables

When working with SQL, the SELECT statement is used to select data from a database. The return of the SELECT statement is a result table, called the result-set.

The SELECT statement is used in conjunction with FROM, to specify the table that the data is selected from.

Syntactically, as SELECT statement would look like this:

SELECT columnNameA, columnNameB, ... FROM tableName

columnA and columnB, are the fields that are selected from the specified table.

To select an entire table, the * symbol is used. For example: SELECT * FROM tableName

Example 1: Select names and grades from the Students table. The columns of the students table includes: name, grade, age, gender.

SELECT name, grade FROM Students

2.2 Create Clause

Creating Tables in SQL is extremely useful as you can create databases within your Relational Database Management System. You can do this by creating an entirely new, independent table or by referencing pre-existing table.

Example 1: Creating an entirely new, independent table

```
CREATE *name of your table* (
*column 1 name* *column 1 datatype*
*column 2 name* *column 2 datatype*
*column 3 name* *column 3 datatype*
*...* *...*
*column ’n’ name* *column ’n’ datatype*
)
```

As you can see, in SQL we must specify the datatype by which we want the column to represent. SQL has many various datatypes, many of which you have probably seen before, but here is a quick overview of the main datatypes we will be working with.

2.3 Data Types

CHAR(size) - fixed length character data type.
VARCHAR(size) - variable length character data type that is of length between 0 and the specified ’size’ parameter.
INT(size) - integer field that is between 0 and the specified ’size’ parameter. Note: ’size’ parameter can be non-specified and thus leads to no constraint on the data being inserted into that field except that it is of the INT datatype.
BOOL or BOOLEAN - True/False value. Note: zeros are false and any nonzero is true. Note: unlike other languages the capitalizing of specific characters within ’True’ and ’False’ values does not matter. For example, BOOLEAN can take in values such as, ’True’, ’TRUE’, or ’true’.
DOUBLE(size, d) - The size parameter specifies the number of digits allowed and the ’d’ parameter specifies how many digits are allowed after the decimal i.e. DOUBLE(5, 2) would allow values 103.52, 106.89, 999.99, etc. but NOT 98.1, 9.09, 100, or 34567.
FLOAT(p) - this datatype refers to floating point values which you may have come across in previous courses. The ’p’ parameter specifies whether or not the datatype of the float will remain a float or will become a double. If ’p’ is from 0 - 24 then the datatype remains a FLOAT() datatype and if ’p’ is in the range 25 - 53 then the datatype becomes a DOUBLE() datatype

2.3.1 Example: Creating a Dog Table

```
CREATE TABLE Dog Table(
Gender CHAR(1),
Breed VARCHAR(20),
Age INT(25),
Name VARCHAR(15),
);
```

Appropriate values might look like:
Gender - ’M’ or ’F’ or ’U’ (for UNKNOWN)
Breed - ’Pomeranian’, ’Labrador’, ’Retriever’, etc.
Age - 1, 2, 3, ..., 25
Name - ’Charlie’, ’Buddy’, ’Max’, etc.

Inappropriate values would look like:
Gender - ’Male’, ’Female’, ’Fem’ - these values would violate the restriction for the length of the value to be of at most length 1
Breed - ’Cutest Labrador retriever you ever saw’ - while the owner may think this, their entry would be invalid as it would not meet the <=20 length criteria.

Age - 26, 27,.., 10000 - these values would be invalid as the entry must be <= 25

2.4 Drop Clause, Primary Keys, & Foreign Keys

The concept of dropping a table using the DROP TABLE clause in SQL is fairly straightforward. Whenever we want to delete a table and all of that table's information that is stored in our database we can use the 'DROP TABLE' clause.

Example: Dropping our Dog Table from before looks like this:

```
DROP TABLE Dog Table;
```

Now Dog Table and all of the dog information that Dog Table held within our database has been deleted.

2.4.1 Primary Keys

Definition - Recall that a primary key is a field attribute of a SQL table that has unique values and thus denotes the 'granularity' of the table.

Example: Office Table
Say we want to create a table about the characters from our favorite TV sitcom, 'The Office'. We might set up the table (or as you might hear "set up the 'schema' of the table") to have one row per character and thus create a unique value denoting each individual character:

```
CREATE TABLE Office_Characters (
INT character_id,
First Name VARCHAR(20),
Last Name VARCHAR(30),
Age INT(99),
PRIMARY KEY (employee_id)
);
```

In our "Office_Characters" table, we can see that the table has one character for each row or we can say equivalently, that the table has an character-level of granularity because the PRIMARY KEY of the table is the employee id.

Note: You may be thinking "Why did we make a 'character_id' field when we could have just used, say, 'First Name'. In specific cases, this may be fine *if and only if* you can ensure that each character has a unique 'First Name'. For example, if there are two characters named 'Angela' then using 'First Name' as a PRIMARY KEY would not work.

2.4.2 Foreign Keys

Example: Office Database
Say we have a relational database management system about the TV sitcom, 'The Office'. In this RDBMS, we might have a 'Characters' table, an 'Actors' table, and 'Managers' table.
These tables may look like:

Building a Characters Table:

```
CREATE TABLE Office_Characters (
character_id INT,
First Name VARCHAR(20),
Last Name VARCHAR(30),
Age INT(99),
PRIMARY KEY (employee_id) );
```

Building a Managers Table:

```
CREATE TABLE Office_Managers(
manager_id INT,
Branch Name VARCHAR(20),
FOREIGN KEY (manager_id)
);
```

In this example, our database is composed of two tables, one of which references the other. The Office_Characters table we are already familiar with from the previous section, but now we have introduced a new table which references the Office_Characters table, the Office_Managers Table.

The Office_Managers table is on the manager-level of granularity meaning that each row of the table denotes a single manager and each row has a unique value for each of the managers, the manager_id.

The manager_id field is a FOREIGN KEY which references the character_id field from the Office_Characters table and thus we know the manager_id is simply the character_id of that manager.

2.5 Where Clause

The WHERE clause adds conditionals to SQL. Using the WHERE clause, we can filter our tables and select the entries that meet the specified conditions.

```
SELECT columnNameA, columnNameB, ...
FROM tableName
WHERE condition;
```

A Condition in a WHERE clause is composed of operators. These are the operators that can be used: =, >, <, <=, >=, <>, BETWEEN, LIKE, IN

For example, we have the table Finals, which has the columns: ”name”, ”SSID”, and ”grade”. If we want to select the names of all students who got an A (90 or above) on the final exam, we would use this statement:

```
SELECT name
FROM Finals
WHERE grade >= 90;
```

2.6 String Functions

SQL has predefined functions that can be used to perform operations on strings input, which return an output string. The functions are used in conjunction with clauses, such as SELECT and INSERT INTO.

There are a wide variety of String Functions, which we will not go over now, but it is a good idea to remember they exist for when they come in handy.

Example:
To find the length of a string, the $char_length$ function is used.

```
    SELECT char_length(”Oski”)
    Output: 4
```

2.7 Aggregation: ORDER BY and GROUP BY Clauses

2.7.1 Group By

In the previous section we saw how to use WHERE clauses to select down our data using a SQL Query. In this section we will see how we can use the aggregate clause, Group By, along with aggregation functions such as SUM, AVG, MIN, MAX, and COUNT to write more sophisticated queries. Before getting into how aggregation queries work, we want to introduce the order of commands in a typical aggregation queries. The typical chain of commands looks like:

SELECT...
FROM...
WHERE...
GROUP BY...
HAVING...
ORDER BY...

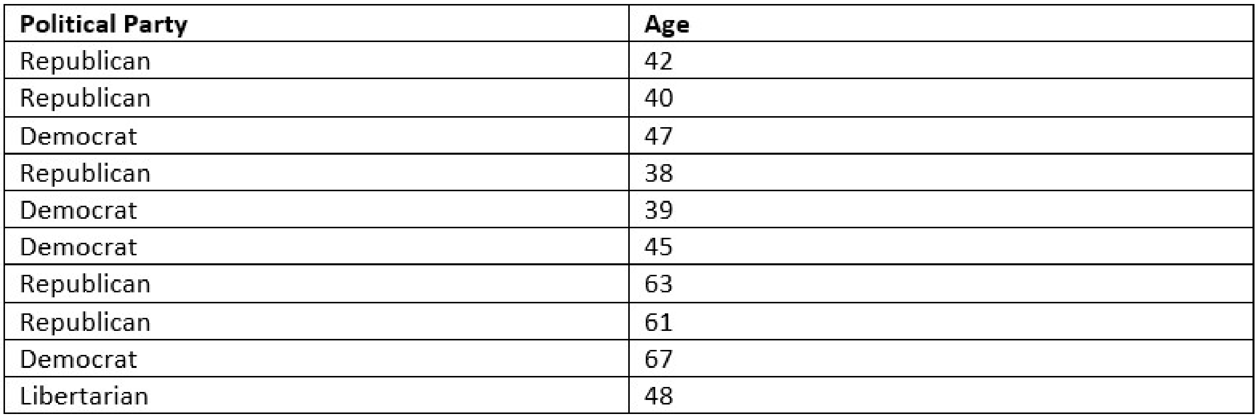
How Aggregation Works
Aggregation using the Group By clause works the same as .groupby() in Pandas if you are familiar with that. If you are not familiar, here is a pictorial description of the steps of how the Group By clause works and then how the subsequent aggregation functions react to the result of the Group By clause.

COUNT example
Step 1: We begin with a simple table

```
CREATE TABLE Politicians(
Political Party VARCHAR(25),
Age INT(99),
);
```

Note: you can ignore this part as it will be discussed in the following section

```
INSERT INTO Politicians (
Political Party, Age)
VALUES
(Republican, 38),
(Democrat, 39),
(Republican, 40),
(Republican, 42),
(Democrat, 45),
(Democrat, 47),
(Libertarian, 48),
(Republican, 61),
(Republican, 63),
(Democrat, 67);
```



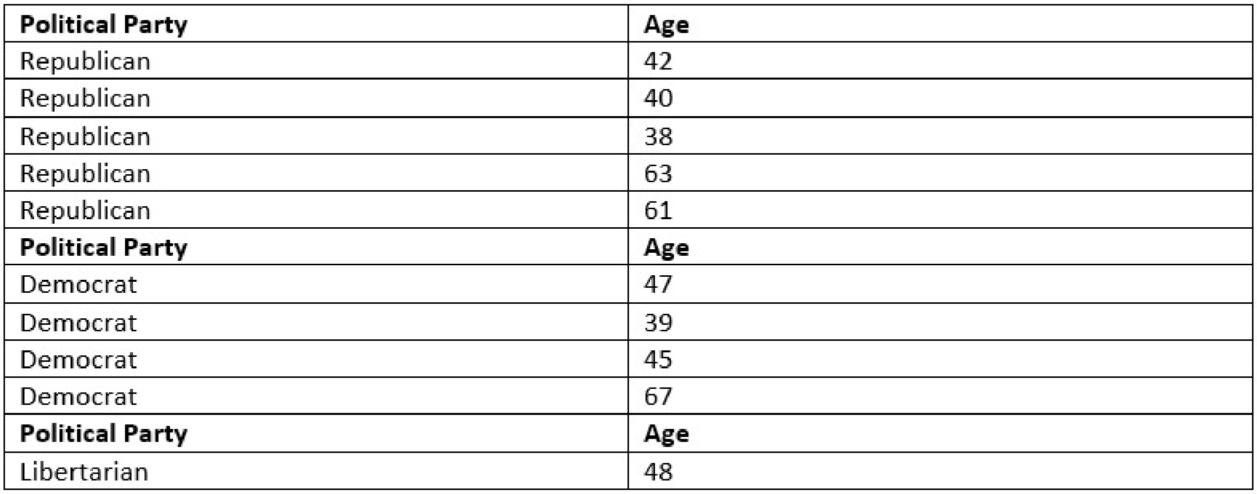
Political Party	Age
Republican	42
Republican	40
Democrat	47
Republican	38
Democrat	39
Democrat	45
Republican	63
Republican	61
Democrat	67
Libertarian	48

Note: this example table does not have a unique identifier/Primary Key, but in practice every SQL Table MUST have a Primary Key

Step 2: We enact the following query on our Politicians table
SELECT Political Party, COUNT(*)
FROM Politicians
GROUP BY Political Party;

Step 3: Assessing the GROUP BY clause
The intermediary step before the aggregation function, COUNT, takes place, is commonly referred to as the creation of a GroupBy Object. While GroupBy objects are specific to Pandas, they also have a placehold in SQL GroupBy as well because the table goes through the same sort of process.

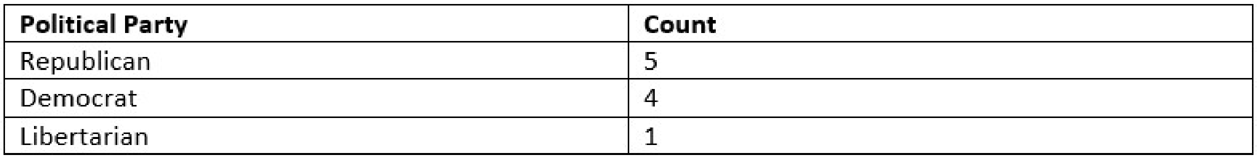
Result of the GROUP BY clause



Political Party	Age
Republican	42
Republican	40
Republican	38
Republican	63
Republican	61
Political Party	**Age**
Democrat	47
Democrat	39
Democrat	45
Democrat	67
Political Party	**Age**
Libertarian	48

Step 4: Assessing the Aggregation Function

The COUNT(*) will be enacted on each of these 3 tables and thus will count how many rows pertain to each unique value in the Political Party field. The result:



Political Party	Count
Republican	5
Democrat	4
Libertarian	1

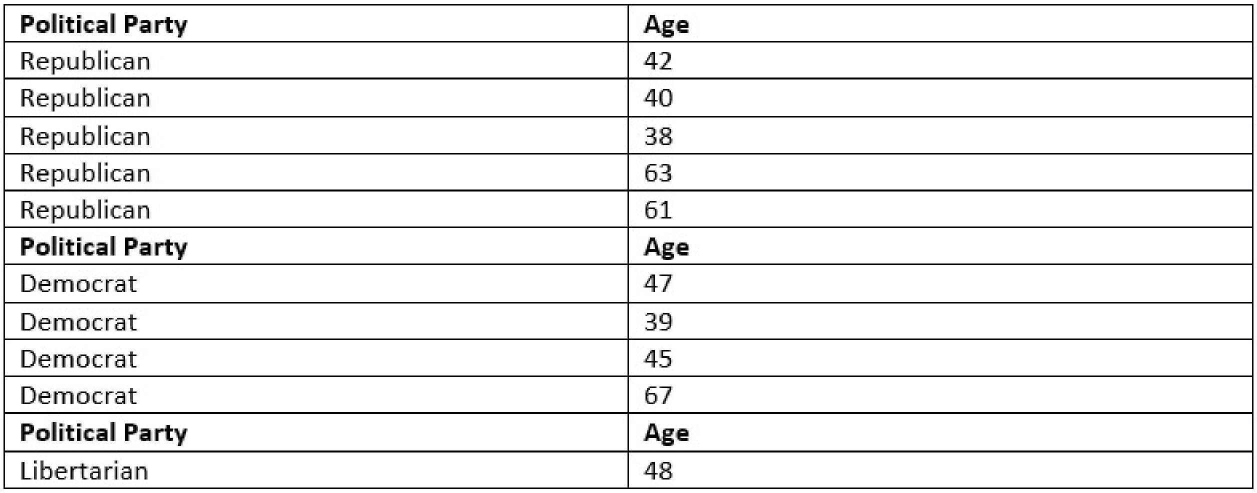
Note: we include the Political Party in the SELECT clause to see which counts pertain to which Political Party, otherwise the query would only return the 'Count' column

AVG example (using the same Politicians Table from the COUNT example)
Step 1: We enact the following query on our Politicians table

SELECT Political Party, AVG(Age)
FROM Politicians
GROUP BY Political Party;

Step 2: Assessing the GROUP BY clause

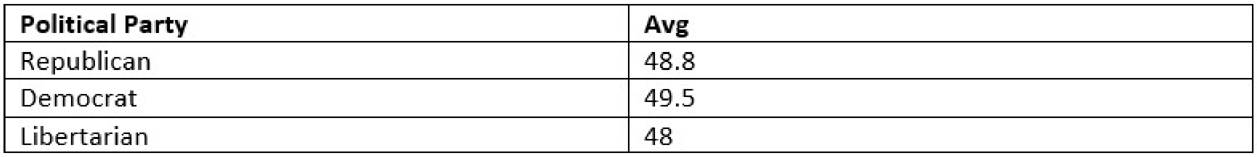
Result of the GROUP BY clause



Political Party	Age
Republican	42
Republican	40
Republican	38
Republican	63
Republican	61
Political Party	**Age**
Democrat	47
Democrat	39
Democrat	45
Democrat	67
Political Party	**Age**
Libertarian	48

Step 3: Assessing the Aggregation Function

The AVG(Age) will be enacted on each of these 3 tables and thus will return the average age for each of these tables:



Political Party	Avg
Republican	48.8
Democrat	49.5
Libertarian	48

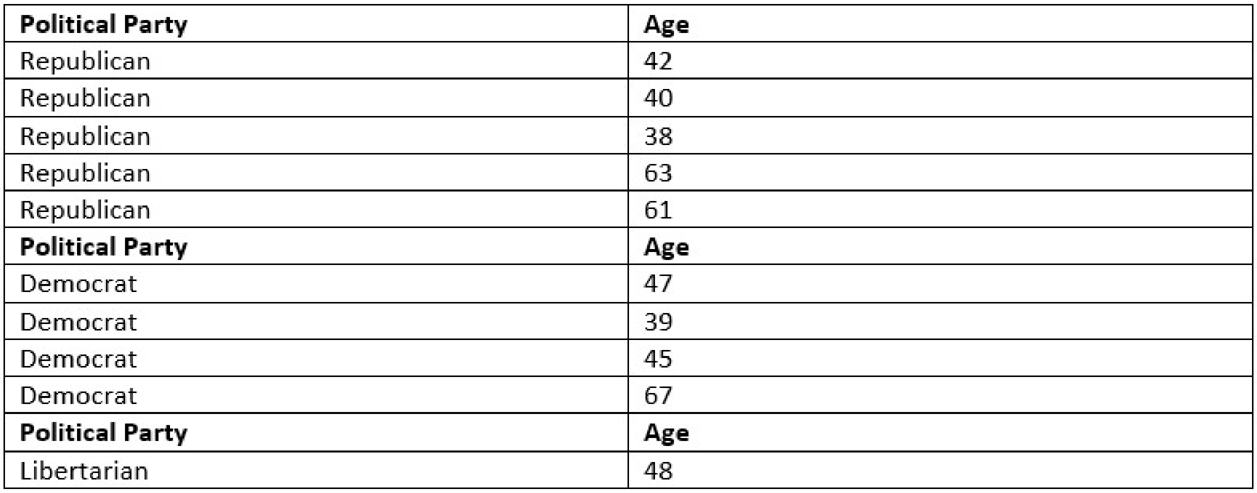
Aggregation Coupled w/ WHERE Clause Example
Step 1: We enact the following query on our Politicians table

SELECT Political Party, COUNT(*)

```
FROM Politicians
WHERE Age >= 60
GROUP BY Political Party;
```

Step 2: Assessing the GROUP BY clause

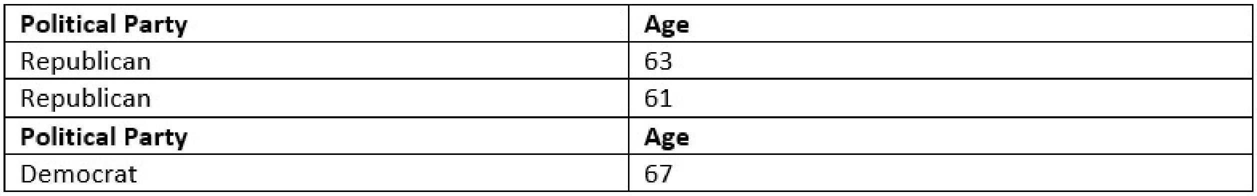
Result of the GROUP BY clause



Political Party	Age
Republican	42
Republican	40
Republican	38
Republican	63
Republican	61
Political Party	**Age**
Democrat	47
Democrat	39
Democrat	45
Democrat	67
Political Party	**Age**
Libertarian	48

Step 3: Assessing the WHERE Clause

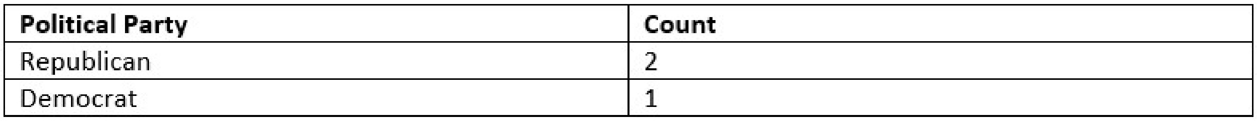
The WHERE Age >= 60 will be enacted on each of these 3 tables and thus will return the average age for each of these tables:



Political Party	Age
Republican	63
Republican	61
Political Party	**Age**
Democrat	67

Step 4: Final Aggregation

The COUNT(*) function gets applied to the two tables from Step 3 resulting in the following outcome:



Political Party	Count
Republican	2
Democrat	1

Note: The column that the aggregation function gets enacted on gets changed as a result of the aggregation. For example, in Example 1 we see that a 'Count' column has been created and in Example 2 an 'Avg' column has been created.

2.7.2 Order By

How Does Order By Work?

The column on which the ORDER BY clause is enacted on, is ordered, by default, in ascending order. The

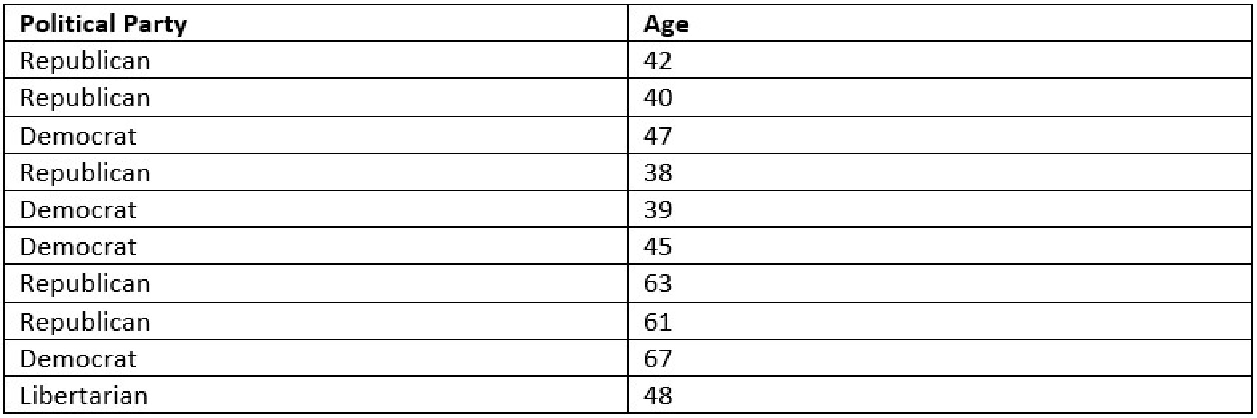
ASC/DESC clause following the ORDER BY clause specifies whether the user wants to order the column in ascending or descending order.

Examples (using our Politicians Table)

Query 1

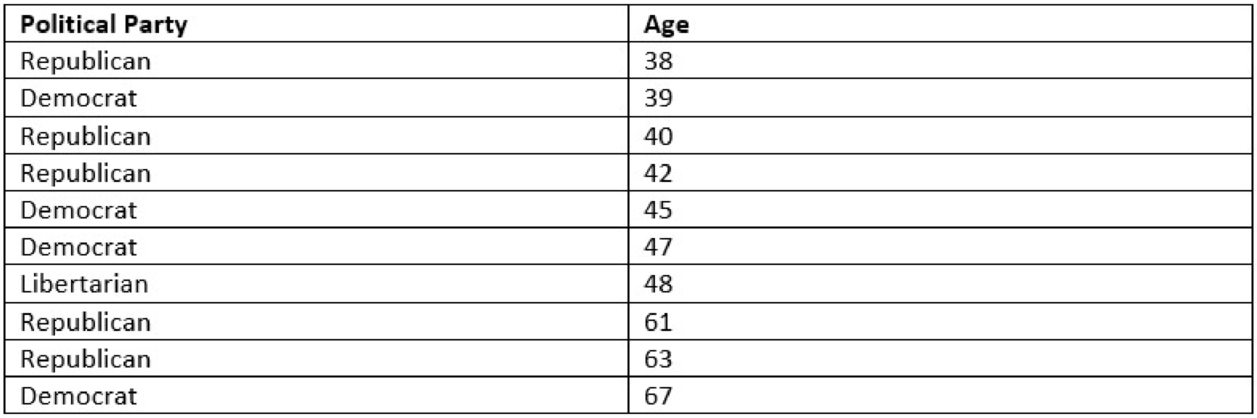
```
SELECT *
FROM Politicians
ORDER BY Age;
```

Recall from the previous section our table



Political Party	Age
Republican	42
Republican	40
Democrat	47
Republican	38
Democrat	39
Democrat	45
Republican	63
Republican	61
Democrat	67
Libertarian	48

The Result of the ORDER BY Clause



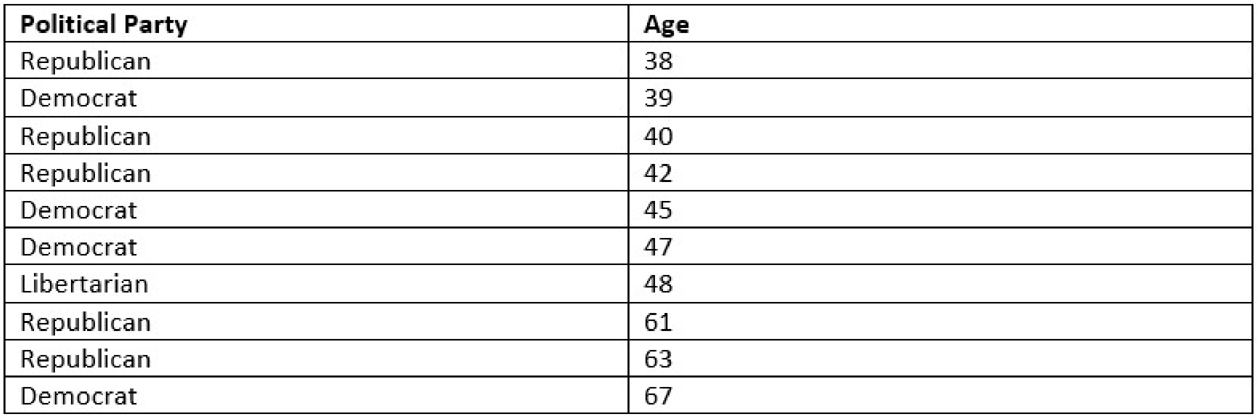
Political Party	Age
Republican	38
Democrat	39
Republican	40
Republican	42
Democrat	45
Democrat	47
Libertarian	48
Republican	61
Republican	63
Democrat	67

This is the end result as we used the SELECT* clause

Query 2

```
SELECT *,
FROM Politicians,
WHERE Political Party = 'Republican',
ORDER BY Age;
```

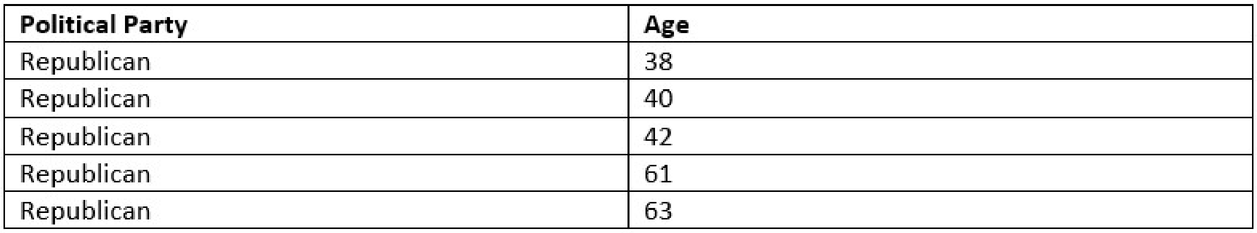
We assess the ORDER BY clause first



Political Party	Age
Republican	38
Democrat	39
Republican	40
Republican	42
Democrat	45
Democrat	47
Libertarian	48
Republican	61
Republican	63
Democrat	67

We finally assess the WHERE clause

In the where clause, we select down our ordered data to only include the Republican politicians.

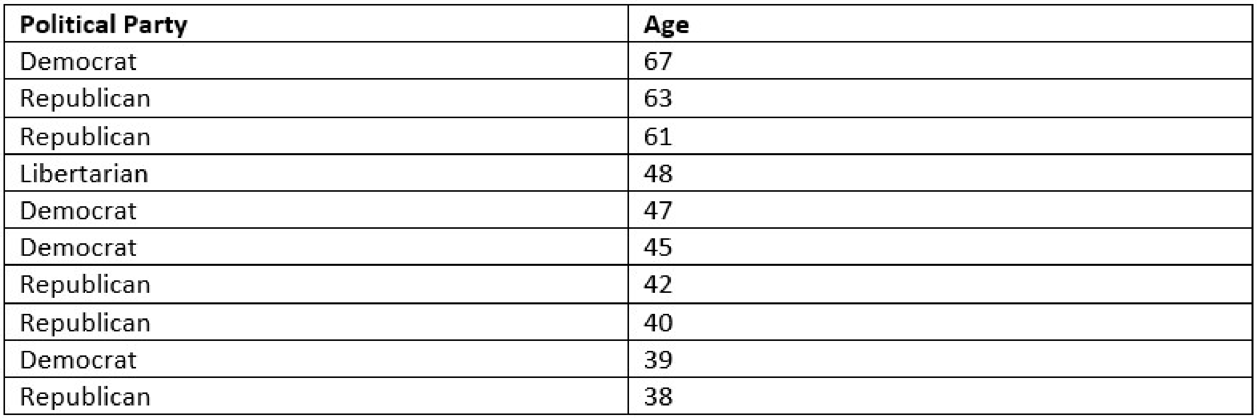


Political Party	Age
Republican	38
Republican	40
Republican	42
Republican	61
Republican	63

Query 3

SELECT Political Party, AVG(age)
FROM Politicians
GROUP BY Political Party
ORDER BY Age DESC;

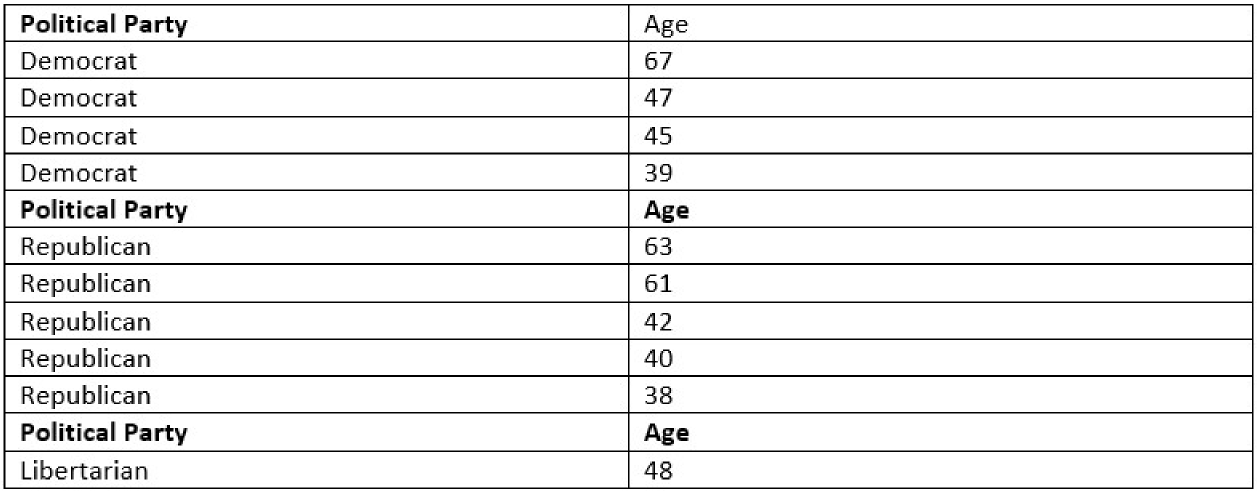
We assess the ORDER BY clause first



Political Party	Age
Democrat	67
Republican	63
Republican	61
Libertarian	48
Democrat	47
Democrat	45
Republican	42
Republican	40
Democrat	39
Republican	38

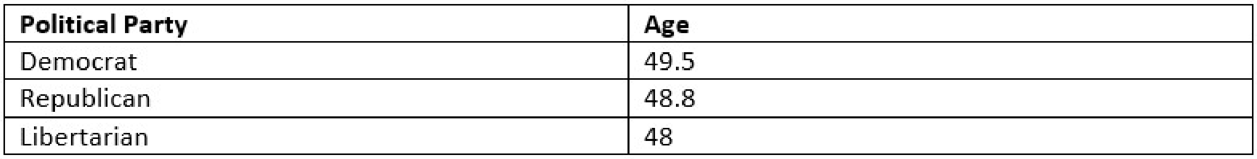
Note: we used the DESC clause in this query

Next we look at the GROUP BY clause



Political Party	Age
Democrat	67
Democrat	47
Democrat	45
Democrat	39
Political Party	**Age**
Republican	63
Republican	61
Republican	42
Republican	40
Republican	38
Political Party	**Age**
Libertarian	48

Finally, we average over each of the three tables



Political Party	**Age**
Democrat	49.5
Republican	48.8
Libertarian	48

2.7.3 HAVING Clause

The HAVING clause functions the same as the WHERE clause, but is **ONLY** enacted on the column which is being aggregated whereas the WHERE clause can be enacted on any of the fields.

Having Example

Say we wanted to find the average age of only the republican politicians. We could do that by writing a query with a WHERE clause such as this:

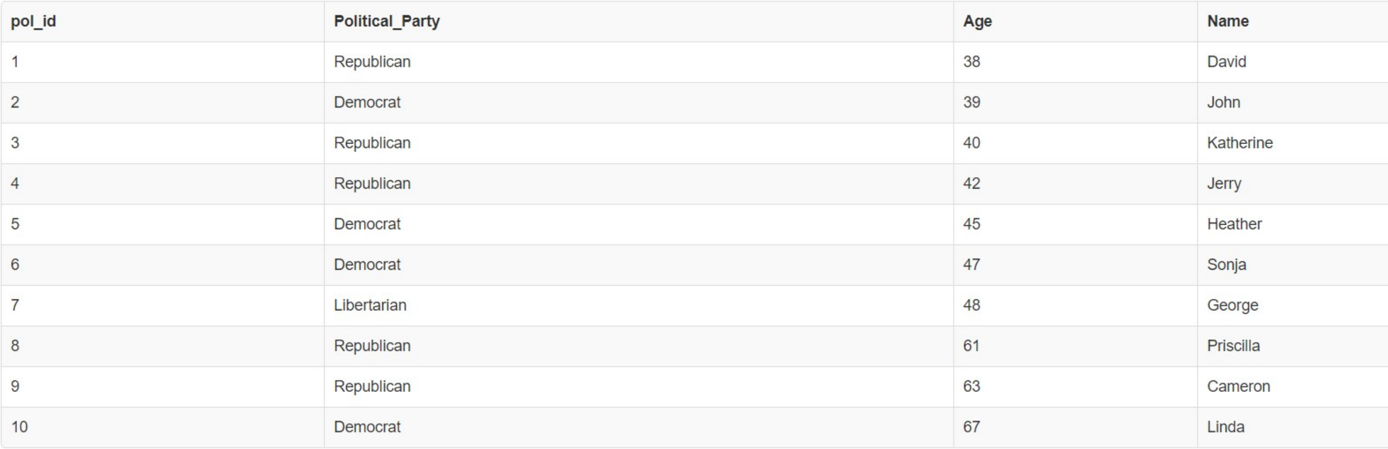
```
SELECT Political Party, AVG(Age)
FROM Politicians
WHERE Political Party = ”Republican”;
```

We could also get the same result writing a query such as this:

```
SELECT Political Party, AVG(Age)
FROM Politicians
GROUP BY Political Party
HAVING Political Party = ’Republican’;
```

Finally, we will end this section with a trickier SQL Puzzle.
Say we have an updated version of our Politicians table with the following schema:

```
CREATE TABLE Politicians(
pol_id INT,
Political_Party VARCHAR(25),
Age INT(99),
Name VARCHAR(10),
PRIMARY KEY(pol_id)
);
INSERT INTO Politicians values
(00001,”Republican”, 38, ”David”),
(00002,”Democrat”, 39, ”John”),
(00003,”Republican”, 40, ”Katherine”),
(00004,”Republican”, 42, ”Jerry”),
(00005,”Democrat”, 45, ”Heather”),
(00006,”Democrat”, 47, ”Sonja”),
(00007,”Libertarian”, 48, ”George”),
(00008,”Republican”, 61, ”Priscilla”),
(00009,”Republican”, 63, ”Cameron”),
(00010,”Democrat”, 67, ”Linda”);
```



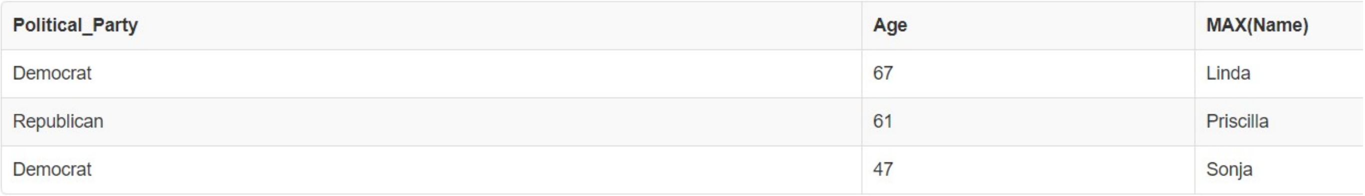
pol_id	Political_Party	Age	Name
1	Republican	38	David
2	Democrat	39	John
3	Republican	40	Katherine
4	Republican	42	Jerry
5	Democrat	45	Heather
6	Democrat	47	Sonja
7	Libertarian	48	George
8	Republican	61	Priscilla
9	Republican	63	Cameron
10	Democrat	67	Linda

Determine the outcome of the following query:

```
SELECT Political_Party, Age, MAX(Name)
FROM Politicians
WHERE Age >= 40
GROUP BY Name
HAVING Name LIKE “%a“
ORDER BY Name;
```

Note: If you are not familiar with text commands such as LIKE in SQL, they are discussed later in this Note. For the sake of the example, the LIKE “%a“ clause will get all of the Names ending in ’a’. Another thing to note is that the MAX and MIN aggregation functions work with string datatypes, but others such as SUM and AVG **DO NOT**. MAX(Name) will get the Name that is last alphabetically.

Solution:



Political_Party	Age	MAX(Name)
Democrat	67	Linda
Republican	61	Priscilla
Democrat	47	Sonja

3 Modifying Tables

3.1 Insert

In order to add records to a table in SQL, we must use the Insert clause. A general structure is given below.

INSERT INTO [*Table_name*] (*column*1, *column*2, ...) VALUES (*value*1, *value*2, ...), (*value*1, ...);

Another way of writing the above query is demonstrated below.

INSERT INTO [*Table_name*] (*column*1, *column*2, ...)
VALUES
(*value*1, *value*2, ...),
(*value*1, *value*2, ...);

For example if we wanted to add Hannah’s female Pomeranian named Lucy that was 8 years old we could write the following SQL Query:

INSERT INTO dogs (Gender, Breed, Age, Name)
VALUES (’F’, ’Pomeranian’, 8, ’Lucy’);

Note: the Query is followed by a semicolon always

As explained earlier in the Datatypes section, A query that would not work for inserting a dog into this table could be something that doesn’t follow the ’schema’ of the table:

INSERT INTO dogs (Gender, Breed, Age, Name)
VALUES (8, 9, 10, 11);

Clearly not all of these datatypes in the query are the datatypes outlined in the table created above. SQL would not accept the bottom as a valid query due to the specified datatypes in the created table.
Insert can be a powerful tool to use to add in sample data into an already existing table.

3.2 Update

If you wish to update the attributes of a particular record, we will use the Update clause. A general structure is given below.

UPDATE [*Table_name*]
SET *column*1 = *value*1, *column*2 = *value*2, ...
WHERE *condition*;

As an example, let’s say that it was Hannah’s dog’s birthday and we wanted to update Lucy’s age from 8 years to 9 years old, we could write the following SQL Query:

```
UPDATE dogs
SET Age = 9
WHERE Name='Lucy' AND Breed='Pomeranian';
```

3.3 Delete

If you wish to delete rows of a particular table, we will use the Delete clause. A general structure is given below.

```
DELETE FROM [Table_name]
WHERE condition;
```

For example, Hannah decided that she doesn't want to use our service anymore as we accidentally caused the dog to develop a rare disease, so to delete her dog's information, we could write the following SQL Query:

```
DELETE FROM dogs
WHERE Name='Lucy' AND Breed='Pomeranian';
```

4 Constraints

4.1 Not NULL

The NOT NULL constraint ensures that a column of a table will not accept a NULL value for a field.

Looking back at the Politicians table, we want to specify that Name is never a NULL value.

```
CREATE TABLE Politicians(
pol_id INT,
Political_Party VARCHAR(25),
Age INT(99),
Name VARCHAR(10) NOT NULL,
PRIMARY KEY (pol_id)
);
```

Note: pol_id is already specified as a non-NULL value because it is a primary key.

4.2 Check

The CHECK constraint specifies a particular range of values that a column can hold. This constraint can be used on specific columns or on the table as a whole.

We want to specify a constraint on the Politicians table that checks the age of the representatives to be over 25 years old.

```
CREATE TABLE Politicians(
pol_id INT,
Political_Party VARCHAR(25),
Age INT(99),
Name VARCHAR(10),
PRIMARY KEY(pol_id),
CHECK (Age≥25) );
```

4.3 Default

The DEFAULT constraint sets a default value for a particular column if no value is specified.

We noticed that some politicians are being entered into the system without a political party. We want to set the default political party to be Independent.

```
CREATE TABLE Politicians(
pol_id INT,
Political_Party VARCHAR(25) DEFAULT 'Independent',
Age INT(99),
Name VARCHAR(10),
PRIMARY KEY(pol_id),
CHECK (Age>=25) );
```

5 Join Variants, Aliases, & the DISTINCT Clause

In industry, you will more often than not be asked to join multiple tables together. Joining tables can help to find similarities between two subjects, find differences between two subjects, or easily allow you to get multiple tables on the same level of granularity. In this section we will be going over 4 different types of joins: Inner and Natural Joins, Left Joins, and Right Joins.

5.0.1 Inner Join

Inner joins are accompanied by a predicate or a condition on which to join by. When that predicate is met, the row from both tables are joined together. Note that for all joins the two tables must either 1) share a column with the same name or 2) share a column that represents the same thing, but may carry a table-specific name. An example of this might be if you have a table of Baseball Players and Baseball Pitchers. The Players table may have a PRIMARY KEY such as, "player_id" and the Pitchers table may have a FOREIGN KEY referencing the Players table such as "pitcher_id". Because these two fields represent the same information and are of the same datatype you may join on these fields. Below is the typical structure of an INNER JOIN:

```
SELECT [column1, column2, ...]
FROM Left Table
INNER JOIN Right Table
ON Predicate;
```

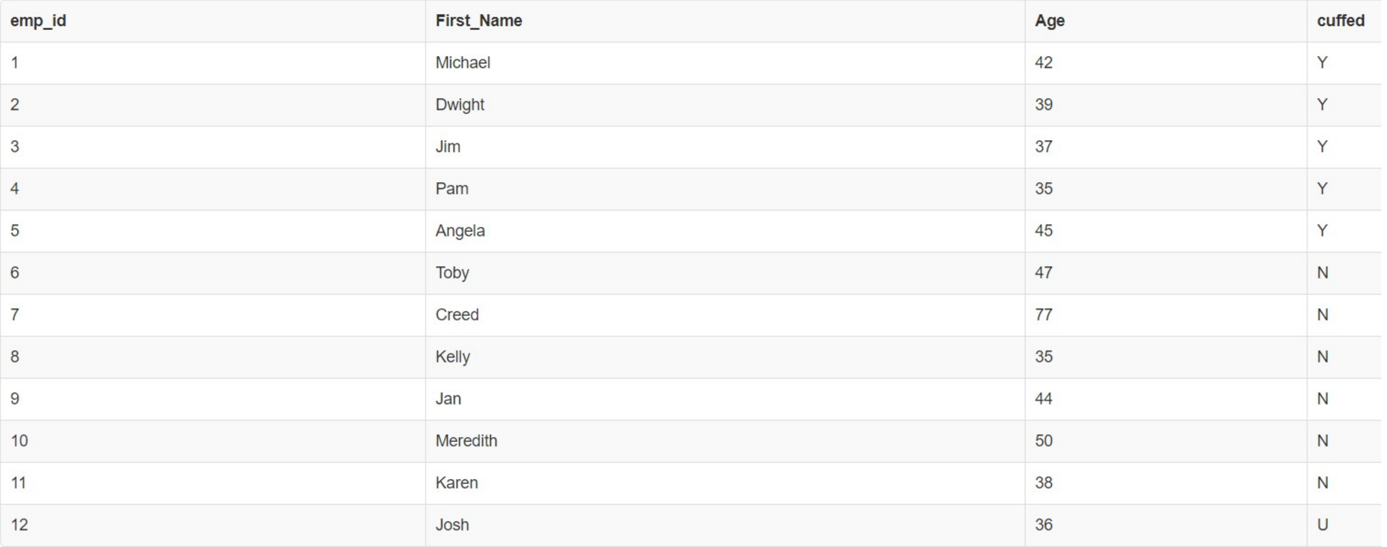
In this section, we will look at some made-up data (as we have been doing) regarding the characters of, "The Office", and their managers. The Office Table includes a list of characters, their unique employee ids, their age, and a field which denotes whether that character is "cuffed" or not ("U" = UNKNOWN). The Managers table provides a list of the managers, their unique manager ids, and the name of the branch they manage.

Here is the syntax to create these two tables along with what they look like:

```
CREATE TABLE Office(
emp_id INT,
First_Name VARCHAR(25),
Age INT(99),
cuffed CHAR(1),
PRIMARY KEY(emp_id)
);
```

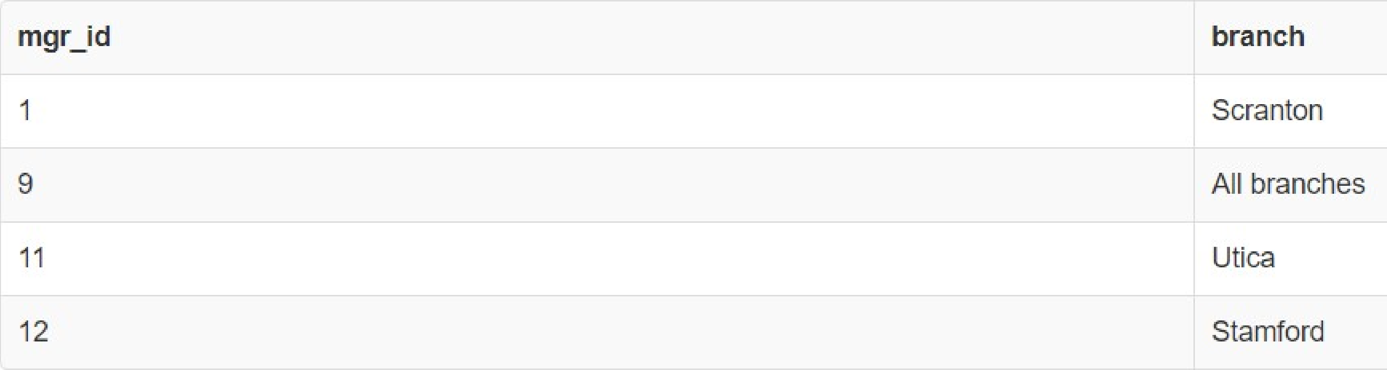
```
INSERT INTO Office VALUES
(01,”Michael”, 42, ”Y”),
(02,”Dwight”, 39, ”Y”),
(03,”Jim”, 37, ”Y”),
(04,”Pam”, 35, ”Y”),
(05,”Angela”, 45, ”Y”),
(06,”Toby”, 47, ”N”),
(07,”Creed”, 77, ”N”),
(08,”Kelly”, 35, ”N”),
(09,”Jan”, 44, ”N”),
(10,”Meredith”, 50, ”N”),
(11, ”Karen”, 38, ”N”),
(12, ”Josh”, 36, ”U”);

CREATE TABLE Managers(
mgr_id INT,
branch VARCHAR(20),
FOREIGN KEY (mgr_id) REFERENCES Office(emp_id)
);

INSERT INTO Managers VALUES
(01, ”Scranton”),
(09, ”All branches”),
(11, ”Utica”),
(12, ”Stamford”);
```

Office Table



emp_id	First_Name	Age	cuffed
1	Michael	42	Y
2	Dwight	39	Y
3	Jim	37	Y
4	Pam	35	Y
5	Angela	45	Y
6	Toby	47	N
7	Creed	77	N
8	Kelly	35	N
9	Jan	44	N
10	Meredith	50	N
11	Karen	38	N
12	Josh	36	U

Manager Table



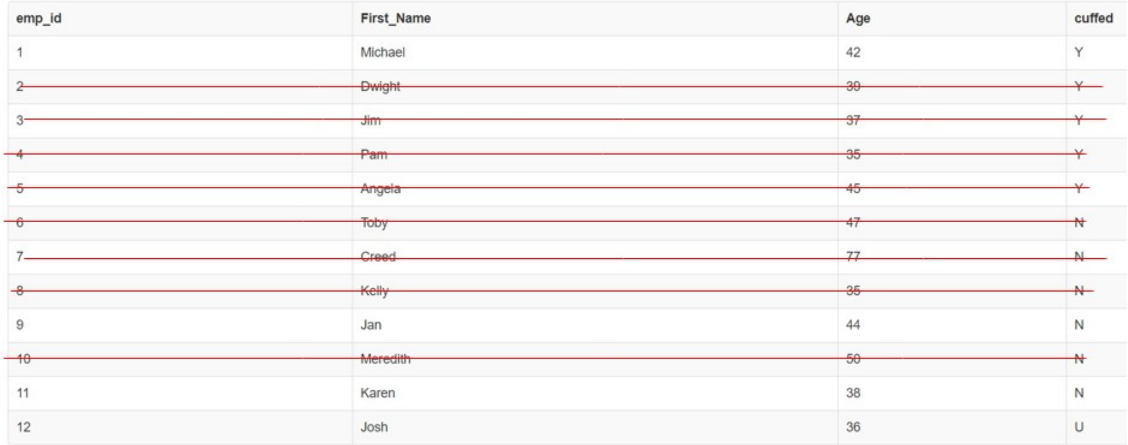
mgr_id	branch
1	Scranton
9	All branches
11	Utica
12	Stamford

Inner Join Example

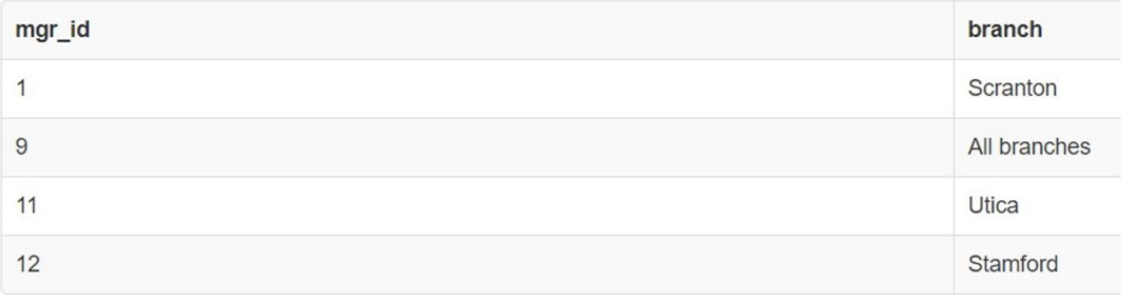
Say we wanted to get all of the manager's information. A query we could write to get that is by enacting an Inner Join on the two tables coupled with the predicate, emp_id = mgr_id. This would look like:

```
SELECT *
FROM Office
INNER JOIN Managers
ON (emp_id = mgr_id);
```

You can think of an Inner Join as moving through the left table and cross-referencing the value in the left column specified in the predicate with the values in the right column specified in the predicate. If those two values satisfy the predicate you join the two rows. Below is a depiction of this process:

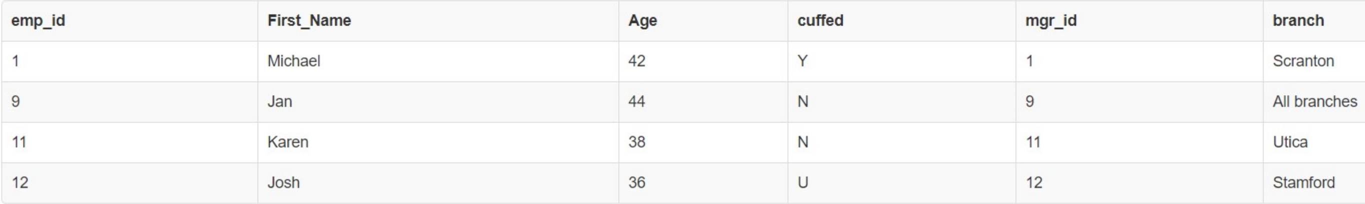


emp_id	First_Name	Age	cuffed
1	Michael	42	Y
~~2~~	~~Dwight~~	~~39~~	~~Y~~
~~3~~	~~Jim~~	~~37~~	~~Y~~
~~4~~	~~Pam~~	~~35~~	~~Y~~
~~5~~	~~Angela~~	~~45~~	~~Y~~
~~6~~	~~Toby~~	~~47~~	~~N~~
~~7~~	~~Creed~~	~~77~~	~~N~~
~~8~~	~~Kelly~~	~~35~~	~~N~~
9	Jan	44	N
~~10~~	~~Meredith~~	~~50~~	~~N~~
11	Karen	38	N
12	Josh	36	U



mgr_id	branch
1	Scranton
9	All branches
11	Utica
12	Stamford

Our result looks like:

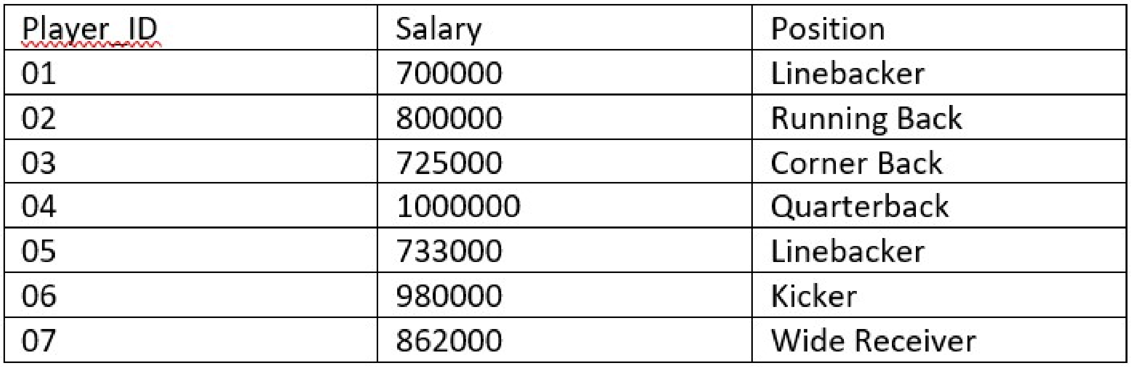


emp_id	First_Name	Age	cuffed	mgr_id	branch
1	Michael	42	Y	1	Scranton
9	Jan	44	N	9	All branches
11	Karen	38	N	11	Utica
12	Josh	36	U	12	Stamford

Note: this is what is known as an 'Equijoin'

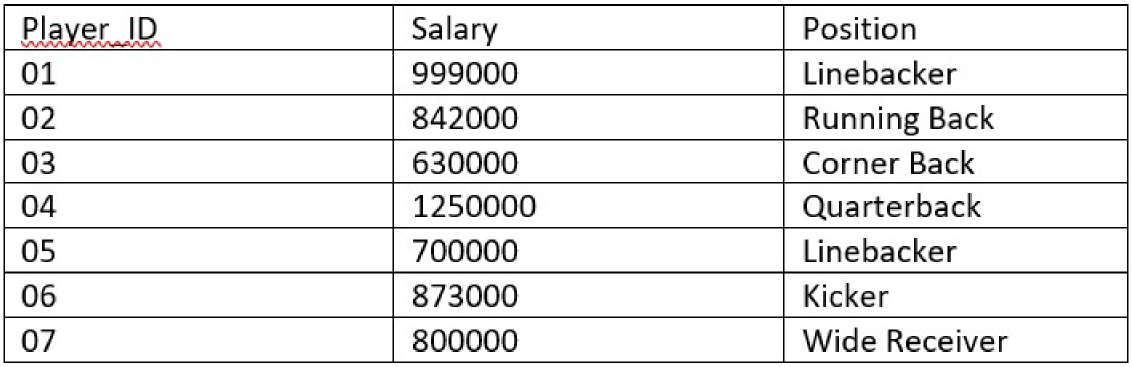
Not an equijoin Example:
Say we have the following two tables about the NFL Teams, The Patriots and The Cowboys:

A Table denoting the position and salary of the players on the Patriots



Player_ID	Salary	Position
01	700000	Linebacker
02	800000	Running Back
03	725000	Corner Back
04	1000000	Quarterback
05	733000	Linebacker
06	980000	Kicker
07	862000	Wide Receiver

A Table denoting the position and salary of the players on the Cowboys



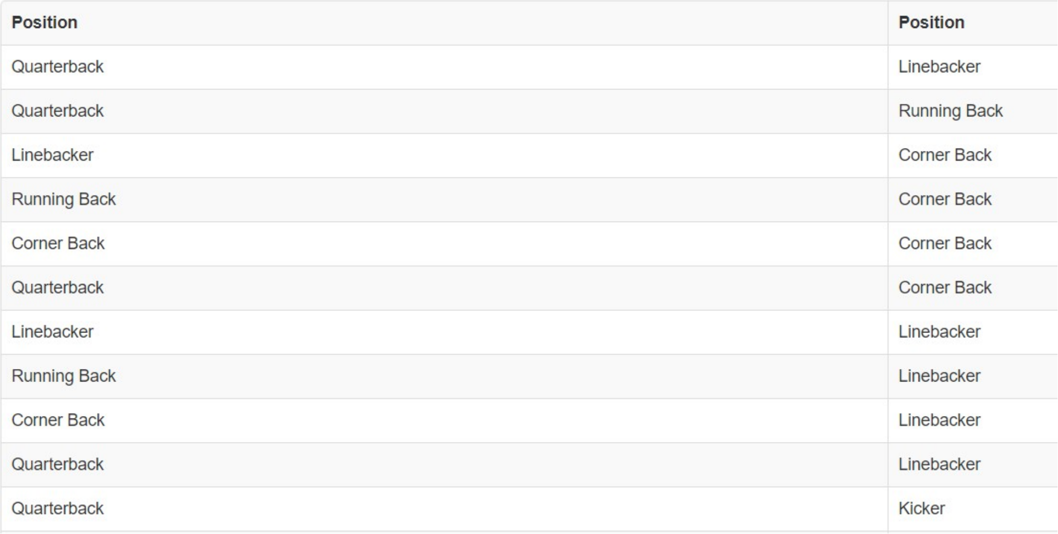
Player_ID	Salary	Position
01	999000	Linebacker
02	842000	Running Back
03	630000	Corner Back
04	1250000	Quarterback
05	700000	Linebacker
06	873000	Kicker
07	800000	Wide Receiver

SQL PUZZLE: Can we write a query such that we can see which positions on the Patriots make more money than positions on the Cowboys?

Query 1:

```
SELECT Patriots.Position, Cowboys.Position
FROM Patriots
INNER JOIN Cowboys
ON Patriots.Salary >= Cowboys.Salary;
```

This returns (see next page):



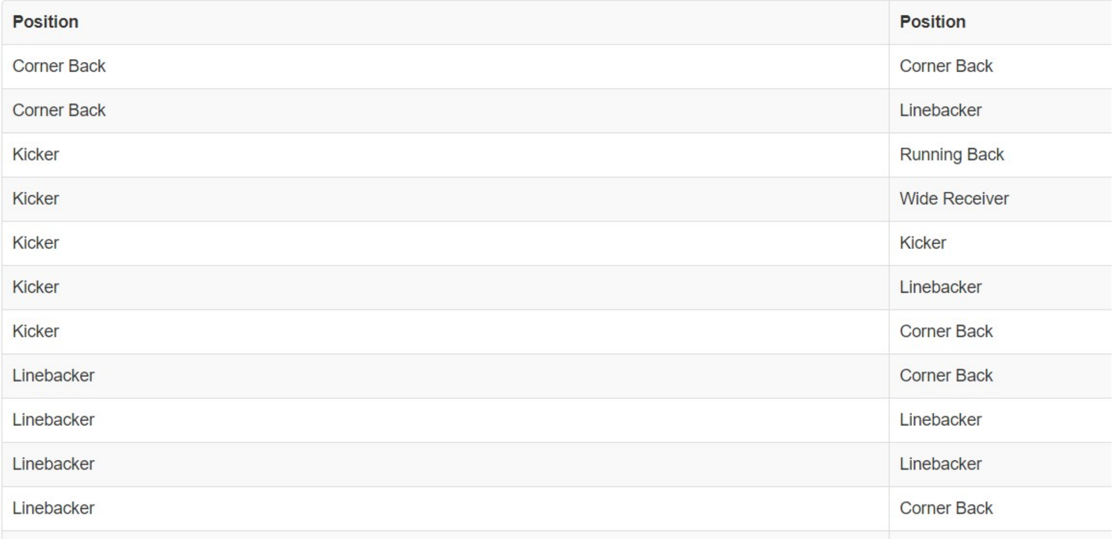
Position	Position
Quarterback	Linebacker
Quarterback	Running Back
Linebacker	Corner Back
Running Back	Corner Back
Corner Back	Corner Back
Quarterback	Corner Back
Linebacker	Linebacker
Running Back	Linebacker
Corner Back	Linebacker
Quarterback	Linebacker
Quarterback	Kicker

Do we notice anything about this result that we might want to change to make the message a little clearer? Notice that if our goal is to show which Patriot positions pay higher than those of the Cowboy's positions we might want to use our ORDER BY clause to organize the resulting table a little better.

Query 2:

```
SELECT Patriots.Position, Cowboys.Position
FROM Patriots
INNER JOIN Cowboys
ON Patriots.Salary >= Cowboys.Salary
ORDER BY Patriots.Position;
```

The resulting table looks like:



Position	Position
Corner Back	Corner Back
Corner Back	Linebacker
Kicker	Running Back
Kicker	Wide Receiver
Kicker	Kicker
Kicker	Linebacker
Kicker	Corner Back
Linebacker	Corner Back
Linebacker	Linebacker
Linebacker	Linebacker
Linebacker	Corner Back

Is there anything else that might stick out to us that we could explore improvement on? Notice how our Linebacker values in the left or Patriot Position column maps to two 'Linebacker' values and two 'Corner Back' values in the right column? To make our resulting data easier to follow we would want to have each individual Patriot position mapping to just 1 Cowboys position. That way when our audience reads the table they can see "Ok, the Linebacker on the Patriots makes more than the Linebacker and Corner Back on the Cowboys." The way we do this is by using the **DISTINCT** clause.

5.0.2 The DISTINCT Clause

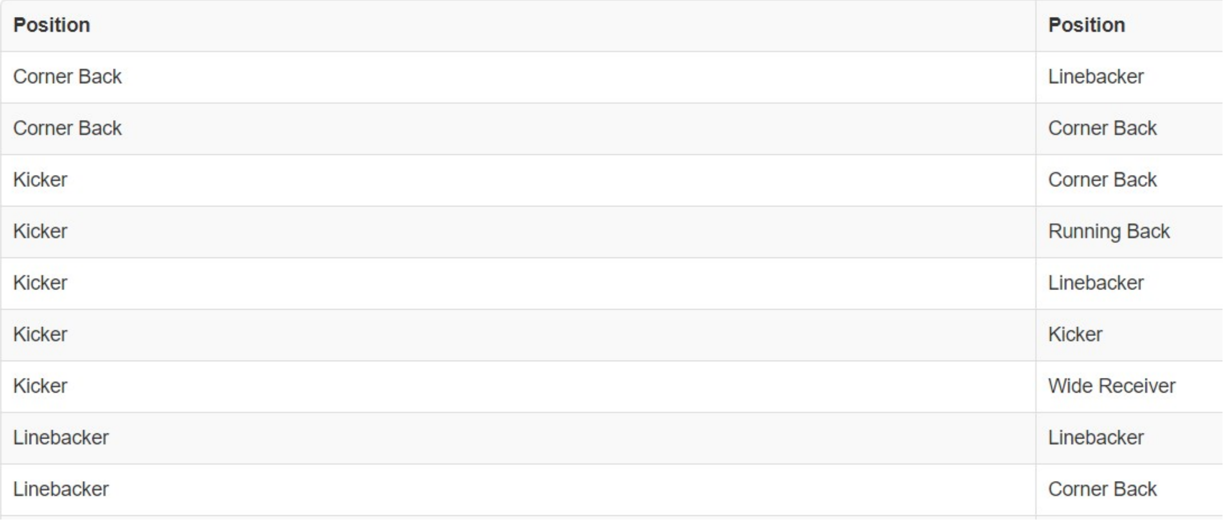
The DISTINCT Clause allows us to specify which column/s we want distinct/unique values from. In our example, we don't want repeating values for the Cowboys Position column and so this can come in handy. The typical structure of the DISTINCT clause is as so:

SELECT **DISTINCT** *column*1, *column*2, ...
FROM *tablename*;

We can now re-write our query with the DISTINCT clause like this:

Query 3:
SELECT Patriots.Position, **DISTINCT** Cowboys.Position
FROM Patriots
INNER JOIN Cowboys
ON Patriots.Salary >= Cowboys.Salary
ORDER BY Patriots.Position;

The resulting table looks like:



Position	Position
Corner Back	Linebacker
Corner Back	Corner Back
Kicker	Corner Back
Kicker	Running Back
Kicker	Linebacker
Kicker	Kicker
Kicker	Wide Receiver
Linebacker	Linebacker
Linebacker	Corner Back

As you can see the result from this is much easier to read. The Corner Back in the left or Patriot Position column has a higher salary than that of the Linebacker and the Corner Back on the Cowboys.

One final tweak to our result, is there anything else that might stick out to us that we could explore improvement on? As we can see our column names aren't particularly useful for understanding what is being presented in the table. One issue is that both columns are named 'Position'. Another issue is that we aren't showing what is being compared. These two issues would be harmful to an third party's understanding of the resulting table and what it represents should it be left unchanged. How can we fix this?

5.0.3 Aliases

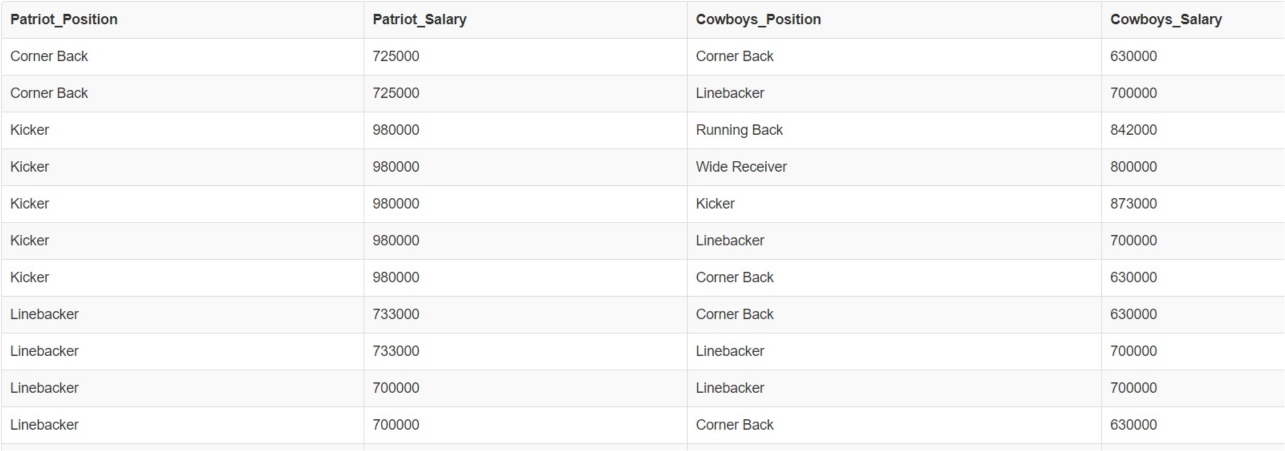
Aliases in SQL allow us to change the names of columns that, if they were not changed, may be confusing to a third party viewer such as in the above query. The typical structure of using Aliases in your SQL Query involves using the 'AS' clause in SQL and follows as so:

SELECT *column_name*1 AS *alias_name*1, *column_name*2 AS *alias_name*2, ...
FROM *table_name*;

Going back to our previous query, we now know that we can fix this issue by doing something like:

Final Query:
SELECT Patriots.Position **AS Patriot_Position**, Patriots.Salary **AS Patriot_Salary**, Cowboys.Position **AS Cowboys_Position**, Cowboys.Salary **AS Cowboys_Salary**
FROM Patriots
INNER JOIN Cowboys
ON Patriots.Salary >= Cowboys.Salary
ORDER BY Patriot_Position

The result of this query looks like (see next page):



Patriot_Position	Patriot_Salary	Cowboys_Position	Cowboys_Salary
Corner Back	725000	Corner Back	630000
Corner Back	725000	Linebacker	700000
Kicker	980000	Running Back	842000
Kicker	980000	Wide Receiver	800000
Kicker	980000	Kicker	873000
Kicker	980000	Linebacker	700000
Kicker	980000	Corner Back	630000
Linebacker	733000	Corner Back	630000
Linebacker	733000	Linebacker	700000
Linebacker	700000	Linebacker	700000
Linebacker	700000	Corner Back	630000

5.0.4 Natural Joins

This section will just be a straightforward explanation as Natural Joins are very similar to Inner Joins.

Natural Joins are the same as Inner Joins EXCEPT they lack a predicate. Natural Joins join the two tables on whichever column it finds in common between the two tables. The commonality between the two tables that the natural join looks for is strictly on a column name-basis so it only joins on two columns if they have the same name. **Note:** If two tables have multiple columns of the same name, the Natural Join aborts and returns 0 records.

5.0.5 Left & Right Joins

Left and Right Joins are commonly used as they act like inner joins, but favor either the Left Table or the Right Table being joined in the query. As you probably guessed, Left Joins favor the Left Table and Right Joins favor the Right Table.

Now what exactly do we mean by ”favor”. What we mean is that in a Left Join the join will always keep the left table’s records whether the predicate is met or not and will join right table records if the predicate is met and if it is not, will return a NULL value.

Structure of the Left/Right Join: The structure of the left/right join is the same as it was for inner joins. Therefore the left/right join follows as such:

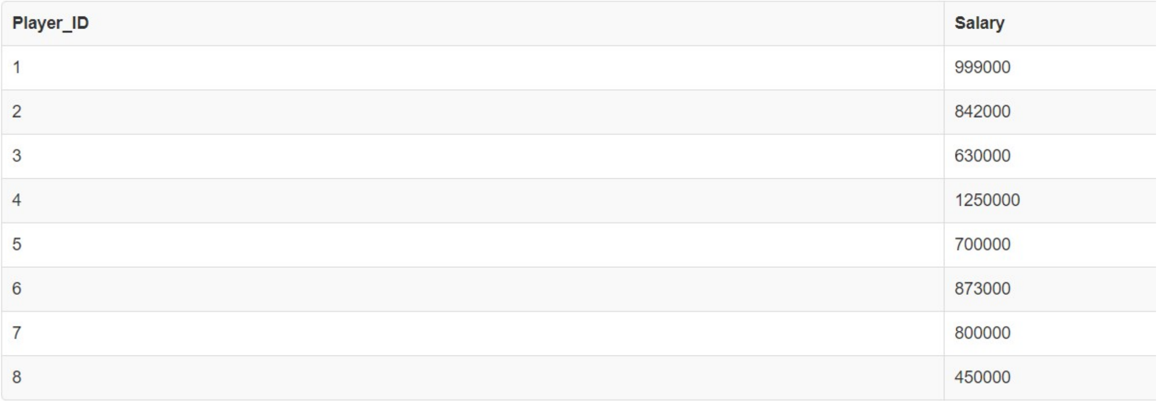
SELECT [*column*1, *column*2, ...]
FROM **Left Table**

LEFT/RIGHT JOIN **Right Table**
ON **Predicate**;

Left/Right Join Example:
Say for example we have two tables, one that has all of the salaries of every baseball player ever called 'Salary' and another that has the names and positions of every hall of fame legend baseball player, and we want to get the salaries of the hall of fame legends. This is where a Left or Right Join might come in handy.

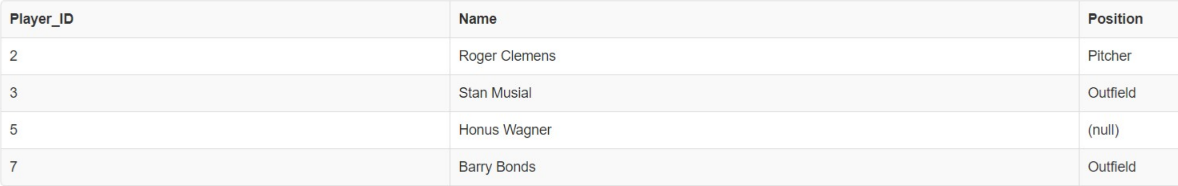
Lets take a look at these tables first:

Salary Table:



Player_ID	Salary
1	999000
2	842000
3	630000
4	1250000
5	700000
6	873000
7	800000
8	450000

Legends Table:



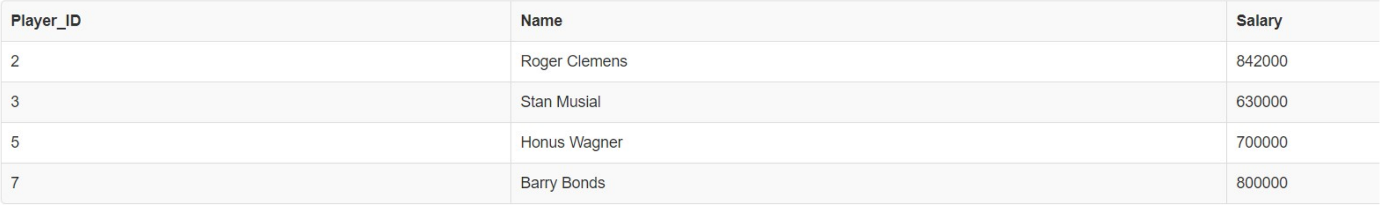
Player_ID	Name	Position
2	Roger Clemens	Pitcher
3	Stan Musial	Outfield
5	Honus Wagner	(null)
7	Barry Bonds	Outfield

In order to get the salaries of only the baseball players who are legends, we need to make sure our query keeps the Players in the Legends Table and grabs the salary values from the Salary table. How do we do this?

Solution: We can do this two ways. The first way is by using a Left Join. Since we are using a Left Join we need to ensure that the Legends Table is our Left Table in the query because that's the table we want to keep the records from. Next we find which column in the Salary table we could join on. This column is the Player_ID column. Thus our final query can look something like:

```
SELECT Legends.Player_ID, Name, Salary
FROM Legends
LEFT JOIN Salary
ON Salary.Player_ID = Legends.Player_ID;
```

The resulting table looks like:



Player_ID	Name	Salary
2	Roger Clemens	842000
3	Stan Musial	630000
5	Honus Wagner	700000
7	Barry Bonds	800000

Note that the following Right Join Query yields the same result:

```
SELECT Legends.Player_ID, Name, Salary
FROM Salary
RIGHT JOIN Legends
ON Salary.Player_ID = Legends.Player_ID;
```

Example 2: Notice in the previous example we could have used an Inner Equijoin on the two tables and gotten the same result. The equivalent Inner Join query looks like:

```
SELECT Legends.Player_ID, Name, Salary
FROM Legends
INNER JOIN Salary
ON Salary.Player_ID = Legends.Player_ID;
```

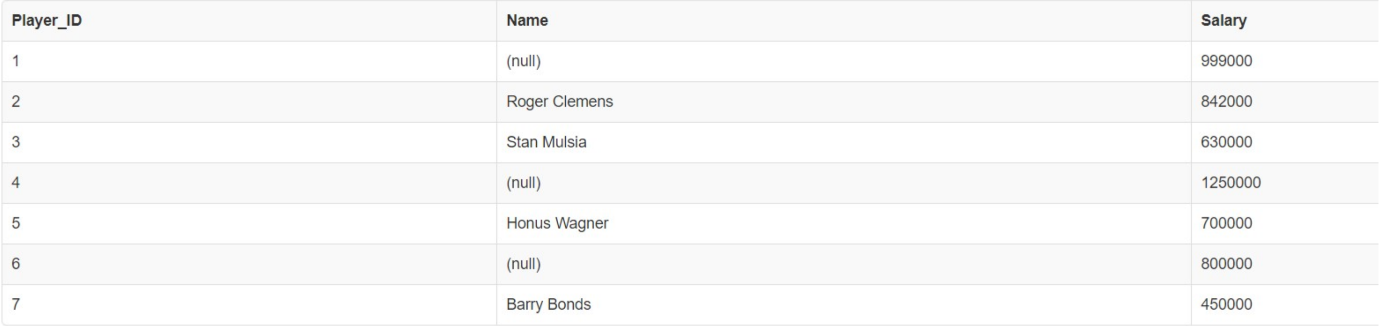
Question: What would be the result of writing a query like:

```
SELECT Salary.Player_ID, Name, Salary
FROM Salary
LEFT JOIN Legends
ON Salary.Player_ID = Legends.Player_ID;
```

i) How many records would be returned by this query?
ii) What does the returned table look like?

Solution to Question 1
The returned table will consist of 7 records because the Salary Table is the left table it will keep all of the Salary Table's records regardless of whether the predicate is met or not (Recall if the predicate is not met a NULL value is returned for the right table columns).

Solution to Question 2



Player_ID	Name	Salary
1	(null)	999000
2	Roger Clemens	842000
3	Stan Mulsia	630000
4	(null)	1250000
5	Honus Wagner	700000
6	(null)	800000
7	Barry Bonds	450000

6 Set Operations

Set operations are useful to extract data from two different tables. There are 4 types of SET operations:

1. UNION
2. UNION ALL
3. INTERSECT
4. MINUS

Each Set operation is composed of the result-set of two or more SELECT statements.

6.1 UNION

The UNION set operator combines the results of two or more SELECT statements. By default, the UNION operator filters out duplicates and only stores one copy of each entry.

In order to use the UNION operator, the results of the two SELECT statements must have the same number of columns, and the same datatype for each column.

6.2 UNION ALL

UNION ALL is identical to UNION, with the difference being that UNION ALL does include duplicate values between the two sets.

As with UNION, the results of the two SELECT statements must have the same number of columns, and the same datatype for each column.

6.3 INTERSECT

The INTERSECT set operator combines two or more SELECT statements, and returns the set of entries that are common to all SELECT statements.

The number of columns, as well as the datatypes muse be the same between all the sets.

6.4 MINUS

The MINUS operators combines two SELECT statements, and returns the set of entries that are unique to the first select statement. This means that if any entries in the first statements are found in the second as well, they will not be included in the resulting set.

The results of the two SELECT statements must have the same number of columns, and the same datatype for each column.

7 Subqueries and Views

Subqueries and Views are powerful tools in SQL that allow us to write much more sophisticated and customized queries. Creating views and incorporating subqueries into our SQL toolkit allow us to query from a table that is not in the database.

7.0.1 Subqueries

Subqueries are a fairly simple concept, but the SQL queries that we are able to write using this simple concept become exponentially more complicated.

Subqueries are used when we would like to construct an intermediate or nested query and in turn query FROM that intermediate query.

So far we have been constrained to only working with the tables that we have at hand which makes the syntax easier to digest, but in turn limits what we can derive from our data.

The Structure of a Subquery
The typical structure of the subquery is as follows:

```
SELECT column1, column2, ...
FROM (SELECT column1, column2, ...
    FROM table1, table2,...
    WHERE condition
    GROUP BY column_name
    HAVING condition
    ORDER BY column_name ASC/DESC)
WHERE condition
GROUP BY column_name
HAVING condition
ORDER BY column_name ASC/DESC;
```

As you can see from the structure of the subquery that the syntax of our queries are now more complex, however it allows us to be more flexible in what information/tables we can query from.

7.0.2 Views

Views much like subqueries allow us to access a table not already in our database, but rather created using another query.

Structure of Views
The structure of VIEWs follow a similar sort of logic as the structure of subqueries however, VIEWs are created before the main query not within them.

```
CREATE VIEW AS OurView
SELECT column1, column2,...
FROM table1, table2,...
WHERE condition
GROUP BY column_name
HAVING condition
ORDER BY column_name ASC/DESC;

SELECT column1, column2, ...
FROM OurView
WHERE...
GROUP BY...
HAVING...
ORDER BY...
```

Conclusion
As you can see VIEWs and subqueries are a great way to make our queries more sophisticated and optimize the amount of information we can retrieve from our database using SQL. We won't provide examples here, but we encourage you to play around with using subqueries and VIEW creation in the assignment this week.

8 Contributors

Justin Haug

Ryan Van de Water

David Zagaynov