SQL for Data Science

Hui Lin

2018-01-10

Contents

Statement

This is my learning notes on SQL. It is initially from Sadie St. Lawrence's Coursera course "SQL for Data Science".

Chapter 1

Getting Started

In this section, you will be able to define SQL and discuss how SQL differs from other computer languages. You will be able to compare and contrast the roles of a database administrator and a data scientist, and explain the differences between one-to-one, one-to-many and many-to-many relationships with databases. You will be able to use the `SELECT` statement and talk about some basic syntax rules. You will be able to add comments in your code and synthesize its importance.

Learning Objectives

- Distinguish between use of SQL for data science applications and SQL for more common data management operations.
- Use an Entity Relationship diagram, describing the data elements, their relationships and inter-dependencies and determine if the existent data is sufficient to address a business question.
- Identify a subset of data needed from a column or set of columns and write and SQL query to limit to those results.
- Create an analysis environment and use `INSERT` to put data into a table
- Add effective comments in your queries so that:
    1. You can remember what you are doing
    2. others can review your work

1.1 What is SQL?

Structured **Q**uery **L**anguage (SQL) is a standard computer language for relational database management and data manipulation. SQL is used often to query, insert, update and modify data. At a basic level, SQL is a way to communicate with database. Many SQL commands are descriptive words and easy to interpret compared to many other computer languages. This makes SQL an easy to understand and learn language.

Another important thing to know about SQL is that it is a non-procedural language. That means you won't be able to write complete applications with it. This makes SQL relatively simple but also very powerful language to interact with data. SQL is all about data and it is used for three things:

- read and retrieve data from database
- write data into database
- update and insert new data

Different SQL users

There are a lot of jobs requre SQL and it is not just for data science. It is important to understand how different roles might use SQL in their jobs. The users can be data scientist, programmers, backend developer, QA engineers, data architects, system engineers and database administrators (DBA). I want to talk a little more about how DBA use SQL comparing to data scientist.

- A DBA is responsible for **managing the entire database and guarding it**.
- A data scientist, on the other hand, is typically **a user of that database**.

The DBA will be responsible for giving permissions to people and determining who has access to what data. They are often responsible for managing the tables and creating them. Data scientist need to get the rights from DBA to create his/her own table and insert data into them. The two roles are similar in that they both use SQL to interact with the data. But the main difference is that the data scientist is really the end user. Whereas the DBA is the one who administers it, governs it and manages the database, as a whole.

Database Admin	Data Scientist
Manages/governs entire database	End user of a database
Gives permissions to users	Retrieve data (mainly)
Determines access to data	May create their own table or test environment
Manages and creates tables	Combine multiple sources together
Uses SQL to query and retrive data	Writes complex queries for analysis (maybe but usually not)

SQL and Database Management System(DBMS)

Despite SQL being standardized since 1986, a lot of different implementations exist. They deviate more or less from each other. You can think of SQL as the interpreter between you and the database. How you write some of the syntax for SQL depends on the relational

database management system you are using. Here are some of the popular ones:

- SQL Server
- IBM DB2 Oracle
- Sybase ASE
- PostgreSQL
- MySQL
- Microsoft SQL Server
- Apache Open Office Base
- SQLite

In this text, we'll use SQLite. It's important to understand that if you copy code from this text into another application at work, it may not work correctly. You should check the type of DBMS you're using and see if that makes a difference. We will talk about this more when we get to the syntax later including some of the ways to figure out what those differences might be.

Summary points:

- How you write syntax will depend on what DBMS you are using
- Each DBMS has its own "dialect"
- SQL can translate
- You will tweak based on the "dialect" your DMBS speaks

Chapter 2

Data Models and Diagrams

Learning Objectives

- Explain why thinking beforing coding is important
- Explain why it is important to understand how the data in a database relates to one another
- Describe what a databse is at its core
- Describe data models
- Define relational database system
- Discuss advent of relational databases in SQL

2.1 Think Before Code

The reason why thinking beforing coding is importan is because you have to understand the structure of the data well to effectively write queries. Understanding your data means the following:

- Understand the business process or subject matter the data is modeled after
- Know the business rules
- Understand how your data is organized and structured in the table

Before you start to write a query, think about:

- what is the problem you are trying to solve?
- what is the data you need to get?
- how does the data relate to each other?
- how does it interact?
- what are some of the problems that you may want to solve with this data and need to be aware of?
- what are the types of joints or business processes in the data modeling?

This will really help you because not only will you get more accurate results, but also speed up the time it takes you to work and get things done. If you start to think about what you're doing before you do it, you should hopefully also have less rework.

Now what is a database and what is a table?

> **Database**: A container (usually a file or set of files) to stre organized data; a set of related information
>
> **Tables**: A structured list of data or a specific type

A database is really a container that is usually a file or set of files and is used to organize and store all of the data. If you think of this in real world terms, It'd be like a filing system that has many cabinets along a wall. Within that system, within a database, we have tables, these tables are structured lists of data elements or specific data type. Going back to our analogy, you can think of this as maybe one of the cabinets within a whole wall of cabinets. Then if we dive further into the cabinet, into a table, what we find is we have columns and rows, which of course is what makes up a table. A table is made up of a series of individual columns, and then a row and a table is a record. Through tables, rows, and columns, ultimately throughout the database we have a mechanism to store and retrieve data.

2.2 What is Data Model?

Data model is what we use to organize information for multiple tables and how they relate to each other together. This helps tremendously in providing structure to the information in the system. Usually a data model represents a business process and it also helps you understand a business process. As a data scientist, you often need to work with a business person in understanding the data and how it fits together. But at the same time, the business person will learn a lot from the data modeller to better understand how their business actually works together by seeing the data and how it interacts with each other.

The data model here is not predictive model which a data scientist often build. It is a way the tables are represented and organized in a database. One thing to remember is that a data model should always represent a real world problem as closely as possible. There are couple different types of models, and there has been an evolution of data models.

The evolution of data model traces back to 1960s. There's been hierarchical, network, relational, entity, relational somatic, and NoSql.

Here we will briefly talk a little about the relational and NoSql. Because we are going to work a lot with relational model. If you are interested in learning more, there is material widely available on the internet and you can do your own research.

The benefits of a relational model are:

- simplify the connections between the data
- allow you to write queries (retrieve/update/write data) easily

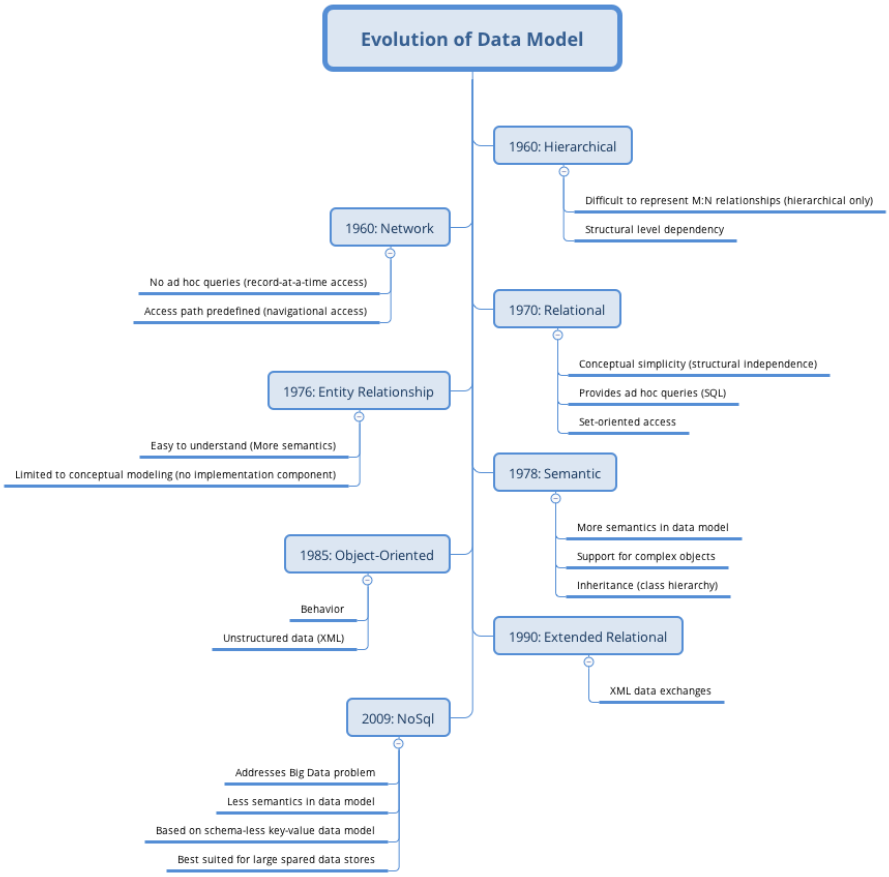

Figure 2.1

NoSQL was part of the Big Data movement that you should have already heard about. It is a mechanism for storage and retrieval where it's not modeled in a tabular relational format. NoSQL was popular when big data and unstructured data first came out because you left it unstructured, but it's now started to soften a little bit, and more commonly referred to as Not Only SQL. One question to think about: does SQL really have a role still in the Big Data world, as new things start to come out like NoSQL and unstructured data?

Next let's talk about the difference between relational and transactional databases. A **relational model** is a database design that shows the relationships between the different tables, optimizes querying data, makes it easy and intuitive to access the data. **Transactional model** is a more operational database. If you are in healthcare, for example, you may have a transactional database that is used to store all the claims information and then this information may not be stored in a great way for querying and using it for analysis. In fact, you may need to take and extract that transactional information from the database

and move it into a relational model.

Most of what we will be working with is the relational model. There are 3 building blocks for relational model:

1. Entities: a person, place, thing or event. These are very distinguishable, unique and distict.
2. Attributes: characteristics of this entity.
3. Relationships: associations among different entities, can be one-to-many, many-to-many and one-to-one.

For example, Pioneer has a great corn seed product called P1197 which could be an entity. And then we have attributes that are characteristics of P1197, such as price, average yeilds, units sold, promotion programs. If you think of a one-to-many relationship, this could be P1197 has many promotion programs. When you think of a many-to-many relationship, this could be an example of many products to many different promotion programs. You may have one product that belongs to different programs or you may have a program that is available for different products. Then, if you think of a one-to-one relationship, it could be one product has a unique price.

To understand these relationships between the tables a lot better, what's often used to depict this are ER diagrams. ER model is composed of entity types and specific relationships that can exist between instances of those entity types. These are usually displayed in a visual format and a relate represents a **relationship** between the tables. It often helps you to understand and represent a **business process** and it will show the **links** between these tables. The links are important when we join these tables. **Being able to look at this diagram and see how they relate to each other** is really important.

We can use the **primary key** or **foreign key** to join tables. The primary key is a column or set of columns whose values uniquely identify every row in a table. Foreign key is one or more columns can be used together to identify a single row in another table. When we're looking at ER diagrams, which again is one of the ways you will start to think before you do, you'll look at maybe an ER diagram and understand what data elements you are trying to join together and how do you need to get them. But one of the things you need to understand is how to read this. We talked a little bit about relationships and the different relationships between a table. There is a different type of notation that explains the relationships.

- Chen notation
- Crow's foot notation
- UML class diagram notation

The Chen notation uses `1:M` for a one-to-many relationship, and `M:N` for a many-to-many relationship and `1:1` for a one-to-one relationship.

In Crow's foot notation, we have the train tracks which represent 1 and then the Crow's foot which represents many.

In UML notation, we have a 1.1 which represents the concept of one and 1.* which represents the concept of many.

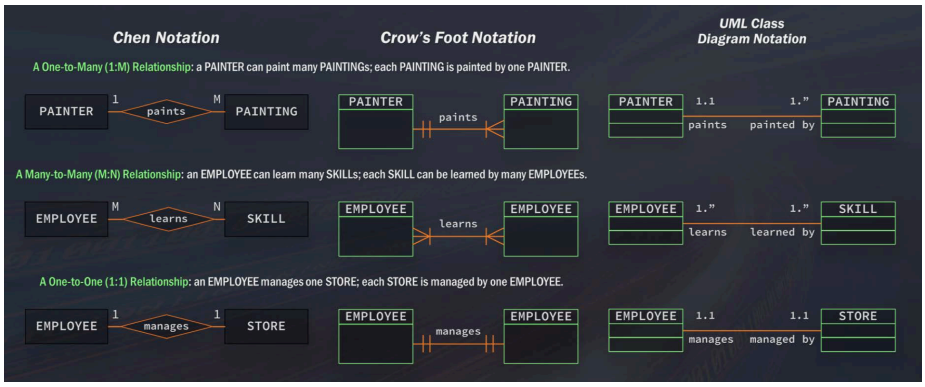

Figure 2.2

Get familar with the different notations since you'll be looking at ER diagrams quite frequently and you'll need to understand these notations when reading ER diagrams so you can understand how you're going to write your query and join the table together or even to find out what's listed in the table. Having a good understanding of why the data is structured in a particular way and how to read the ER diagrams is necessary for writing queries and ensuring accurate results.

2.3 Retrive Data with SELECT

The majority of what data scientists are doing with SQL is retrieving data. To get started, the first statement to learn is SELECT. There are two pieces of information to specify in SELECT statement: what to select and where it is from. It will be straightforward with an example.

```
SELECT prod_name
FROM Products;
```

```
prod_name
Shampoo
Toothpaste
Deodorant
Toothbrush
```

Ihis example selects product name which a column from the table (`prod_name`). Then you need to specify where to get it from. Here it is from a table named `Products`. The output of this is then going to look like the column listed below. Which it has a column product name and then all of the list of products. We have shampoo, toothpaste, deodorant, and toothbrush. If you want to retrieve more than a single column from a table, then what you need to add the names of the individual columns separated with a comma:

```
SELECT prod_name, prod_id, prod_price
FROM Products;
```

Request all columns by using the asterisk (*) wildcard character instead of column names:

```
SELECT *
FROM Products;
```

This is going to go ahead and grab everything from the Products table, each individual column, and put it into your output.

A lot of times we may want get a view of the table to understand what data is in there without pulling all the records. In this case, we may do a "`SELECT *`". But if there are 5 million records in it, we just need a tiny sample of that. In this case, we can use `LIMIT` to limit the results.

```
SELECT prod_name
FROM Products
LIMIT 5;
```

Here is an example of how different RDMS differ in syntax:

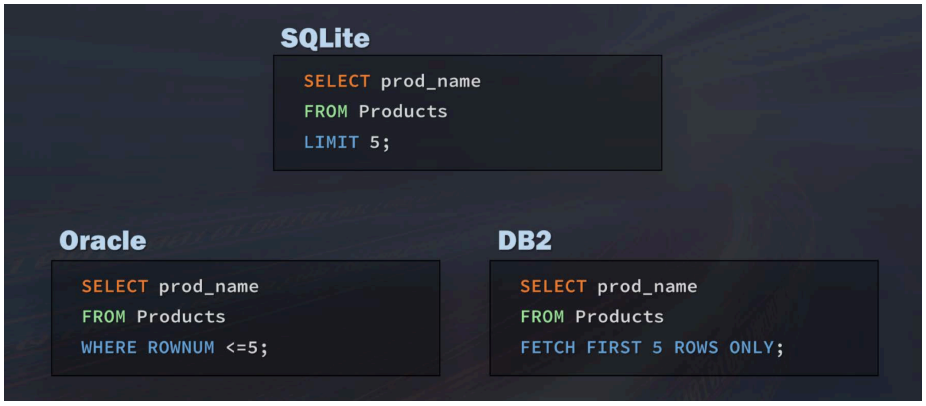

Figure 2.3

If you are using SQLite and understand it's `LIMIT 5`. When switch over to a DB2 system, you can easily Google in terms of the syntax in DB2.

2.4 Create Table

As a data scientist, you are always building models and making predictions. You may want to take those predictions that you create and write them back to a database. This ensures that someone else could then pick up those predictions and use them in a dashboard. Or you may want to create a dashboard or visualize it with another tool that can be hooked

up and used with that database. It's also helpful if you're extracting data off the web or scraping it from somewhere and you want to store this data in a database. As we have previously discussed, the data scientist isn't usually the one in charge of managing the entire database, that usually left to the DBA or some type of administrator. Even so, it's helpful to have a basic understanding of how this works. You can use the statement `CREATE TABLE` to do this.

```
CREATE TABLE Shoes
(
Id     char(10)     PRIMARY KEY,
Brand  char(10)     NOT NULL,
ShoeTypes  char(250)    NOT NULL,
Color  char(250)    NOT NULL,
Price  decimal(8,2) NOT NULL,
Descp  Varchar(750) NULL
);
```

In the above example, you create a table named "`Shoes`" by putting the name after `CREATE TABLE`. Then define the list of columns in the brackets. The first column is `Id`. It is a character with lenth 10. And the `Id` is going to be primary key. So you define data type (how many characters or decimals you allow to be inserted into this column), whether or not you allow null values. By default, it assumes that null values are accepted.

The syntax for creating these tables varies greatly by relational database management system that you're using. The above example exhibits the basic structure to create a table. However, it's important to check the specifications and syntax of the relational database management system you're using. An important thing to note when creating a table is to define whether a column can contain a null value or is a primary key. There are several things to pay special attention here:

1. Don't confuse null value with empty string. Null value is the absence of everything, whereas empty string is a value there, such as space.
2. A primary key cannot accept null values. The `Id` in the example can never have an empty value or not any value.
3. If you indicate that a column cannot be null, then you will get an error if there is any missing value in that column.

There are two ways to incert data to the table after defining the variables:

(1) Use `INSERT INTO` statement

```
INSERT INTO Shoes
VALUES ( '14535974',
'Gucci',
'Slippers',
'Pink',
'695.00',
NULL
);
```

The first method will take the first value indicated and put it in the first column; the second value will go to the second column, and so on and so forth. It works. However, a potential problem of this method is that you have no guarantee that data is going into the correct column. It is better to be more specific.

(2) Use `INSERT INTO` and `VALUES`

```
INSERT INTO Shoes
(Id,
Brand,
ShoeTypes,
Color,
Price,
Descp
)
VALUES
(
'14535974',
'Gucci',
'Slippers',
'Pink',
'695.00',
NULL
);
```

The second method also lists the columns. This can be really beneficial if you want to insert values into some columns but not all. I will recommend using this method. It's a little safer because you have more control and know exactly which value is going and into which column.

Now you have learned how to create tables using SQL. Another option is to create a copy or get a subset of an existing table. A table created this way is a **temporary table**. The most important thing to know about temporary tables is that they will be deleted when the current client session is terminated. That's why they're called temporary tables. Why do we need temporary table? Because it is mush faster than creating a real table. If you have complex queries and you want to simplify it a bit by creating a subset and then joining to that subset and driving a new calculation from that, then temporary table is a great option. You can use the statement `CREATE TEMPORARY TABLE` to do this:

```
CREATE TEMPORARY TABLE Sandals AS
(
SELECT *
FROM Shoes
WHERE ShoeTypes = 'sandals'
)
```

In this case, we create a subset of data from `Shoes` table and name the new table `Sandals`. The new table is all records with shoe type sandals.

2.5 SQL Comments

You may go back to some historical query and modify the query to retrieve some new data. Comments can help you remember what you were doing and why. You can also use the comments to mute the expression of some code, frequently referred to as commenting out code. This technique helps you troubleshoot some of the issues you have with your query. You can effectively get rid of parts of your query without actually getting rid of the statements themselves. And then bring them back in one by one to see where your query goes awry.

There are two ways of comment.

(1) Single line

```
SELECT Id,
-- Brand,
ShoeTypes,
Color,
Price,
Descp
FROM Shoes
```

The above code uses `--` to comment out `Brand,`

(2) Section

```
SELECT Id,
/* Brand,
ShoeTypes,
Color, */
Price,
Descp
```

You can use a combination of a backslash(\) and an asterisk(*). What this is effectively saying is, don't run anything between the two backslashes and the asterisk.

2.6 Summary

In this chapter, we went over many materials. We began by defining SQL. You should know now that SQL stands for Structured Query Language which is a standard language to communicate with relational database management systems (RDMS). You should also know the difference between a transactional and a relational database. We also introduced important concept of primary key, foreign key, and table relationship. Make sure you are familiar with these concepts because we're going to revisit them a lot in more detail later when we start talking about Joints. We also discussed some basic SQL Syntax. And you should now be able to write basic queries statements using `SELECT` and `FROM`.

Finally, we wrapped up by going over how to write comments in your code which are essential to include so that both you and your colleagues can follow what you were doing and reuse your code. It's a really good habit to get into as you're starting out. So be sure to keep practicing that aspect as you begin to write your first query statement.

Here are some resources for further learning:

- What is SQL and How is it Used?[1]
- NTC Hosting: Structured Query Language[2]
- SQLite Tutorial[3]
- Entity Relationship Diagram[4]
- Norwalk Aberdeen: Entity-Relationsip Diagrams (9 Minute YouTube Video)[5]
- Star Schema vs. Snowflake Schema[6]
- Explain Star Schema & Snow Flake Design (5 Minute YouTube Video)[7]
- Data Modeling 101[8]
- What is Data Modeling - An Introduction for Business Analysts[9]
- Wikipedia: Data Modeling[10]

[1] https://www.thebalance.com/what-is-sql-and-uses-2071909
[2] https://www.ntchosting.com/encyclopedia/databases/structured-query-language/
[3] https://www.tutorialspoint.com/sqlite/index.htm
[4] http://www.information-management-architect.com/entity-relationship-diagram.html
[5] https://www.youtube.com/watch?v=c0_9Y8QAstg
[6] http://www.vertabelo.com/blog/technical-articles/data-warehouse-modeling-star-schema-vs-snowflake-schem
[7] https://www.youtube.com/watch?v=KUw0cip7Zzc
[8] http://www.agiledata.org/essays/dataModeling101.html
[9] http://business-analysis-excellence.com/what-is-data-modeling/
[10] https://en.wikipedia.org/wiki/Data_modeling

Chapter 3

Filtering, Sorting, and Calculating Data with SQL

3.1 Filter

3.1.1 WHERE

Learning Objectives

- Describe the basics of filtering your data
- Use the WHERE clause with common operators
- Use BETWEEN clause
- Explain the concept of a NULL value

Filtering allows us to narrow the data we want to retrieve. Filtering is also used when you're doing analysis to get very specific about the data you want to analyze as part of your model. To do this we use what's called the WHERE clause. And the WHERE clause comes after SELECT and FROM

```
SELECT column_name, column_name
FROM table_name
WHERE column_name operator value;
```

Here is a table of common operators:

Operator	Description
=	Equal

Operator	Description
<>	Not equal. Note: In some versions of SQL this operator may be written as !=
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
BETWEEN	Between an inclusive range
IS NULL	Is a null value

Let's look at some examples.

```
SELECT ProductName,
UnitPrice,
SupplierID
FROM Products
WHERE ProductName = 'Tofu';
```

In the above example, we filter on a single condition. We look at ProductName, UnitPrice and SupplierID for tofu. You can similarly look at products whose prices are greater than or equal to 75.

```
SELECT ProductName,
UnitPrice,
SupplierID
FROM Products
WHERE UnitPrice >= 75;
```

Or you can filter out all records except one value:

```
SELECT ProductName,
UnitPrice,
SupplierID
FROM Products
WHERE ProductName <> 'Tofu';
```

You can filter for a range of values:

```
SELECT ProductName,
Unitprice,
```

```
SupplierID,
UnitsInStock,
FROM Products
WHERE UnitsInStock BETWEEN 15 AND 80;
```

You can filter NULL values by IS NULL

```
SELECT ProductName,
UnitPrice,
SupplierID,
UnitsInStock
FROM Products
WHERE ProductName IS NULL;
```

3.1.2 IN, OR and NOT

Learning Objectives

- Use the IN and OR operators to filter your data and get results you want
- Differentiate between use of the IN and BETWEEN operators
- Discuss importance of order of operations
- Explain how and when to use the NOT operator

Let's start fromIN. To use the IN operator, we need to specify a range of conditions or a set of values.

```
SELECT
ProductID,
UnitPrice,
SupplierID
FROM Products
WHERE SupplierID IN (9, 10, 11);
```

Another operator is the OR operator. An important thing to know about this is that a database management system will not evaluate the second condition in a WHERE clause if the first condition is met.

```
SELECT
ProductName,
ProductID,
UnitPrice,
SupplierID,
ProductName
FROM Products
WHERE ProductName = 'Tofu' OR 'Konbu';
```

You can use OR with AND. There is somthing you need to pay extra attention. Look at the following two examples:

- Example1:

```
SELECT
ProductID,
UnitPrice,
SupplierID,
FROM Products
WHERE SupplierID = 9 OR
SupplierID = 11
AND UnitPrice > 15;
```

	ProductID	UnitPrice	SupplierID
1	22	21	9
2	23	9	9
3	26	31.23	11
4	27	43.9	11

Figure 3.1

- Example2:

```
SELECT
ProductID,
UnitPrice,
SupplierID
FROM Products
WHERE (SupplierID = 9 OR
SupplierID = 11)
AND UnitPrice > 15;
```

	ProductID	UnitPrice	SupplierID
1	22	21	9
2	26	31.23	11
3	27	43.9	11

Figure 3.2

Since SQL processes the `OR` before the `AND`, in example1, SQL will stop after processing `OR`, so it won't even get to the operation after `AND`. In order to solve that, you need to use parentheses.

Even you don't have to use a parenthesis, but it's always really recommended. This way you're not relying on the default order of operations.

You can use `NOT` to get the supplementary part:

```
SELECT *
FROM Employees
```

```
WHERE NOT City='London' AND
NOT City='Seattle';
```

Wildcards

Have you ever come across data where you knew either the beginning or end of something, but didn't know the rest of it? Or maybe you know that something is like something else, but slightly different. In the rest of this section, we are going to discuss the use of the wildcards and the LIKE operator. You will learn the concept of wildcards, including their advantages and disadvantages.

Wildcard is a really powerful especially for string values or text data. A wildcard is a special character used to match parts of a value. You search for a pattern of string. LIKE works for both string and numerical variables. But wildcards cannot be used for numerical data.

% Wildcards

Wildcard	Action
%carrot	Grab anything ending with the word carrot
carrot%	Grab anything after the word carrot
%carrot%	Grab anything containing the word carrot
today%happy	Grab anything that starts with today and ends with happy
t%@gmail.com	Grab gmail address that start with t

Underscore(_) wildcard matches a single character but it is not supported by DB2.

```
WHERE size LIKE `_carrot`
```

Output:

```
lcarrot
mcarrot
scarrot
```

It is identical with:

```
WHERE size LIKE `%carrot`
```

Bracket ([]) wildcard specifies a set of characters in a specific location. It does not work with all DBMS. It does not work with SQLite.

There are some downsides to using wildcards:

- Takes longer to run
- Better to use another operator (if possible) =,<,>= etc.
- Depending on the system
- Hard to read

3.2 Sort

Learning Objectives

- Explain some of the rules related to using the `ORDER BY` clause
- Use the `ORDER BY` clause to sort data either in ascending or descending order

`ORDER BY` allows us to sort data by particular columns. Now there are a few rules when using `ORDER BY`:

1. It can take multiple column names.
2. If you're doing multiple columns, you just want to make sure you're adding a comma after that.
3. You can sort by a column that you didn't retrieve.
4. It must always be the last clause in the select statement.

You can sort by column position. For example, sort by column 2 and 3:

```
ORDER BY 2,3
```

There are also some directions as with any type of sorting, either in ascending, `ASC`, or descending order, `DESC`. This is only applied to the column name it directly proceeds. If you're using order by descending and have unit price, it's not going to do it for all of columns after the `DESC`. You have to specify each individual columns for ascending and descending, if you want it that way.

3.3 Math Operations

Learning Objectives

- perform basic math calculations
- discuss in more detail the concept and the order of operations
- describe what can be done in terms of analysis by using math operators and SQL with math calculations

Let's start with some simple ones. We have the addition, subtraction, multiplication, and division.

```
SELECT
ProductID,
UnitsOnOrder,
UnitPrice,
UnitsOnOrder * UnitPrice AS Total_Order_Cost
FROM Products
```

In this example, we get the total order cost by multiplying units on order with unit price. And then, we name the new column `Total_Order_Cost` by using `AS`.

You dont't have to SELECT the unit price and the units on the order to calculate the total cost. You could have just selected the product ID and then calculated the new field. But it is a nice thing to make sure it all adds up and looks right.

Now let's add these operators together. Just as any math you're doing, it's going to follow normal order of operations. You probably remember the order of operations from past math classes. Operations in parentheses are handled first, then power, multiplication, division, addition, and subtraction.

```
SELECT
ProductID,
Quantity,
UnitPrice,
Discount,
(UnitPrice - Discount)/Quantity AS Total_Cost
FROM Products
```

3.4 Aggregrate Functions

Learning Objectives

- Describe various aggregate functions
- Use various aggregate functions: AVERAGE, COUNT, MIN, MAX and SUM to summarize and analyze data
- Describe the DISTINCT function

Aggregate functions are used for various things such as finding the highest or lowest values, total number of records, average value, etc. We're going to use a lot of these different types of aggregate functions to get descriptive statistics. The aggregate functions we can use are AVG, COUNT, MIN, MAX, and SUM and all of these are pretty self explanatory.

Function	Description
AVG()	Averages a column of values
COUNT()	Counts the number of values
MIN()	Finds the minimum value
MAX()	Finds the maximum value
SUM()	Sums the column values

- AVG()

```
SELECT AVG(UnitPrice) AS avg_price
FROM Products;
```

- COUNT(*): Counts all the rows in a table containing NULL values

```
SELECT COUNT(*) AS total_customers
FROM Customers;
```

- `COUNT(column)`: Counts all the rows in a specific column ignoring `NULL` values

```
SELECT COUNT(CustomerID) AS total_customers
FROM Customers;
```

- `SUM()`

```
SELECT SUM(UnitPrice*UnitsInStock) AS total_price
FROM Products
WHERE SupplierID = 23;
```

One important thing to use with aggregate functions is the word `DISTINCT`. If the word `DISTINCT` isn't specific in a statement, SQL will always assume you want all the data. For example, you may have a customer who's in a table multiple times. If you're counting customer IDs, you may count duplicate records in there. And this is really helpful to run queries where you're counting distinct and to see if there are duplicates in a column. There are some things to keep in mind when using `DISTINCT` with our aggregate function of `COUNT` You can't use `DISTINCT` on the `COUNT(*)`.

```
SELECT COUNT(DISTINCT CustomerID)
FROM Customers;
```

3.5 Group Data

Learning Objectives

- perform additional aggregations using `GROUP BY` and `HAVING` clause
- discuss how `NULL`s are or aren't affected by the `GROUP BY` and `HAVING` clauses
- use the `GROUP BY` and `ORDER BY` clauses together to better sort your data

A lot of times, we'll be looking at the average price for different types of products, or total perchasing for different customer segments. Then we need to aggregate data by groups. In this example, assume we want to know the number of customers we have by each region.

```
SELECT Region, COUNT(CustomerID) AS total_customers
FROM Customers
GROUP BY Region;
```

If we were to just have our `SELECT` statement with `Region, COUNT(CustomerID) AS total_customers`, but without specifying the variable to `GROUP BY`, we're going to get an error return. Because the computer doesn't know how to count the customer IDs. So we put the `GROUP BY` clause in the end. There are three things to pay attention:

1. `GROUP BY` clauses can contain multiple columns.

2. Every column in your SELECT statement must be present in a GROUP BY clause, except for aggregated calculations.
3. NULLs will be grouped together if your GROUP BY column contains NULLs.

HAVING clause works for groups. But WHERE filters on rows not groups.

```
SELECT CustomerID, COUNT (*) AS orders
FROM Orders
GROUP BY CustomerID
HAVING COUNT (*) >=2;
```

To sum up, WHERE filters before the data is grouped and then HAVING filters after the data is grouped. Rows eliminated by the WHERE clause will not be included in the GROUP BY clause. It is important to know when you should use WHERE versus HAVING.

Another thing to note about GROUP BY is that it's always a good practice to use the ORDER BY clause. The GROUP BY does not sort the data in any fashion. It only groups it together. In our previous examples, we have a list of states, a list of regions. It's not going to sort those regions in alphabetical order. It's just going to group them by different regions. It is recommended to use ORDER BY in this situation. It makes the results a little easier to read.

```
SELECT SupplierID, COUNT(*) AS Num_Prod
FROM Products
WHERE UnitPrice >= 4
GROUP BY SupplierID
HAVING COUNT (*) >= 2;
```

3.6 Summary

Here is a table of key SQL clauses in order:

Clause	Description	Required
SELECT	Columns or expressions to be returned	Yes
FROM	Table from which to retrieve data	Only if selecting data from a table
WHERE	Row-level filtering	No
GROUP BY	Group specification	Only if calculating aggregates by group
HAVING	Group-level filter	No
ORDER BY	Output sort order	No

SQL for various data science languages

- SQL for R (You should really learn dplyr……the package here is just FYI)
  - SQLDF Package[1]

[1] https://cran.r-project.org/web/packages/sqldf/index.html

  - Documentation[2]
  - Examples[3]
- SQL for Spark
  - Documentation[4]
- SQL with Hadoop
  - Hive Overview[5]
  - Documentation[6]
- SQL for Python
  - Python-SQL Package Documentation[7]

[2] https://cran.r-project.org/web/packages/sqldf/sqldf.pdf
[3] https://www.r-bloggers.com/manipulating-data-frames-using-sqldf-a-brief-overview/
[4] https://spark.apache.org/docs/latest/sql-programming-guide.html#overview
[5] https://hive.apache.org
[6] https://cwiki.apache.org/confluence/display/Hive/LanguageManual
[7] https://pypi.python.org/pypi/python-sql

Chapter 4

Subqueries and Joins

4.1 Subqueries

- Define subqueries
- Discuss advantages and disadvantages of using subqueries
- Explain how subqueries help us merge data from two or more tables
- Write more efficient subqueries

Subqueries are queries embedded into other queries. Since data is stored in multiple tables, you may need to get information from multiple sources. Data scientists often use subqueries to:

- Select specific records or columns and then use the criteria as filtering criteria for the next thing they want to select.
- Add additional criteria like filtering criteria that are not in your current table from another table into your query.

Example: Need to know the region each customer is from who has had an order with freight over 100. The freight information is in table `Orders` and the customer information is in table `Customers`.

```
SELECT customerID, CompanyName, Region
FROM Customers
WHERE customerID in (SELECT customerID FROM Orders WHERE Freight > 100);
```

DBMS is performing two operations:

1. Get the customer ID for the orders selected
2. Pass that to the `WHERE` clause and processing the overall `SELECT` statement

Note that it always perform the innermost `SELECT` portion first.

- **Best Practices and Considerations**

  - There is no limit to the number of subqueries you can have
  - Performance slows when you nest too deeply.
  - Subquery selects can only retrieve a single column

Also, when you use subqueries, the code gets harder to understand. One of the best practices for writing subqueries is to be clear and consistent with indenting to make it easier to read. This website will pre-format SQL code for you: www.poorsql.com

Subqueries can also be used as calculations.

Example: Get the total number of orders by each customer

```
SELECT customer_name,
    customer_state (SELECT COUNT(*) AS orders FROM Orders WHERE
        Orders.customer_id = Customer.customer_id) AS orders
FROM customers
ORDER BY Customer_name
```

As you can see, we're selecting customer name, the region and then just as if we were going to select a different column name we have a whole subquery in there. We have `(SELECT COUNT(*) AS orders FROM Orders`, and count these based on the customer IDs. It's aggregating the orders based on the customer IDs (`Orders.customer_id = Customer.customer_id`). In the end, we order the table by customer name.

4.2 Joins

Benefits of breaking data into tables:

- Efficient storage
- Easier manipulation
- Greater scalability
- Logically models a process
- Tables are linked by common values (keys)

Joins associate corrects from each table and allow data retrieval from multiple tables in one query. Note that joins are not physical. They persist for the duration of the query execution.

- Cartesian (cross) joins: each row from the first table joins with all the rows of another table

If table 1 has n_1 rows and table 2 has n_2 rows, then after cartesian join, the resulted table will have $n_1 \times n_2$ rows. It is not frequently used and is computationally taxing.

Example: - Table 1: vendor_name - Table 2: product_name, product_price

```
SELECT vender_name, product_name, product_price
FROM Vendors, Products
WHERE Vendors.vendor_id = Products.vendor_id;
```

It does not match anything. It's just multiplying what you had in the first table with those records in the second table.

- **INNER JOIN** selects records that have matching values in both tables

It is one of the most frequently used joins in SQL.

Example:

```
SELECT suppliers.CompanyName, ProductName, UnitPrice
FROM Suppliers INNER JOIN Products
ON Suppliers.supplierid = Products.supplierid
```

Inner join syntax:

- Join type is specified (INNER JOIN)
- Join condition is in the FROM clause and uses the On clause
- Joining more tables together affects overall database performance
- You can join multiple tables, no limit
- List all the tables, then define conditions

Example:

```
SELECT o.OrderID, c.CompanyName, e.LastName
FROM ( (Orders o INNER JOIN Customer c ON o.CustomerID = c.CustomerID)
INNER JOIN Employees e ON o.EmployeeID = e.EmployeeID
);
```

Be careful about the names and make sure you don't make unnecessary joins.

Aliases and self-joins

SQL aliases give a table or column a temporary name. It can make column names more readable. An alias only exists for the duration of the query.

- WITHOUT alias

```
SELECT vendor_name, product_name, product_price
FROM Vendors, Products
WHERE Vendors.vendor_id = Products.vendor_id;
```

- WITH alias

```
SELECT vendor_name, product_name, product_price
FROM Vendors AS v, Products AS p
WHERE v.vendor_id = p.vendor_id;
```

Self-join is to join table to itself. For example, assume you want to match customers from the same city. You can take the table and treat it like two separate tables. Join the original table to itself.

```
SELECT column_name(s)
FROM table1 T1, table2 T2
WHERE condition;
```

```
SELECT A.CustomerName AS CustomerName1, B.CustomerName AS CustomerName2, A.City
FROM Customers A, Customers B
WHERE A.CustomerID = B.CustomerID
AND A.City = B.City
ORDER BY A.City;
```

- Left/Right/Full outer joins

LEFT JOIN returns all rows from the left table, even if there are no matches in the right table. **RIGHT JOIN** returns all rows from the right table. **FULL JOIN** returns rows in either table.

- Union

The **UNION** operator is used to combine the result-set of two or more SELECT statements. Each SELECT statement within UNION must have the same number of columns. Columns must have similar data types. The columns in each SELECT statement must be in the same order.

Example:

```
SELECT column_name(s) FROM
table1
UNION
SELECT column_name(s) FROM
table2;
```

- **Best Practices**
  - Check the number of records
  - Does it seem logical given the kind of join you are performing?
  - Check for duplicates
  - Check the number of records each time you make a new join
  - Are you getting the expected results?
  - Start small: one table at a time
  - Think about what you are trying to do and map how you are joining data tables first
  - Use a join condition
- Summary Diagram
- Further Reading
  - Thinking in SQL vs Thinking in Python[1]
  - Difference Between Union and Union All - Optimal Performance Comparison[2]

[1] https://blog.modeanalytics.com/learning-python-sql/
[2] https://blog.sqlauthority.com/2009/03/11/sql-server-difference-between-union-vs-union-all-optimal-perfo

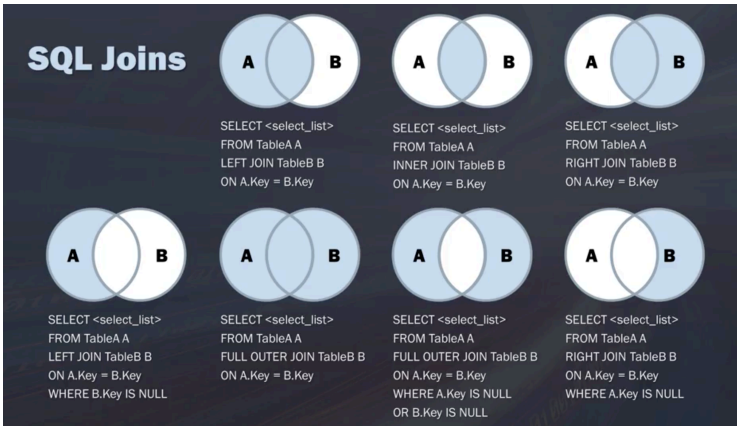
SQL Joins
A
B
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL

Figure 4.1

Bibliography