Cigna

Introduction to SQL

Student Workbook

Version 6.0 C

Table of Contents

Module 1

Introduction to Relational Databases

Section 1–1

Relational Databases

Databases and DBMS

- **Databases are collections of organized data stored such that a program called a database management system (DBMS) can interact with**

- **There are many DBMS systems out there, including:**
  - SQL Server
  - MySQL
  - Oracle
  - IBM Db2
  - PostgresSQL
  - Access
  - MongoDb

- **Modern DBMS fall into two broad categories:**
  - ***SQL based databases*** (ex: SQL Server) are called relational databased and manage their data in collections of related tables
  - ***NoSQL databases*** (ex: MongoDb) have an entirely different way of managing data (often resembling JSON documents)

- **In this course, we will examine only relational databases and the SQL Server DMBS**

Database Operations

- **When we store data in a database, there are four basic types of operations we perform with data**
  - Query the data for answers to questions
  - Add new data
  - Change data
  - Delete data
- **There is a common acronym for these four types of operations: CRUD**
  - Create
  - Read
  - Update
  - Delete

Relational Databases

- **With relational databases, the data is organized in around tables made up of rows and columns.**
  - Each ***column*** represents a particular 'property' we want to store
  - Each ***row*** represents one set of data and is often called a 'record' or 'entry' (or row!)

Results Messages

	customer_id	store_id	first_name	last_name	email	address_id	active	create_date	last_update
1	1	1	MARY	SMITH	MARY.SMITH@sakilacustomer.org	5	1	2006-02-14 22:04:36.000	2006-02-15 04:57:20.000
2	2	1	PATRICIA	JOHNSON	PATRICIA.JOHNSON@sakilacustomer.org	6	1	2006-02-14 22:04:36.000	2006-02-15 04:57:20.000
3	3	1	LINDA	WILLIAMS	LINDA.WILLIAMS@sakilacustomer.org	7	1	2006-02-14 22:04:36.000	2006-02-15 04:57:20.000
4	4	2	BARBARA	JONES	BARBARA.JONES@sakilacustomer.org	8	1	2006-02-14 22:04:36.000	2006-02-15 04:57:20.000
5	5	1	ELIZABETH	BROWN	ELIZABETH.BROWN@sakilacustomer.org	9	1	2006-02-14 22:04:36.000	2006-02-15 04:57:20.000
6	6	2	JENNIFER	DAVIS	JENNIFER.DAVIS@sakilacustomer.org	10	1	2006-02-14 22:04:36.000	2006-02-15 04:57:20.000
7	7	1	MARIA	MILLER	MARIA.MILLER@sakilacustomer.org	11	1	2006-02-14 22:04:36.000	2006-02-15 04:57:20.000
8	8	2	SUSAN	WILSON	SUSAN.WILSON@sakilacustomer.org	12	1	2006-02-14 22:04:36.000	2006-02-15 04:57:20.000
9	9	2	MARGARET	MOORE	MARGARET.MOORE@sakilacustomer.org	13	1	2006-02-14 22:04:36.000	2006-02-15 04:57:20.000
10	10	1	DOROTHY	TAYLOR	DOROTHY.TAYLOR@sakilacustomer.org	14	1	2006-02-14 22:04:36.000	2006-02-15 04:57:20.000
11	11	2	LISA	ANDERSON	LISA.ANDERSON@sakilacustomer.org	15	1	2006-02-14 22:04:36.000	2006-02-15 04:57:20.000
12	12	1	NANCY	THOMAS	NANCY.THOMAS@sakilacustomer.org	16	1	2006-02-14 22:04:36.000	2006-02-15 04:57:20.000
13	13	2	KAREN	JACKSON	KAREN.JACKSON@sakilacustomer.org	17	1	2006-02-14 22:04:36.000	2006-02-15 04:57:20.000
14	14	2	BETTY	WHITE	BETTY.WHITE@sakilacustomer.org	18	1	2006-02-14 22:04:36.000	2006-02-15 04:57:20.000

- **When you create a table, you must specify:**
  - the names of each column
  - the type of data (integer, string, etc.) held in each column
- **You can also set other properties for each column, such as whether it can be null**

SQL

- **Modern relational database management systems use a language called SQL to interact with the data**
  - SQL stands for ‘Structured Query Language’
- **Programmers are opinionated about how to pronounce SQL**
  - The standard says that ‘Ess-cue-ell’ is the correct way pronounce the language
  - Famous authors who originally wrote about the technology, including Jennifer Widom and Christoper Date, pronounce it ‘sequel’
- **A command to fetch data written using SQL is called a ‘query’ and might look like this:**

Example

```
SELECT FirstName, LastName
FROM Customers
WHERE LastName = 'Griffin'
```

- **The information returned from a query is called the ‘result’ or a ‘result set’.**

Peter	Griffin
Stewie	Griffin
Brian	Griffin

Databases: A Collection of Related Tables

- **Relational databases typically consist of many tables**

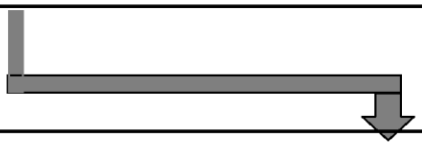
- **Typically, each table contains one type of thing**
  - ex: Products, stores, orders, etc.

- **Relationships between tables are established by having associated columns**
  - The column that uniquely identifies a row is called the ***primary key***
  - When a column in one table refers to the primary key of another table, it is called a ***foreign key*** and establishes a relationship between the two tables

- **One-to-many relationships are represented as shown here:**

DEPARTMENTS

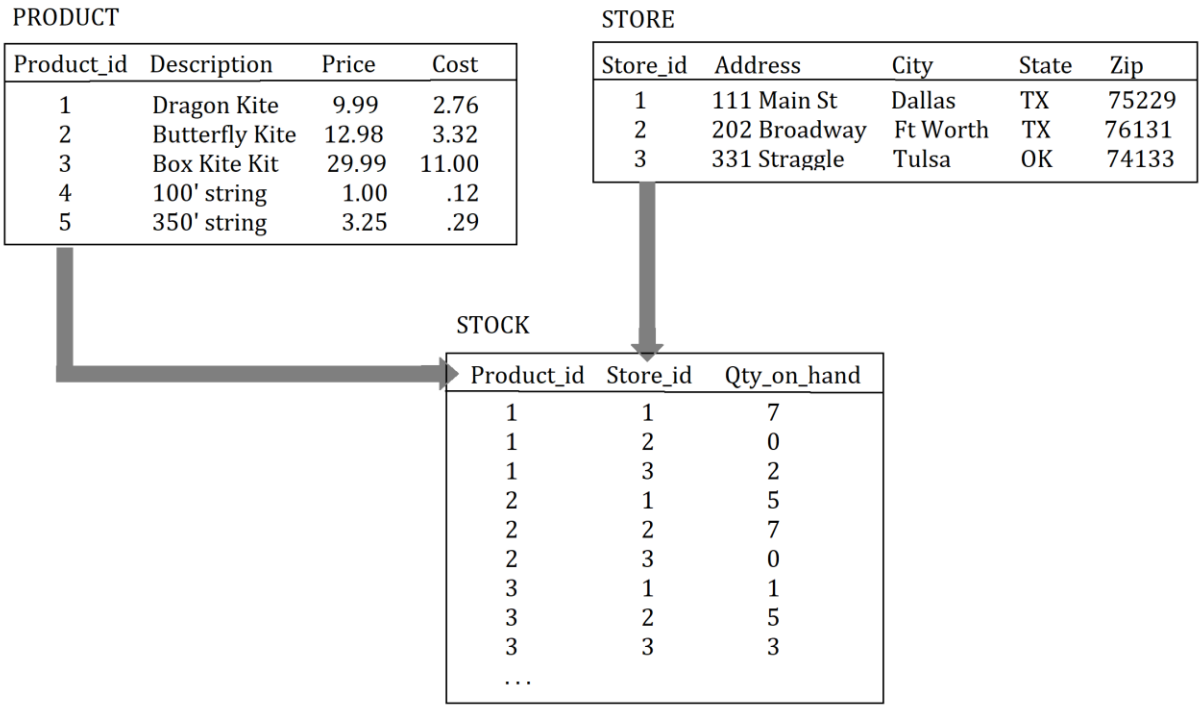
Dept_id	Department	Main Office
BIOL	Biology	Griffith Hall 203
CSCI	Computer Science	Kerr Hall 101
MATH	Mathematics	Gardner 221



FACULTY

Faculty_id	Name	Rank	Dept_id
1	Sallie Sheppard	Professor	CSCI
2	Mike Murphy	Asst Professor	MATH
3	Jane Smith	Assoc Professor	BIOL
4	Virginia Salter	Assoc Professor	CSCI
5	Frank Barnes	Asst Professor	MATH

- **Many-to-many relationships are represented using a third table called an *intersection table*, as shown here:**



PRODUCT

Product_id	Description	Price	Cost
1	Dragon Kite	9.99	2.76
2	Butterfly Kite	12.98	3.32
3	Box Kite Kit	29.99	11.00
4	100' string	1.00	.12
5	350' string	3.25	.29

STORE

Store_id	Address	City	State	Zip
1	111 Main St	Dallas	TX	75229
2	202 Broadway	Ft Worth	TX	76131
3	331 Straggle	Tulsa	OK	74133

STOCK

Product_id	Store_id	Qty_on_hand
1	1	7
1	2	0
1	3	2
2	1	5
2	2	7
2	3	0
3	1	1
3	2	5
3	3	3
. . .		

Normalization: Minimizing Redundancy

- **Reducing redundancy is one of the goals of designing tables**
  - When you duplicate data, changes to data become complicated because you have to make changes in multiple places
  - The process of designing a database that reduces redundancy is called ***normalization***
- **There's no 'one answer' to database design**
  - It is a matter of give and take and what is right for the situation
- **For example, how do you represent Customers and Addresses?**
- **One organization might include customers and their addresses in a single table**

CustomerID	CompanyName	Address	City	Region	PostalCode	Country
ALFKI	Alfreds Futterkiste	Obere Str. 57	Berlin	NULL	12209	Germany
ANATR	Ana Trujillo Emparedados y helados	Avda. de la Constitucin 2222	Mxico D.F.	NULL	05021	Mexico
ANTON	Antonio Moreno Taquera	Mataderos 2312	Mxico D.F.	NULL	05023	Mexico
AROUT	Around the Horn	120 Hanover Sq.	London	NULL	WA1 1DP	UK
BERGS	Berglunds snabbkp	Berguvsvgen 8	Lule	NULL	S-958 22	Sweden
BLAUS	Blauer See Delikatessen	Forsterstr. 57	Mannheim	NULL	68306	Germany

- **Another organization might be aware that multiple customers live at the same address and separate address info into one table and customer info into another**
- **A row in the Customer table is linked to a specific row in the Address table using a key**

Address Table

addressId	addressLine1	addressLine2	city	state	zip
1	222 SW Main Street		Tulsa	OK	55422
2	543 North Washington Avenue		Portland	OR	97204
3	998 Blair Blvd.	Suite 34	New York	NY	22001

Customer Table

customerId	firstName	lastName	addressId
1	Jimmy	Stewart	1
2	Shelly	Duvall	3
3	Robert	Duvall	3
4	Fubar	Wilco	2

Module 2

The SQL Language and Querying Data

Section 2–1

The SQL Language

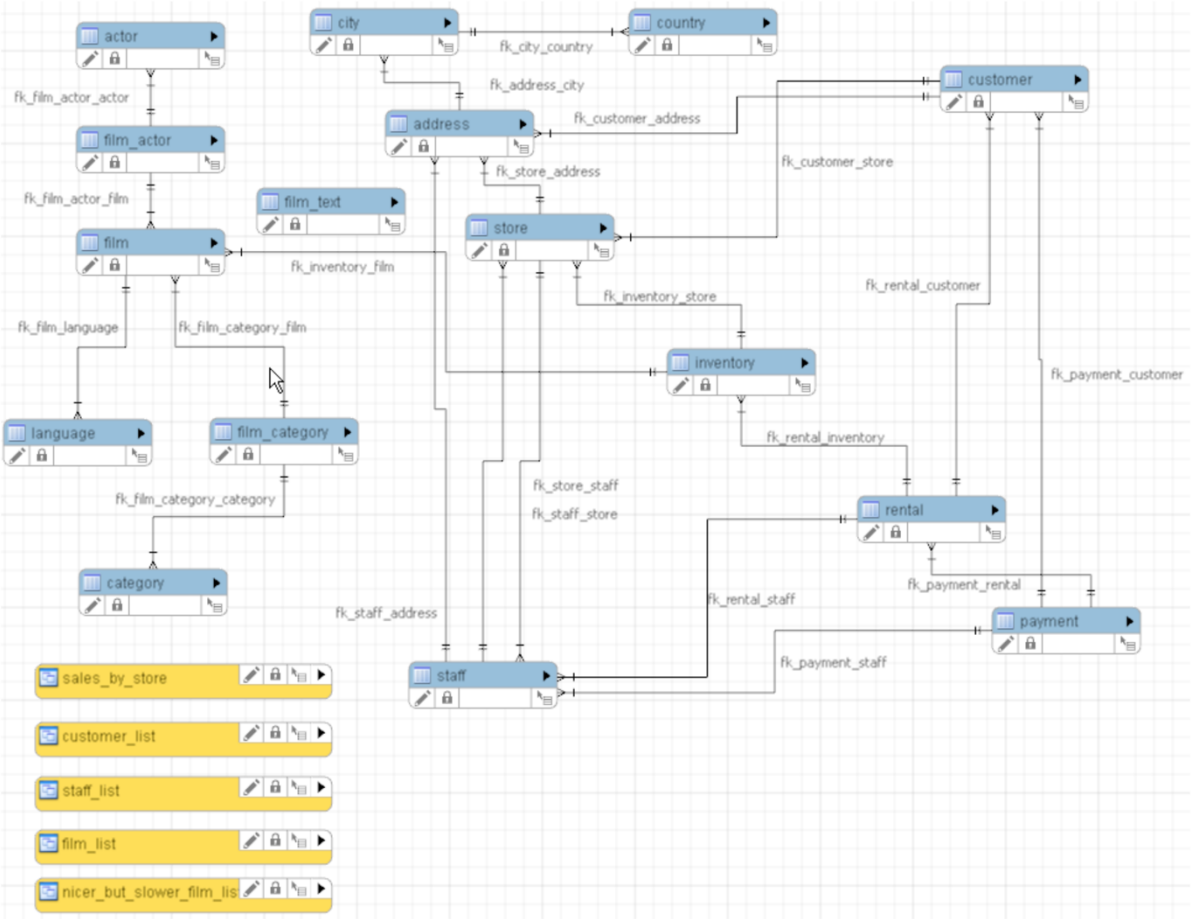
The SQL Language

- **When you want to interact with SQL databases, you use SQL**
- **SQL is a *declarative* language**
  - JavaScript and Java are *procedural* languages
- **The difference is that, in JavaScript, you must specify *how* to do things that you want -**
  - Example: executing a group of statements multiple times with a `for` loop
- **With SQL, you specify *what* you want.**
  - All SQL databases have sophisticated execution optimizers that figure out the best way to produce your results based on what the database looks like when you execute your command
- **Over the next several modules, we will see examples of, and practice, SQL**

Installing SQL Server and Understanding the Database

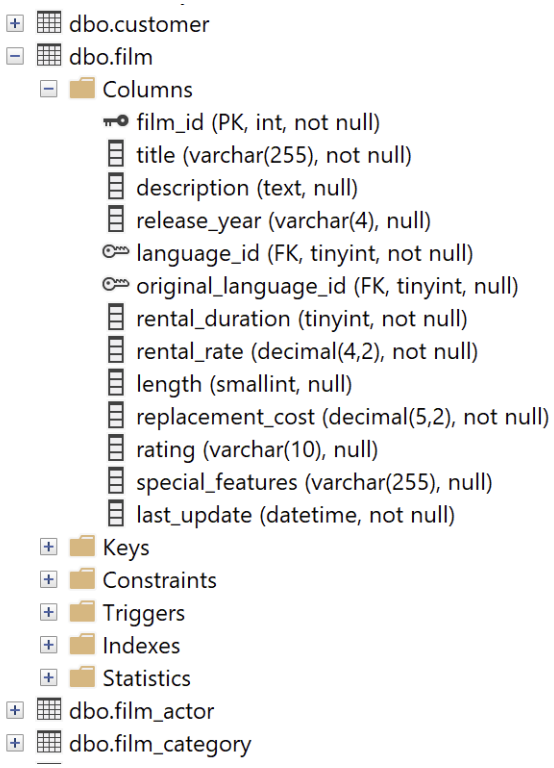
- **The examples in this section of the course often use a sample database called sakila**

- **It is a database for a company that sells movies**
  - Originally created for MySQL, the popular sakila database has been ported over to most other popular databases
  - We will be using the SQL Server port: https://github.com/ivanceras/sakila/tree/master/sql-server-sakila-db



- **Many of our early queries will use the film table**
  - The film table is a list of all films that potentially might be in stock in the stores
  - The actual in-stock copies of each film are represented in the inventory table.

- **The film table contains the following columns:**

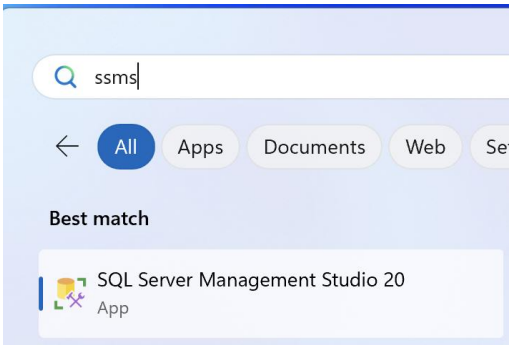


Section 2–2

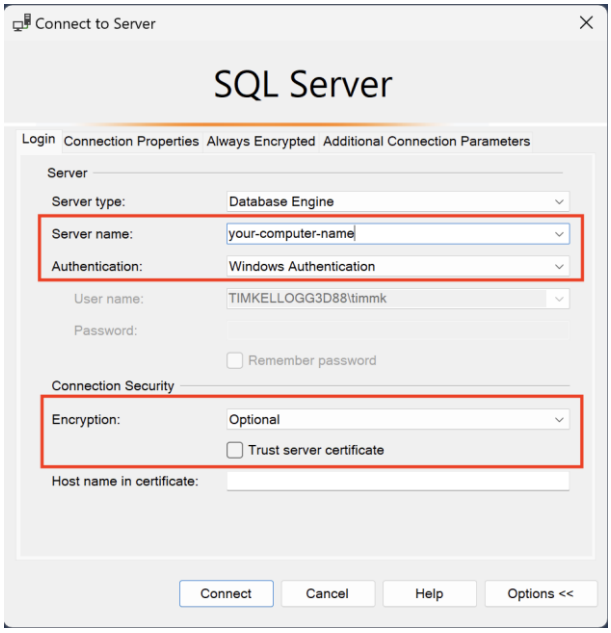
Working with SQL Server Management Studio (SSMS)

Working with SQL Server Management Studio (SSMS)

- **You can run queries against SQL databases using the SSMS**



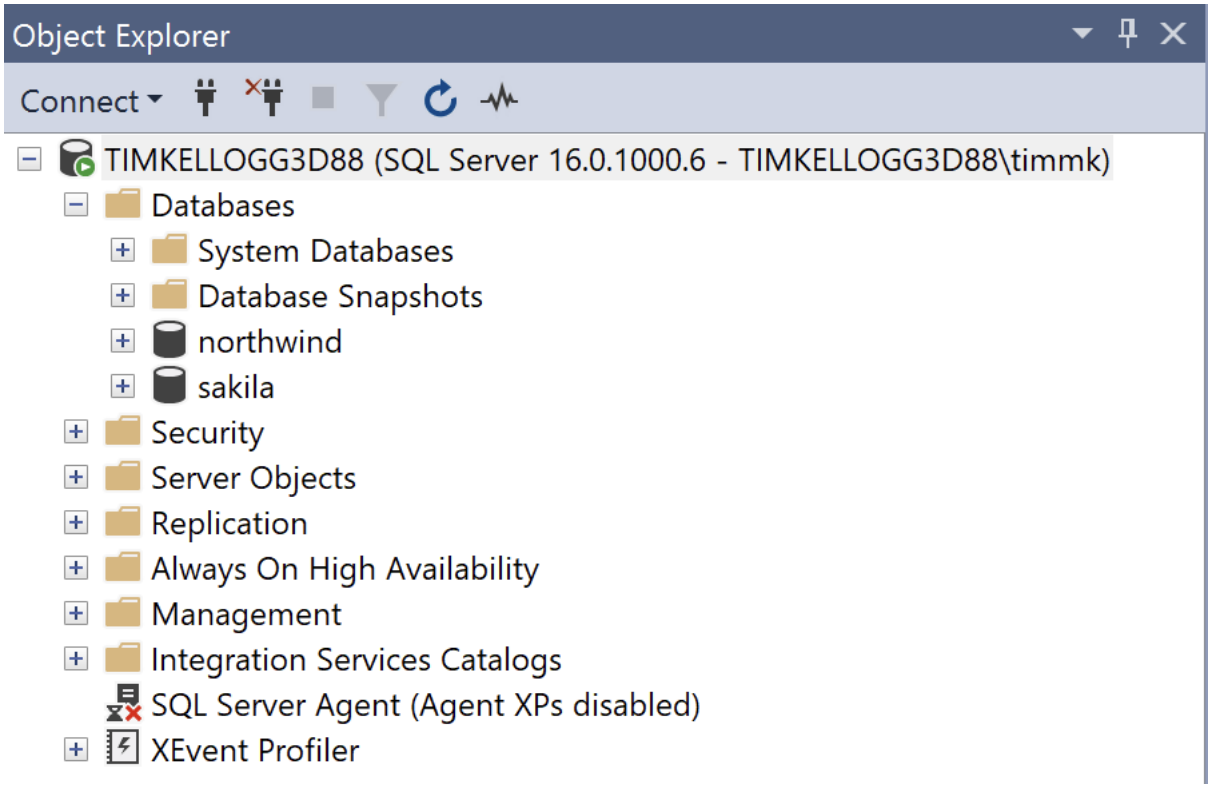
- **Once you launch it, you should select your instance**
  - We will be using Windows Authentication
  - Use your computer name as the Server name



  - Select **Optional** for the Encryption option and click **Connect**

SQL Server Object Explorer

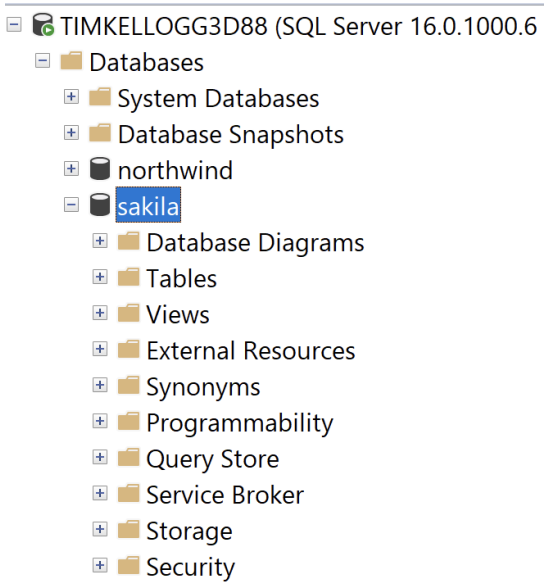
- **The Object Explorer window docked on the left of the SSMS window lets you:**
  - view and manage objects in each instance of SQL Server
  - work with your databases



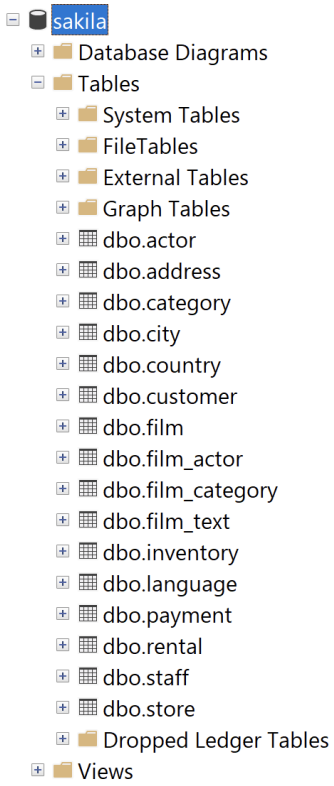
- **Expand the Server [your computer name] and Database dropdowns to see existing local databases**
  - You should have the Sakila database we created and seeded in the prior exercise
  - You will have to explicitly add other databases you want to interact with

Exploring a Database

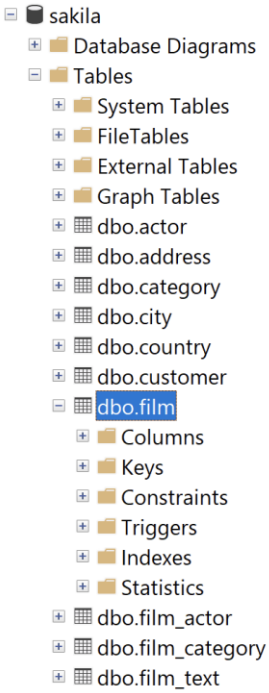
- **If you drill into a database using Object Explorer, you can:**
  - see the categories of elements in the database



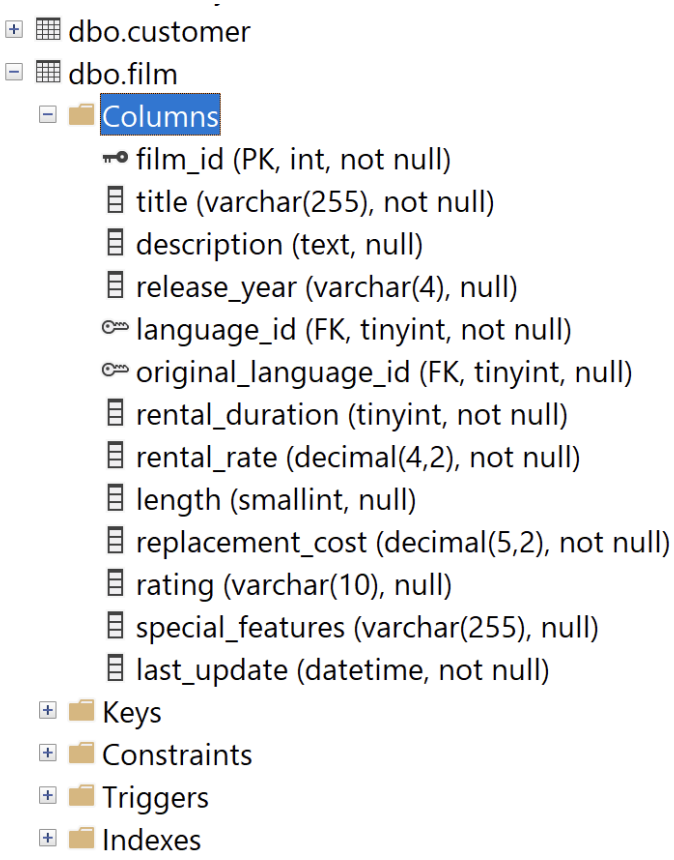
  - see all of the tables in the database



– see the characteristics of a table



– see the columns in a specific table and their types



Viewing Records

- **If you right-click on a table in the Object Explorer window, you can choose [Select Top 1000 Rows] to view data**
  - It opens the Query tab, generates a SELECT statement for the data, and runs the query

SQLQuery3.sql...88\timmk (62))

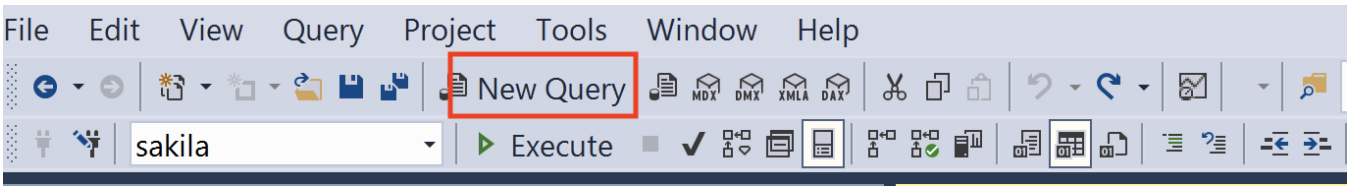
```
SELECT TOP (1000) [film_id]
      ,[title]
      ,[description]
      ,[release_year]
      ,[language_id]
      ,[original_language_id]
      ,[rental_duration]
      ,[rental_rate]
      ,[length]
      ,[replacement_cost]
      ,[rating]
      ,[special_features]
      ,[last_update]
  FROM [sakila].[dbo].[film]
```

100 %

Results Messages

	film_id	title	description	release_year	language_id	original_language_id	rental_duration	rental_rate
1	1	ACADEMY DINOSAUR	An Epic Drama of a Feminist And a Mad Scientist who ...	2006	1	NULL	6	0.99
2	2	ACE GOLDFINGER	A Astounding Epistle of a Database Administrator And ...	2006	1	NULL	3	4.99
3	3	ADAPTATION HOLES	A Astounding Reflection of a Lumberjack And a Car w...	2006	1	NULL	7	2.99
4	4	AFFAIR PREJUDICE	A Fanciful Documentary of a Frisbee And a Lumberjac...	2006	1	NULL	5	2.99
5	5	AFRICAN EGG	A Fast-Paced Documentary of a Pastry Chef And a De...	2006	1	NULL	6	2.99
6	6	AGENT TRUMAN	A Intrepid Panorama of a Robot And a Boy who must ...	2006	1	NULL	3	2.99
7	7	AIRPLANE SIERRA	A Touching Saga of a Hunter And a Butler who must D...	2006	1	NULL	6	4.99
8	8	AIRPORT POLLOCK	An Epic Tale of a Moose And a Girl who must Confront...	2006	1	NULL	6	4.99
9	9	ALABAMA DEVIL	A Thoughtful Panorama of a Database Administrator A...	2006	1	NULL	3	2.99
10	10	ALADDIN CALENDAR	A Action-Packed Tale of a Man And a Lumberjack who...	2006	1	NULL	6	4.99
11	11	ALAMO VIDEOTAPE	A Boring Epistle of a Butler And a Cat who must Fight ...	2006	1	NULL	6	0.99
12	12	ALASKA PHANTOM	A Fanciful Saga of a Hunter And a Pastry Chef who m...	2006	1	NULL	6	0.99

- **To open additional Query tabs, click on the [New Query] button at on the toolbar**



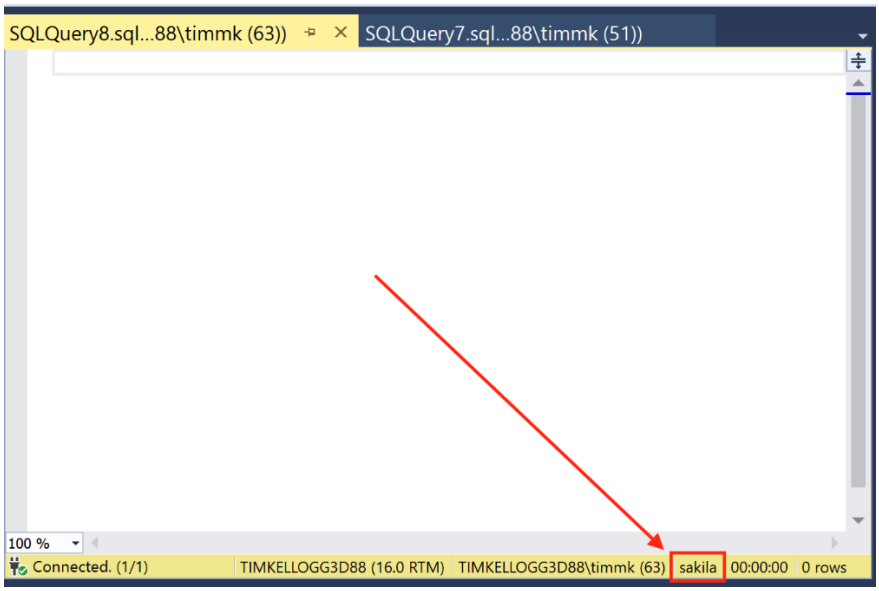
Executing Queries

- **You can enter any query you want in the Query tab and then execute it**
  - You execute it by specifying the 

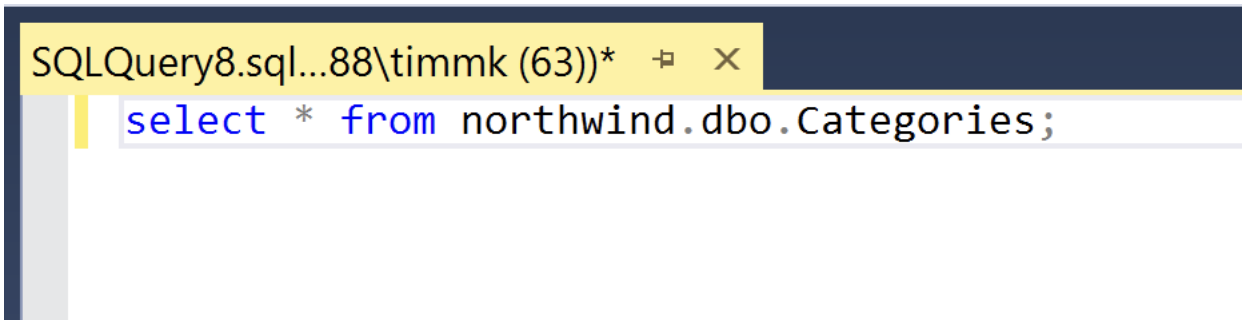
button on the Query tab just above where you enter the SQL statement

- **However, because you may have several databases available, you must select the database to run the query against**

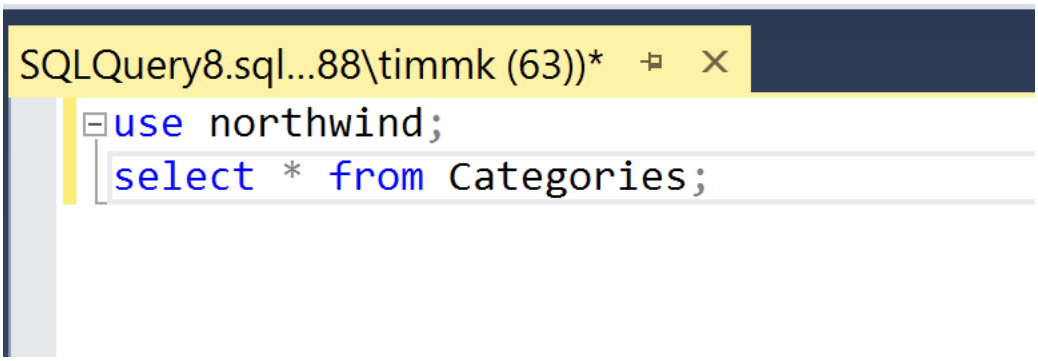
- **One way to do this is open the Query tab by right clicking the database you want to query and clicking [New Query]**
  - You can verify the database your query will run against by checking for the database name in the bottom right corner of the query tab
  - The text is small and easy to miss!



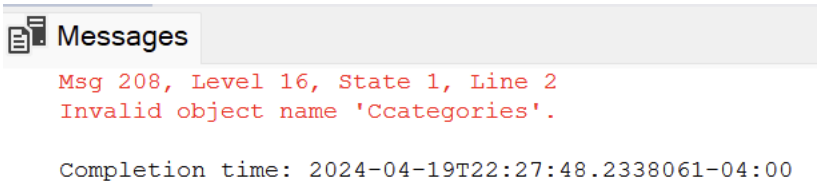
- **Another way is to specify the database is to type it and a period in front of the table name**



- **Yet another way is to place a USE statement at the top of the script specifying the database**



- **If you have a syntax error in your query, the Output window will describe the error**



- **For more information on SSMS see: https://learn.microsoft.com/en-us/sql/ssms/sql-server-management-studio-ssms**

Section 2–3

Querying Data

Using SELECT Statements to Query Data

- **Much of the time, when we interact with SQL databases, we will be looking for specific data to read**

- **In SQL, you do this with a SELECT statement**

- **The SELECT statement's minimum requirements are:**
  - SELECT (a keyword)
  - the list of columns to be displayed (or * for all columns)
  - FROM (a keyword)
  - the table to get data from

Syntax

```
SELECT column1, column2, etc
FROM table-name;
```

Syntax

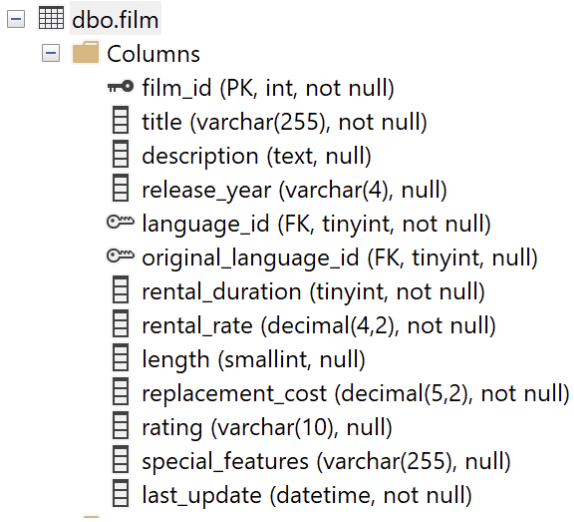
```
SELECT *
FROM table-name;
```

- **The case of the keywords SELECT and FROM isn't important**
  - It can be in uppercase, lowercase, or mixed case

- **The statement, and all other SQL statements, must end with a semicolon**

Example: Simple Select

SAKILA DATABASE: The **`film`** table is shown below:



QUERY: What are the titles of the films we sell and their ratings?

APPROACH: Use a SELECT statement, list the columns we want after the keyword SELECT, and list the film table after the keyword FROM.

Query

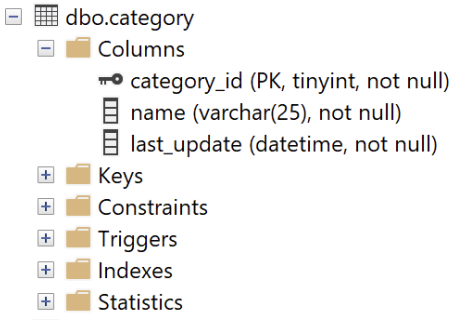
```
SELECT title, rating
FROM film
```

RESULTS:

```
+------------------------------+---------+
| title                        | rating  |
+------------------------------+---------+
| ACADEMY DINOSAUR             | PG      |
| ACE GOLDFINGER               | G       |
+------------------------------+---------+
| ... there are many rows returned ...   |
+------------------------------+---------+
```

Example: Select *

SAKILA DATABASE: The **`category`** table is shown below:



QUERY: What are the categories of films we carry?

APPROACH: Use a SELECT statement and list all columns from the category table.

Query

```
SELECT *
FROM category
```

RESULTS:

```
+-------------+----------------------+---------------------+
| category_id | name                 | last_update         |
+-------------+----------------------+---------------------+
| 1           | Action               | 2006-02-15 04:46:27 |
| 2           | Animation            | 2006-02-15 04:46:27 |
+-------------+----------------------+---------------------+
| ... there are many rows returned ...                     |
+-------------+----------------------+---------------------+
```

Optional Clauses in a SELECT Statement

- **The SELECT statement has several optional clauses, including:**
  - ORDER BY specifies how to order the returned rows
  - WHERE specifies which rows to retrieve
  - AS can be used to provide an alias for a column or expression in the SELECT
  - GROUP BY groups rows based on a specified characteristic so that an aggregate function can be applied to each group
  - HAVING specifies which groups to include in the result

ORDER BY Clause

- **ORDER BY lets you control the order of the data that is returned**
  - ORDER BY is generally the last clause in the SELECT statement
- **By default, it returns data in ascending order based on the column selected**

SYNTAX

```
SELECT column1, column2, etc
FROM table
ORDER BY any-column;
```

- **However, you can use the DESC keyword after the column name to specify rows should be returned in descending order**

SYNTAX

```
SELECT column1, column2, etc
FROM table
ORDER BY any-column DESC;
```

- **NOTE: In some situations, you may want to have more than one field you order by**
  - For example, order by state and, within state, by city

Examples: ORDER BY

QUERY: What are the films we carry ordered by the length of the film (shortest first)?

APPROACH: Use an ORDER BY on the query to sort by the length column.

Query

```
SELECT film_id, title, length
FROM film
ORDER BY length;
```

RESULTS:

```
+---------+----------------------+---------+
| film_id | title                | length  |
+---------+----------------------+---------+
| 15      | ALIEN CENTER         | 46      |
| 469     | IRON MOON            | 46      |
+---------+----------------------+---------+
| ... there are many rows returned ...      |
+---------+----------------------+---------+
```

QUERY: What are the films we carry ordered by the length of the film (longest first)?

APPROACH: Use an ORDER BY on the query to sort by the length column and use a descending sort

Query

```
SELECT film_id, title, length
FROM film
ORDER BY length DESC;
```

RESULTS:

```
+---------+----------------------+---------+
| film_id | title                | length  |
+---------+----------------------+---------+
| 141     | CHICAGO NORTH        | 185     |
| 182     | CONTROL ANTHEM       | 185     |
+---------+----------------------+---------+
```

```
| ... there are many rows returned ...      |
+---------+----------------------+---------+
```

QUERY: What are the films we carry ordered by the length of the film (longest first), but if multiple films have the same length, then order by film_id?

APPROACH: Use an ORDER BY on the query to sort by the length column and use a descending sort

Query

```
SELECT film_id, title, length
FROM film
ORDER BY length DESC, film_id;
```

RESULTS:

```
+---------+----------------------+---------+
| film_id | title                | length  |
+---------+----------------------+---------+
| 141     | CHICAGO NORTH        | 185     |
| 182     | CONTROL ANTHEM       | 185     |
+---------+----------------------+---------+
| ... there are many rows returned ...      |
+---------+----------------------+---------+
```

WHERE Clause

- **The WHERE clause limits the rows returned to those that match a specified condition**

- **The condition is specified using the comparison operators**
  = <> > >= < <=

SYNTAX

```
SELECT column1, column2, etc
FROM table
WHERE expression
```

- **You can build a complex condition using the logical operators AND, OR, and NOT**

- **You can use parentheses to control order of precedence**

Example: WHERE

QUERY: What films have a ‘PG’ rating?

APPROACH: Use a WHERE clause and specify a condition that the rating column must be ‘equal to’ the string ‘PG’.

Query

```
SELECT film_id, title, rating
FROM film
WHERE rating = 'PG';
```

RESULTS:

```
+---------+----------------------+---------+
| film_id | title                | rating  |
+---------+----------------------+---------+
| 1       | ACADEMY DINOSAUR     | PG      |
| 6       | AGENT TRUMAN         | PG      |
+---------+----------------------+---------+
| ... there are many rows that match ...    |
+---------+----------------------+---------+
```

Example: Complex WHERE

QUERY: What films have a 'PG' rating that run between 90 and 120 minutes? List the results in descending order by length. If two or more films have the same length, then list them in alphabetical order by title.

APPROACH: Use a WHERE clause and specify a condition that the rating column must be 'equal to' the string 'PG' and that the value in the rating column must be between 90 and 120.

Query

```
SELECT film_id, title, rating, length
FROM film
WHERE rating = 'PG' AND (length >= 90 AND length <= 120)
ORDER BY length DESC, title;
```

RESULTS:

```
+---------+----------------------+---------+---------+
| film_id | title                | rating  | length  |
+---------+----------------------+---------+---------+
| 477     | JAWVREAKER BROOKLYN  | PG      | 118     |
| 645     | OTHERS SOUP          | PG      | 118     |
+---------+----------------------+---------+---------+
| ... there are many rows that match ...             |
+---------+----------------------+---------+---------+
```

Case Sensitivity

- **Case sensitivity in queries depends on the SQL Server instance configuration**

- **Y can use the COLLATE keyword to perform a case-insensitive search on a case-sensitive database**

Query

```
SELECT film_id, title, rating, length
FROM film
WHERE rating = 'pg'
COLLATE SQL_Latin1_General_CP1_CI_AS;
```

RESULTS (all movies with PG rating):

```
+---------+----------------------+---------+
| film_id | title                | rating  |
+---------+----------------------+---------+
| 1       | ACADEMY DINOSAUR     | PG      |
| 6       | AGENT TRUMAN         | PG      |
+---------+----------------------+---------+
| ... there are many rows that match ...    |
+---------+----------------------+---------+
```

Using LIKE and BETWEEN For Comparisons

- **Instead of the standard relational operators in a WHERE clause, you can use LIKE or BETWEEN**

- **LIKE allows you to specify a partial string match using the percent sign (%) as a wildcard match**

Example

```
WHERE title LIKE 'SIT%'

    means the value starts with 'SIT'

WHERE title LIKE '%SIT'

    means the value ends with 'SIT'

WHERE title LIKE '%SIT%'

    means contains 'SIT' anywhere in the string
```

- **You can use the BETWEEN operator in a WHERE clause to specify a range of values**
  - Specify the minimum and maximum values separated by the word AND
    - NOTE: The values are inclusive in the range.

Example

```
WHERE length BETWEEN 89 AND 91
```

Example: LIKE

QUERY: What films titles start with the word ‘Theory’?

APPROACH: Use LIKE with a wildcard value of ‘Theory%’ in the query.

```
SELECT film_id, title, rating
FROM film
WHERE title LIKE 'THEORY%';
```

RESULTS:

```
+---------+----------------------+---------+
| film_id | title                | rating  |
+---------+----------------------+---------+
| 886     | THEORY MERMAID       | PG-13   |
+---------+----------------------+---------+
```

QUERY: What films have the phrase ‘sec’ anywhere in their title?

APPROACH: Use LIKE with a wildcard value of ‘%sec%’ in the query.

```
SELECT film_id, title, rating
FROM film
WHERE title like '%SEC%'
ORDER BY title;
```

RESULTS:

```
+---------+----------------------+---------+
| film_id | title                | rating  |
+---------+----------------------+---------+
| 231     | DINOSAUR SECRETARY   | R       |
| 461     | INSECTS STONE        | NC-17   |
+---------+----------------------+---------+
| ... there are many rows that match ...   |
+---------+----------------------+---------+
```

Example: BETWEEN

QUERY: What films have a running length in the range 89-91 minutes?

APPROACH: Use BETWEEN to specify a range of values for length.

```
SELECT film_id, title, length
FROM film
WHERE length BETWEEN 89 AND 91
ORDER BY title;
```

RESULTS:

```
+---------+----------------------+---------+
| film_id | title                | length  |
+---------+----------------------+---------+
| 28      | ANTHEM LUKE          | 91      |
| 57      | BASIC EASY           | 90      |
+---------+----------------------+---------+
| ... there are many rows that match ...    |
+---------+----------------------+---------+
```

QUERYING FOR NULL VALUES

- **Some tables will allow NULL for some fields**
  - This does complicate queries a tiny bit
- **For example:**
  - if you are querying the film table for all films that have an original_language_id of 1, films that have a NULL for that columns won't be included
  - But if you are querying for all films that DO NOT have an original_language_id of 1, films that have a NULL for that columns ***still won't be included***.
- **You must use a specific test to match a column to NULL**

Syntax

```
WHERE column-name IS NULL

WHERE column-name IS NOT NULL
```

Example: Querying with NULL

QUERY: Find all films that don't have a value for original_language_id

```
SELECT film_id, title
FROM film
WHERE original_language_id IS NULL;
```

QUERY: Find all films that do have a value for original_language_id

```
SELECT film_id, title
FROM film
WHERE original_language_id IS NOT NULL;
```

SELECT DISTINCT

- **The SELECT DISTINCT statement returns only distinct (unique) values**

- **When applied to a row, it returns a row where the totality of all the fields is different from others returned**

Syntax

```
SELECT DISTINCT column1, column2, ...
FROM table_name;
```

- **When applied to a column, it returns only unique values for the column**

Syntax

```
SELECT DISTINCT(column1)
FROM table_name;
```

Example: DISTINCT

QUERY: Find all the unique prices we rent films for

```
SELECT DISTINCT(rental_rate)
FROM film;
```

RESULTS:

```
+--------------+
| rental_rate  |
+--------------+
| 0.99         |
| 4.99         |
| 2.99         |
+--------------+
```

Comments

- **You can add both single-line and multiline comments to a SQL script**

- **Single line comments start with --**
  - The rest of the line after the -- is ignored

Example

```
--Select all films that run 1.5 hrs or less

SELECT film_id, title, length
FROM film
WHERE length <= 90;    -- 90 minutes is 1.5 hrs
```

- **Multiline comments start with /* and end with */**
  - All text between /* and */ is ignored

Example

```
/*
  Select all films that run 1.5 hrs or less
*/

SELECT film_id, title, length
FROM film
WHERE length <= 90;
```

Exercises

EXERCISE 1

If you didn't install the **sakila** database during the class demo, complete that step before moving on to **Exercise 2!**

First import the Schema to create the database and tables:
https://github.com/ivanceras/sakila/blob/master/sql-server-sakila-db/sql-server-sakila-schema.sql

Then populate the tables with data:
https://github.com/ivanceras/sakila/blob/master/sql-server-sakila-db/sql-server-sakila-insert-data.sql

EXERCISE 2

Need to practice the basic SQL syntax? Take a few minutes to do the following practice exercises:

SQL Select: https://www.w3schools.com/sql/exercise.asp?filename=exercise_select1

SQL Order By: https://www.w3schools.com/sql/exercise.asp?filename=exercise_orderby1

SQL Where: https://www.w3schools.com/sql/exercise.asp?filename=exercise_where1

SQL Like: https://www.w3schools.com/sql/exercise.asp?filename=exercise_like1

SQL Between: https://www.w3schools.com/sql/exercise.asp?filename=exercise_between1

Got that down? Try running queries again the database created in **Exercise 1**!

Exercises cont'd

EXERCISE 3

In this exercise, you will install the Northwind database and then run some simple queries against it.

We will use the Microsoft Northwind database for many of the exercises and examples in this workbook. You can find the SQL script file(s) and install instructions for SQL Server here: https://github.com/microsoft/sql-server-samples/blob/master/samples/databases/northwind-pubs/instnwnd.sql

- NOTE: There are two distinct SQL files needed to install the Northwind Database – one to create the database and tables, and another to add the seed data. Make sure you run both!

To see your new database in the Navigator window, you may have to refresh it. Right-click in the Navigator window and choose Refresh All.

Northwind is a database for a small grocery store. Take a few minutes to examine the schema. Then answer the following questions by either looking at the tables, the columns, or running a query.

NOTE: You may want to add these to a `.sql` file with comments or a txt file and save them in a GitHub repo for future reference.

You can put all SQL statements in the same script with comments in front of them and then only run the selected query by pressing the 2nd lightning bolt.

1. What is the name of the table that holds the items Northwind sells?
2. Write a query to list the product id, product name, and unit price of every product.
3. Write a query to list the product id, product name, and unit price of every product. Except this time, order then in ascending order by price.
4. What are the products that we carry where the unit price is $7.50 or less?
5. What are the products that we carry where we have at least 100 units on hand? Order them in descending order by price.

6. What are the products that we carry where we have at least 100 units on hand? Order them in descending order by price. If two or more have the same price, list those in ascending order by product name.

7. What are the products that we carry where we have no units on hand, but 1 or more units of them are on backorder? Order them by product name.

8. What is the name of the table that holds the types (categories) of the items Northwind sells?

9. Write a query that lists all the columns and all of the rows of the categories table? What is the category id of seafood?

10. Examine the Products table. How does it identify the type (category) of each item sold? Write a query to list all the seafood items we carry.

11. What are the first and last names of all the Northwind employees?

12. What employees have 'manager' in their titles?

13. List the distinct job titles in employees.

14. What employees have a salary that is between $2000 and $2500?

15. List all the information about all of Northwind's suppliers.

16. Examine the Products table. How do you know what supplier supplies each product? Write a query to list all of the items that 'Tokyo Traders' supplies to Northwind

(OPTIONAL) EXERCISE 4

The `sqlbolt.com` web site has some nice tutorials and practice exercises that build up to more complex situations.

Anytime you finish an exercise early, go there and start working your way through the exercises.

Module 3

Additional Querying Features

Section 3–1

Aggregate Functions

Aggregate Functions

- **SQL has functions that can be used to perform calculations on values from groups of rows, including**
  - The COUNT() function counts the number of occurrences of a value in a specified column
    - It can also be used to count the number of rows
  - The SUM() function adds up values in the specified column
  - The AVG() function returns the average value from a specific column
- **SQL also has functions that can be used to find the minimum or maximum value of a column**
  - The MIN() function finds the minimum value
  - The MAX() function finds the maximum value

Example: COUNT()

QUERY: How many films are in the films table?

APPROACH: Use the COUNT() function to count the number of rows in the film table.

```
SELECT COUNT(*)
FROM film;
```

RESULTS:

```
+----------+
| COUNT(*) |
+----------+
| 1000     |
+----------+
```

QUERY: How many distinct ratings are represented in the films table?

APPROACH: Use the COUNT() function combined with DISTINCT to count the number of ratings in the film table.

```
SELECT COUNT(DISTINCT(rating))
FROM film;
```

RESULTS:

```
+-------------------------+
| COUNT(DISTINCT(rating)) |
+-------------------------+
| 5                       |
+-------------------------+
```

Example: SUM()

QUERY: If I wanted to watch all the movies in the film catalog, how long would it take?

APPROACH: Use the SUM() function to add up all the length values in the films table.

```
SELECT SUM(length)
FROM film;
```

RESULTS:

```
+---------------+
| SUM(length)   |
+---------------+
| 115272        |
+---------------+
```

Example: AVG()

QUERY: What is the average cost to rent a ‘G’-rated film?

APPROACH: Use the AVG() function to find the average value in the rental_rate column of all films whose rating is ‘G’.

```
SELECT AVG(rental_rate)
FROM film
WHERE rating = 'G ';
```

RESULTS:

```
+------------------+
| AVG(rental_rate) |
+------------------+
| 2.888876         |
+------------------+
```

Example: MIN() and MAX()

QUERY: How short is the shortest film? What about the longest?

APPROACH: Use the MIN() and MAX() function to examine the length.

```
SELECT MIN(length)
FROM film;
```

RESULTS:

```
+-------------+
| MIN(length) |
+-------------+
| 46          |
+-------------+
```

```
SELECT MAX(length)
FROM film;
```

RESULTS:

```
+-------------+
| MAX(length) |
+-------------+
| 185         |
+-------------+
```

Section 3–2

Grouping Results

GROUP BY clause

- **The GROUP BY clause allows you to execute aggregate functions on 'groups' of data created by the query**
  - GROUP BY specifies the column(s) used to create the groups
  - The aggregate functions return a value for EACH group created

Syntax

```
SELECT column1, column2, etc
FROM table_name
WHERE condition
GROUP BY column_name(s)
ORDER BY column_name(s);
```

Example: GROUP BY

QUERY: How many movies are available broken down by rating (G, PG, PG-13, etc.)?

APPROACH: Use the GROUP BY clause to create groups of films by rating and then use the COUNT() function to count the number of rows in each group.

```
SELECT rating, COUNT(*)
FROM film
GROUP BY rating;
```

RESULTS:

```
+----------+----------+
| rating   | COUNT(*) |
+----------+----------+
| PG       | 194      |
| G        | 178      |
| NC-17    | 210      |
| PG-13    | 223      |
| R        | 195      |
+----------+----------+
```

QUERY: What is the average price to rent a movie broken down by rating (G, PG, PG-13, etc.)??

APPROACH: Use the GROUP BY clause to create groups of films by rating and then use the AVG() function to calculate the average rental_rate of rows in each group.

```
SELECT rating, avg(rental_rate)
FROM film
GROUP BY rating;
```

RESULTS:

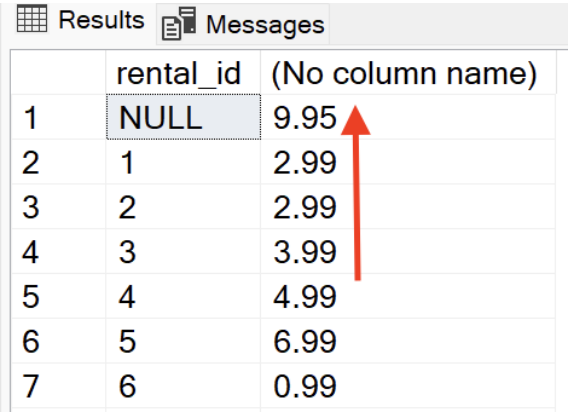
```
+----------+------------------+
| rating   | AVG(rental_rate) |
+----------+------------------+
| PG       | 3.051856         |
| G        | 2.888876         |
| NC-17    | 2.970952         |
| PG-13    | 3.034843         |
| R        | 2.9387818        |
+----------+------------------+
```

Renaming Computed Fields - Using AS

- **Computed fields don't have an official name in a SQL query**

Example

```
SELECT rental_id, SUM(amount)
FROM payment
GROUP BY rental_id
ORDER BY rental_id;
```


Results Messages

	rental_id	(No column name)
1	NULL	9.95
2	1	2.99
3	2	2.99
4	3	3.99
5	4	4.99
6	5	6.99
7	6	0.99

- **This can be a problem if you want to use it to order the results**

- **SQL provides the AS keyword to create an alias for the column name**
  - You can use it for ordering or other purposes

Example

```
SELECT rental_id, SUM(amount) AS total_amount
FROM payment
GROUP BY rental_id
ORDER BY rental_id;
```

  - AS can also create an alias for a table name; we will see this in the next module

Example: AS keyword

QUERY: What is the average price to rent a movie broken down by rating (G, PG, PG-13, etc.) and displayed in ascending order by average price?

APPROACH: Use the GROUP BY clause to create groups of films by rating and then use the `AVG()` function to calculate the average rental_rate of rows in each group. Make sure to name the value returned by the `AVG()` function so that we can use it in the ORDER BY clause.

```
SELECT rating, AVG(rental_rate) AS avg_rate
FROM film
GROUP BY rating
ORDER BY avg_rate;
```

RESULTS:

```
+----------+------------+
| rating   | avg_rate   |
+----------+------------+
| G        | 2.888876   |
| R        | 2.938781   |
| NC-17    | 2.970952   |
| PG-13    | 3.034843   |
| PG       | 3.051856   |
+----------+------------+
```

HAVING clause

- **The HAVING clause is used with the GROUP BY clause**
- **It allows you to include only those groups that meet a specified condition**

Syntax

```
SELECT column1, columns2, etc
FROM table_name
WHERE row-condition
GROUP BY column_name(s)
HAVING group-condition
ORDER BY column_name(s);
```

- **If the WHERE clause is used to select rows, the HAVING clause is used to select groups**

Example: HAVING

QUERY: What is the average rating for movies broken down by rating (G, PG, PG-13, etc.)? NOTE: I'm not interested in the rating if there are less than 200 films in the group.

APPROACH: Use the GROUP BY clause to create groups of films by rating and then use the COUNT() function to count the number rows in each group. Only display the groups that have at least 200 rows.

```
SELECT rating, COUNT(*)
FROM film
GROUP BY rating
HAVING COUNT(*) >= 200
ORDER BY rating;
```

RESULTS:

```
+----------+-----------+
| rating   | COUNT(*)  |
+----------+-----------+
| NC-17    | 210       |
| PG-13    | 223       |
+----------+-----------+
```

Exercises

EXERCISE 1

Let's look back at W3Schools again to test ourselves with aggregate functions and GROUP BY. Take a few minutes to do the following practice exercises:

SQL Functions: https://www.w3schools.com/sql/exercise.asp?filename=exercise_functions1

SQL Group By: https://www.w3schools.com/sql/exercise.asp?filename=exercise_groupby1

EXERCISE 2

In this exercise, will continue to execute queries against the Northwind database.

NOTE: You may want to add these to a `.sql` file with comments or a txt file and save them in a GitHub repo for future reference.

1. How many suppliers are there? Use a query!
2. What is the sum of all the employee's salaries?
3. What is the price of the cheapest item that Northwind sells?
4. What is the average price of items that Northwind sells?
5. What is the price of the most expensive item that Northwind sells?
6. What is the supplier ID of each supplier and the number of items they supply? You can answer this query by only looking at the Products table.
7. What is the category ID of each category and the average price of each item in the category? You can answer this query by only looking at the Products table.
8. For suppliers that provide at least 5 items to Northwind, what is the supplier ID of each supplier and the number of items they supply? You can answer this query by only looking at the Products table.
9. List the product id, product name, and inventory value (calculated by multiplying unit price by the number of units on hand). Sort the results in descending order by value. If two or more have the same value, order by product name.

(OPTIONAL) EXERCISE 3

The `sqlbolt.com` web site has some nice tutorial and practice exercises that build up to more complex situations.

Anytime you finish an exercise early, go there and start working your way through the exercises.

Section 3–3

Nested Queries

Nested Queries (Subselects)

- **Sometimes, you need to do a select to get one result and then use that result in another SELECT statement's WHERE clause**
  - This is often called a subselect, inner query or nested query
- **When you write subqueries, they must be enclosed within parentheses**
  - They can't have an ORDER BY clause
- **A subquery usually has only one column in the SELECT clause**
  - Subqueries that return more than one row can only be used with multiple value operators such as the IN operator.

Syntax

```
SELECT column1, column2, etc
FROM   table
WHERE  column_name OPERATOR (SELECT column_name
                             FROM table1
                             WHERE condition)
GROUP BY column_name
HAVING group-condition
ORDER BY column;
```

Example: Nested Query

QUERY: Which film(s) are the most expensive to replace?

APPROACH: Use the SQL `max()` function to find the largest replacement_cost in the film table, and then use that maximum cost in a different query to select the film(s) that have that replacement cost.

```
SELECT film_id, title, replacement_cost
FROM film
WHERE replacement_cost = (SELECT MAX(replacement_cost)
                          FROM film);
```

RESULTS:

```
+---------+--------------------------+-------------------+
| film_id | title                    | replacement_cost  |

+---------+--------------------------+-------------------+
| 34      | ARABIA DOGMA             | 29.99             |

| 52      | BALLROOM MOCKINGBIRD     | 29.99             |
+---------+--------------------------+-------------------+
| ... there were many, many rows that matched ...        |
+---------+--------------------------+-------------------+
```

QUERY: Which film(s) are described as documentaries and how long do they run?

APPROACH: If we research the film_text table in the sakila database, we find it contains 3 columns named film_id, title, and description. We can run a query to find the films that have 'documentary' in their descriptions. But the length of the film isn't available in film_text.

In the solution below, we keep the film_id values of the query that searches for documentaries and then use ANOTHER query against the film table to find all films in that 1st query's film_id list. Note that the where uses the keyword 'in' rather than an '=' to match film_id values.

```
SELECT title, length
FROM film
WHERE film_id IN (SELECT film_id
                  FROM film
                  WHERE description LIKE '%Documentary%');
```

RESULTS:

```
+--------------------------+--------+
| title                    | length |
+--------------------------+--------+
| AFFAIR PREJUDICE         | 117    |
| AFRICAN EGG              | 130    |
+--------------------------+--------+
| ... many rows matched ...         |
+--------------------------+--------+
```

Exercises

EXERCISE 1

In this exercise, will continue to execute queries against the Northwind database.

NOTE: You may want to add these to a `.sql` file with comments or a txt file and save them in a GitHub repo for future reference.

1. What is the product name(s) of the most expensive products? HINT: Find the max price in a subquery and then use that value to find products whose price equals that value.

2. What is the order id, shipping name and shipping address of all orders shipped via 'Federal Shipping'? HINT: Find the shipper id of 'Federal Shipping' in a subquery and then use that value to find the orders that used that shipper.

3. What are the order ids of the orders that ordered 'Sasquatch Ale'? HINT: Find the product id of 'Sasquatch Ale' in a subquery and then use that value to find the matching orders from the `order details` table. Because the `order details` table has a space in its name, you will need to surround it with back ticks in the FROM clause.

4. What is the name of the employee that sold order 10266?

5. What is the name of the customer that bought order 10266?

(OPTIONAL) EXERCISE 2

The `sqlbolt.com` web site has some nice tutorial and practice exercises that build up to more complex situations.

Anytime you finish an exercise early, go there and start working your way through the exercises.

Module 4

Querying Multiple Tables (JOINS)

Section 4–1

Querying Multiple Tables (JOINS)

Querying Multiple Tables (JOINS)

- **Sometimes the information you want must be created by combining data in more than one table**
  - In this case, SQL provides a way for us to ***join*** two or more tables together to get the information we want

- **A join is where you take a row from one table and *join* it to a row in another table based on some condition**
  - The condition is usually matching a foreign key to a primary key

employee

id	first_name	last_name	pay	pay_grade_id
100001	Greg	Smith	32000.00	1
100002	Cindy	Jones	49000.00	3
100003	Nick	Schwartz	41000.00	2
100004	Ken	McCaskill	38000.00	2

pay_grade

id	description	min_val	max_val
1	ES05 Pay Grade	22000.00	37000.00
2	ES10 Pay Grade	33000.00	46000.00
3	ES15 Pay Grade	39000.00	57000.00
4	ES20 Pay Grade	52000.00	75000.00

- **The result set is 'new' rows containing columns from all the tables in the join**

- **There are several different types of JOIN operations, including:**
  - INNER JOIN (most common)
  - OUTER JOIN

Inner Joins

- **With an INNER JOIN, a row in one table is joined with a row in another table based on a column match**

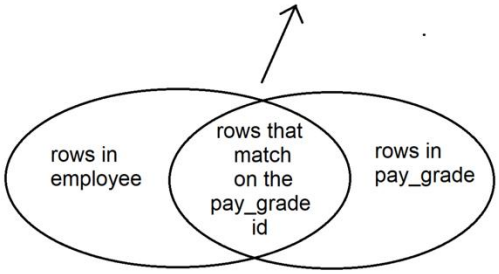
- **Only matched rows will be included in the result**
  - If a row in the first table doesn't match any rows in the second table, it will be excluded
  - Similarly, if a row in the second table doesn't match a row in the first table, it will be excluded

employee

id	first_name	last_name	pay	pay_grade_id
100001	Greg	Smith	32000.00	1
100002	Cindy	Jones	49000.00	3
100003	Nick	Schwartz	41000.00	2
100004	Ken	McCaskill	38000.00	2

pay_grade

id	description	min_val	max_val
1	ES05 Pay Grade	22000.00	37000.00
2	ES10 Pay Grade	33000.00	46000.00
3	ES15 Pay Grade	39000.00	57000.00
4	ES20 Pay Grade	52000.00	75000.00



- **The syntax for doing an INNER JOIN is a bit more complex than a simple SELECT because you must specify both tables and the join condition**

Syntax

```
SELECT column1, columns2, ...
FROM table1
INNER JOIN table2
      ON table1.column-name = table2.column-name;
```

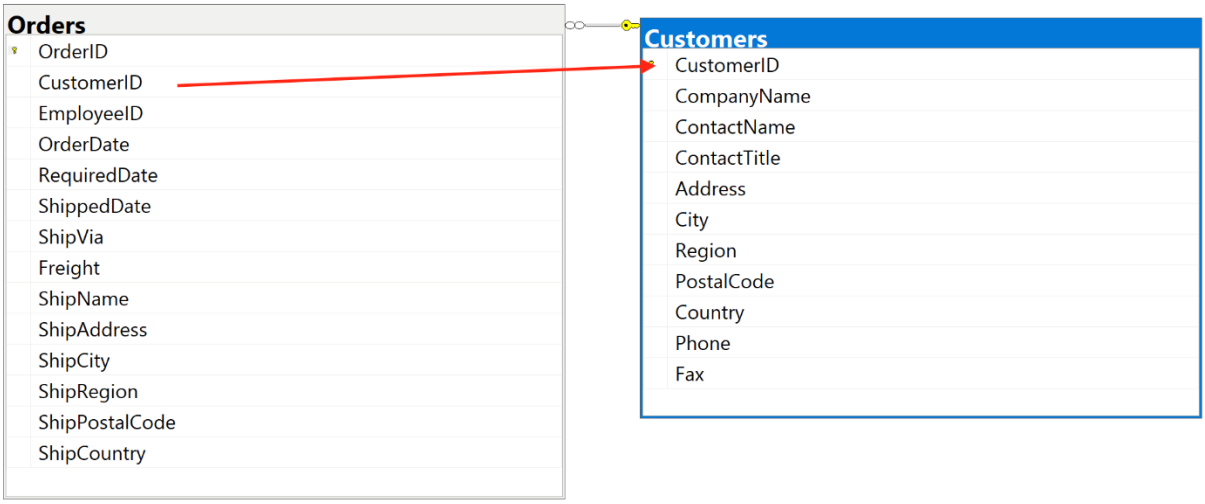
- **You don't have to include the word INNER**
  - A JOIN is considered to be an INNER JOIN unless specifically written as one of the other types you will learn about

Syntax

```
SELECT column1, columns2, ...
FROM table1
JOIN table2
  ON table1.column-name = table2.column-name;
```

Example: INNER JOIN

QUERY: We want to list each order, along the name of the customer (Company) that made the order.



APPROACH: Retrieve orders along with the name of the customer made them, joining the Orders table with the Customers table based on the common CustomerID column

```
SELECT Employees.EmployeeID, Employees.FirstName, Employees.LastName, Orders.OrderID, Order
s.OrderDate

FROM Employees

INNER JOIN Orders ON Employees.EmployeeID = Orders.EmployeeID;
```

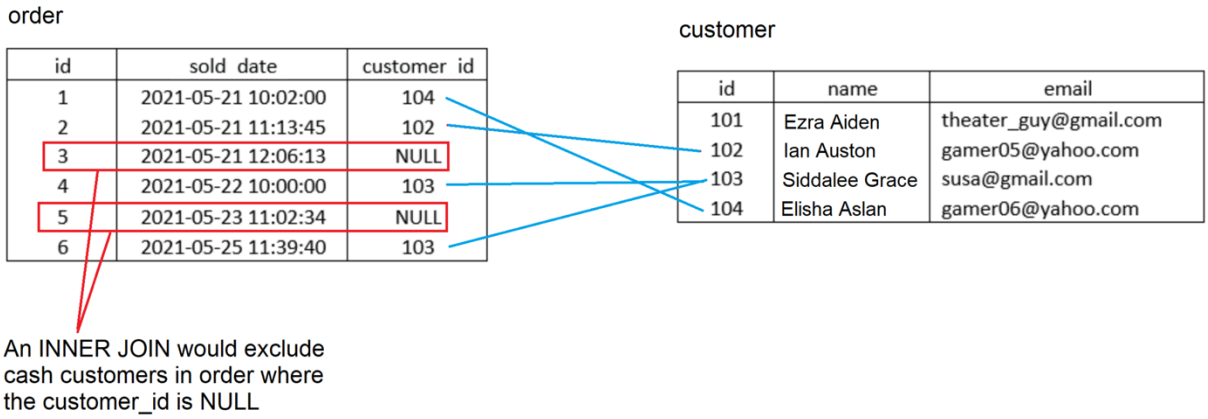
RESULTS:

```
+----------+--------------------------+---------------------+
| OrderID  |      CompanyName         |      OrderDate      |
+----------+--------------------------+---------------------+
|  10248   | Vins et alcools Chevalier| 1996-07-04 00:00:00 |
|  10249   | Toms Spezialitäten       | 1996-07-05 00:00:00 |
|  10250   | Hanari Carnes            | 1996-07-08 00:00:00 |
|  10251   | Victuailles en stock     | 1996-07-08 00:00:00 |
+----------+--------------------------+---------------------+
  … many more records!
```

We could have added the `WHERE, ORDER BY, GROUP BY`, and `HAVING` clauses if we wanted. And although this example does a JOIN on two tables, you can JOIN as many tables as you need to by continuing to add additional JOIN clauses.

Beyond Inner Joins

- **The INNER JOIN puts data together from multiple tables based on common values in the tables.**
  - A row is ONLY created in the result set when a row in one table and a row in the other table match a specified condition
- **But this may not give us what we need**
- **For example, consider a commercial business**
  - A company take orders from customers
  - When an order is created for a customer, the customer id is stored in the order to identify which customer the order is for.
  - However, the company also allows cash orders and doesn't have a customer id for those orders
    - This means the order table must allow NULLs
- **If we try to join the order table to the customer table, these orders would be excluded**



```
SELECT order.id, sold_date, name, email
FROM `order`
JOIN customer
  ON order.customer_id = customer.id;
```

RESULTS:

```
+-----+---------------------+-----------------+-------------------+
| id  | sold_date           | name            | email             |
+-----+---------------------+-----------------+-------------------+
| 1   | 2021-05-21 10:02:00 | Elisha Aslan    | gamer06@yahoo.com |
| 2   | 2021-05-21 11:13:45 | Ian Auston      | gamer05@yahoo.com |
| 4   | 2021-05-22 10:00:00 | Siddalee Grace  | susa@gmail.com    |
| 6   | 2021-05-25 11:39:40 | Siddalee Grace  | susa@gmail.com    |
+-----+---------------------+-----------------+-------------------+
```

- **Orders where the customer_id is NULL do NOT show, but we wanted all orders!**

Outer Joins

- **An OUTER JOIN is used when might want rows in one table that don't match rows in the other table**
  - You must specify which of the two tables isn't required to have matching data in its JOIN column
- **You do this by creating a LEFT OUTER JOIN or a RIGHT OUTER JOIN**
  - The first table listed in the JOIN is considered to be on the left and the second is considered to be on the right
- **For a LEFT OUTER JOIN, the first table doesn't require matching data to be included**
  - Selected columns without a matching value will be NULL
- **For a RIGHT OUTER JOIN, the second table doesn't require matching data to be included**
  - Selected columns without a matching value will be NULL
- **NOTE: the word OUTER isn't required but is often included for readability**

Example: Outer Joins

order

id	sold_date	customer_id
1	2021-05-21 10:02:00	104
2	2021-05-21 11:13:45	102
3	2021-05-21 12:06:13	NULL
4	2021-05-22 10:00:00	103
5	2021-05-23 11:02:34	NULL
6	2021-05-25 11:39:40	103

orders 3 and 5 are included in a LEFT OUTER JOIN

customer

id	name	email
101	Ezra Aiden	theater_guy@gmail.com
102	Ian Auston	gamer05@yahoo.com
103	Siddalee Grace	susa@gmail.com
104	Elisha Aslan	gamer06@yahoo.com

QUERY: What orders were sold when?

```
SELECT order.id, sold_date, name, email
FROM order
LEFT JOIN customer
       ON order.customer_id = customer.id;
```

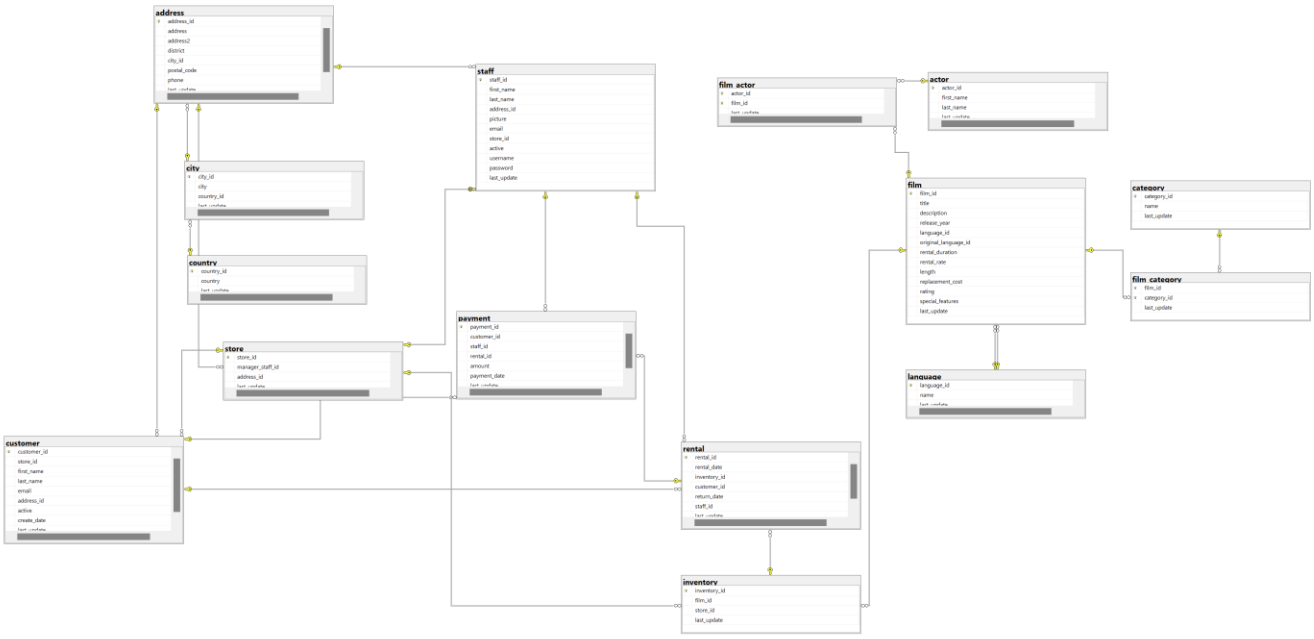
RESULTS:

```
+-----+---------------------+------------------+--------------------+
| id  | sold_date           | name             | email              |
+-----+---------------------+------------------+--------------------+
| 1   | 2021-05-21 10:02:00 | Elisha Aslan     | gamer06@yahoo.com  |
| 2   | 2021-05-21 11:13:45 | Ian Auston       | gamer05@yahoo.com  |
| 3   | 2021-05-21 12:06:13 | NULL             | NULL               |
| 4   | 2021-05-22 10:00:00 | Siddalee Grace   | susa@gmail.com     |
| 5   | 2021-05-23 11:02:34 | NULL             | NULL               |
| 6   | 2021-05-25 11:39:40 | Siddalee Grace   | susa@gmail.com     |
+-----+---------------------+------------------+--------------------+
```

Viewing the Database

- **When there are many tables involved and your queries are complex, it can be helpful to have a visual representation of the database**

- **You can use Object Explorer create database diagrams that help graphically show the structure of your database**



- **See the following for the process to create one:** https://learn.microsoft.com/en-us/sql/ssms/visual-db-tools/create-a-new-database-diagram-visual-database-tools

Exercises

EXERCISE 1

Let's start by checking out the exercises at W3Schools.

SQL Join: https://www.w3schools.com/sql/exercise.asp?filename=exercise_join1

EXERCISE 2

Now take a few minutes to look at this great visual diagram of the different types of joins:

`https://www.codeproject.com/Articles/33052/Visual-Representation-of-SQL-Joins`

EXERCISE 3

Let's continue working with Northwind.

NOTE: You may want to add these to a `.sql` file with comments or a txt file and save them in a GitHub repo for future reference.

1. List the product id, product name, unit price and category name of all products. Order by category name and within that, by product name.
2. List the product id, product name, unit price and supplier name of all products that cost more than $75. Order by product name.
3. List the product id, product name, unit price, category name, and supplier name of every product. Order by product name.
4. What is the product name(s) and categories of the most expensive products? HINT: Find the max price in a subquery and then use that in your more complex query that joins products with categories.
5. List the order id, ship name, ship address, and shipping company name of every order that shipped to Germany.
6. List the order id, order date, ship name, ship address of all orders that ordered 'Sasquatch Ale'?

(OPTIONAL) EXERCISE 4

The `sqlbolt.com` web site has some nice tutorial and practice exercises that build up to more complex situations.

Anytime you finish an exercise early, go there and start working your way through the exercises.

Module 5

Project:

Querying the sakila Database

Section 5–1

Project Description

Project Description

Answer the following questions by creating SQL queries that run against the sakila database. Below the query is the expected result that you should get back.

Use Google, Stack Overflow, etc. to figure out how to write your query. Then, use SSMS to test your queries.

Exercises

1. Display the first and last name of each actor in a single column in upper case letters. Name the column Actor Name.

Result set

Actor Name
PENELOPE GUINESS
NICK WAHLBERG
ED CHASE
JENNIFER DAVIS
JOHNNY LOLLOBRIGIDA
BETTE NICHOLSON
GRACE MOSTEL
MATTHEW JOHANSSON
JOE SWANK
CHRISTIAN GABLE

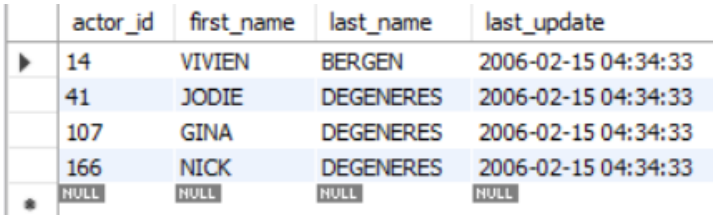
2. You need to find the ID number, first name, and last name of an actor, of whom you know only the first name, 'Joe.'

Result set

actor_id	first_name	last_name
9	JOE	SWANK
NULL	NULL	NULL

3. Find all actors whose last name contain the letters GEN.

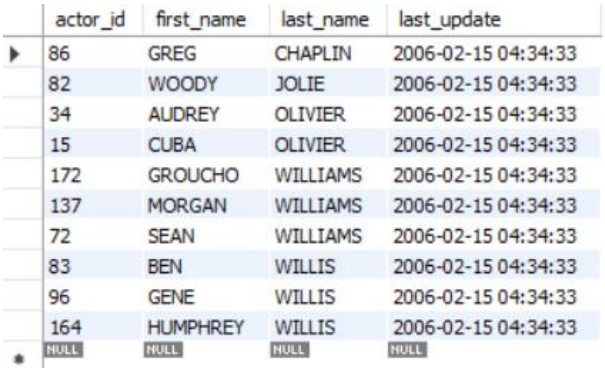
Result set



actor_id	first_name	last_name	last_update
14	VIVIEN	BERGEN	2006-02-15 04:34:33
41	JODIE	DEGENERES	2006-02-15 04:34:33
107	GINA	DEGENERES	2006-02-15 04:34:33
166	NICK	DEGENERES	2006-02-15 04:34:33
NULL	NULL	NULL	NULL

4. Find all actors whose last names contain the letters 'LI'. This time, order the rows by last name and first name, in that order.

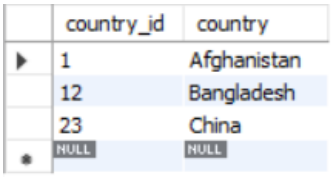
Result set



actor_id	first_name	last_name	last_update
86	GREG	CHAPLIN	2006-02-15 04:34:33
82	WOODY	JOLIE	2006-02-15 04:34:33
34	AUDREY	OLIVIER	2006-02-15 04:34:33
15	CUBA	OLIVIER	2006-02-15 04:34:33
172	GROUCHO	WILLIAMS	2006-02-15 04:34:33
137	MORGAN	WILLIAMS	2006-02-15 04:34:33
72	SEAN	WILLIAMS	2006-02-15 04:34:33
83	BEN	WILLIS	2006-02-15 04:34:33
96	GENE	WILLIS	2006-02-15 04:34:33
164	HUMPHREY	WILLIS	2006-02-15 04:34:33
NULL	NULL	NULL	NULL

5. Using IN, display the country_id and country columns of the following countries: Afghanistan, Bangladesh, and China.

Result set



country_id	country
1	Afghanistan
12	Bangladesh
23	China
NULL	NULL

6. List last names of actors and the number of actors who have that last name, but only for names that are shared by at least two actors

Result set

last_name	actor_count
KILMER	5
NOLTE	4
TEMPLE	4
AKROYD	3
ALLEN	3
BERRY	3
DAVIS	3
DEGENERES	3
GARLAND	3
GUINESS	3
HARRIS	3
HOFFMAN	3

7. The actor HARPO WILLIAMS was accidentally entered in the actor table as GROUCHO WILLIAMS. Write a query to fix the record, and another to verify the change.

Result set

actor_id	first_name	last_name	last_update
72	SEAN	WILLIAMS	2006-02-15 04:34:33
137	MORGAN	WILLIAMS	2006-02-15 04:34:33
172	HARPO	WILLIAMS	2021-06-11 12:13:11
NULL	NULL	NULL	NULL

8. Perhaps we were too hasty in changing GROUCHO to HARPO. It turns out that GROUCHO was the correct name after all! In a single query, if the first name of the actor is currently HARPO, change it to GROUCHO. Then write a query to verify your change.

Result set

actor_id	first_name	last_name	last_update
72	SEAN	WILLIAMS	2006-02-15 04:34:33
137	MORGAN	WILLIAMS	2006-02-15 04:34:33
172	HARPO	WILLIAMS	2021-06-11 12:13:11
NULL	NULL	NULL	NULL

9. Use JOIN to display the total amount rung up by each staff member in August of 2005. Use tables staff and payment.

Result set

first_name	last_name	sum(pay.amount)
Mike	Hillyer	11853.65
Jon	Stephens	12218.48

10. List each film and the number of actors who are listed for that film. Use tables film_actor and film. Use inner join.

Result set

title	number_of_actors
LAMBS CINCINATTI	15
BOONDOCK BALLROOM	13
CHITTY LOCK	13
CRAZY HOME	13
DRACULA CRYSTAL	13
MUMMY CREATURES	13
RANDOM GO	13
ARABIA DOGMA	12
HELLFIGHTERS SIERRA	12
LESSON CLEOPATRA	12
LONELY ELEPHANT	12
SKY MIRACLE	12

11. How many copies of the film HUNCHBACK IMPOSSIBLE exist in the system?

Result set

title	number_in_inventory
HUNCHBACK IMPOSSIBLE	6

12. The music of Queen and Kris Kristofferson have seen an unlikely resurgence. As an unintended consequence, films starting with the letters K and Q have also soared in popularity. Use **subqueries** to display the titles of movies starting with the letters K and Q whose language is English.

Result set

title
KANE EXORCIST
KARATE MOON
KENTUCKIAN GIANT
KICK SAVANNAH
KILL BROTHERHOOD
KILLER INNOCENT
KING EVOLUTION
KISS GLORY
KISSING DOLLS
KNOCK WARLOCK
KRAMER CHOCOLATE
KWAI HOMEWARD

13. Insert a record to represent Mary Smith renting the movie ACADEMY DINOSAUR from Mike Hillyer at Store 1 today. Then write a query to capture the exact row you entered into the rental table.

Result set (your rental date value will of course show the date and time you entered the record)

rental_id	rental_date	inventory_id	customer_id	return_date	staff_id	last_update
16050	2021-06-11 12:39:20	1	1	NULL	1	2021-06-11 12:39:20
NULL	NULL	NULL	NULL	NULL	NULL	NULL

Module 6

Modifying the Data and the Database

Section 6–1

Inserting, Updating and Deleting Data

Inserting, Updating and Deleting Data

- **Up until now, most of the SQL we have shown has been used to query data**
- **However, you often need to make changes to the data in a database**
- **You do this using the SQL statements INSERT, UPDATE, and DELETE**

INSERT INTO Statement

- **If you want to add a new row to a table, you use the INSERT INTO statement.**

- **There are two possible techniques:**
  - In the first, you specify the table name and the values of the rows to be added
  - In the second, you specify the table's column names and the values of the rows to be added
    - In this version, you could omit some columns and they would become NULL

Syntax

```
INSERT INTO table-name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

Syntax

```
INSERT INTO table-name
VALUES (value1, value2, value3, ...);
```

- **If you attempt to add a record with NULL in a non-nullable column, the INSERT statement will fail**

Example: INSERT INTO

TASK: Add a new country to the sakila country table

STATEMENT (option 1):

```
INSERT INTO country(country, last_update)
VALUES('Zimbabwe', CURRENT_TIMESTAMP);
```

STATEMENT (option 2):

```
INSERT INTO country
VALUES('Zimbabwe', CURRENT_TIMESTAMP);
```

UPDATE Statement

- **Use the UPDATE statement to edit rows in a table**
  - It allows you to change the value in one or more columns of a row
- **Use the WHERE clause to specify which row(s) will have the changes made to them**

Syntax

```
UPDATE table-name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

- ***Be careful with the UPDATE statement. If you forget the WHERE clause, all the records in that table will be updated!***

Example: UPDATE

TASK: Change the first and last name for the customer whose customer_id is 2.

```
UPDATE customer
SET first_name = 'PATTY', last_name = 'JOHNSTON'
WHERE customer_id = 2;
```

TASK: Change all PATTY first names to PATRICE.

```
UPDATE customer
SET first_name = 'PATRICE'
WHERE first_name = 'PATTY';
```

DELETE Statement

- **To delete one or more rows from a table, use the DELETE statement**

Syntax

```
DELETE FROM table-name
WHERE condition;
```

- ***Be careful with the UPDATE statement. If you forget the WHERE clause, all the records in that table will be updated!***

- **Deleting a record is a precarious action because the row you want to delete often is referenced in many tables!**
  - For example, if you try to delete a specific film from the film table, you will now have a reference in inventory to a film_id that doesn't exist

- **NOTE: Many companies don't actually delete records… Instead, they have a column whose name might be 'active' or something similar**
  - You set active to true when the row is created and mark it 'deleted' by using the UPDATE statement to set active to false.
  - This will allow the database to maintain referential integrity between related tables.

Example: DELETE

TASK: Delete all references to the payment whose payment_id is 100

```
DELETE FROM payment
WHERE payment_id = 100;
```

Exercises

EXERCISE 1

Let's start by checking out the exercises at W3Schools.

SQL Insert: https://www.w3schools.com/sql/exercise.asp?filename=exercise_insert1

SQL Update: https://www.w3schools.com/sql/exercise.asp?filename=exercise_update1

SQL Delete: https://www.w3schools.com/sql/exercise.asp?filename=exercise_delete1

EXERCISE 2

Let's continue working with Northwind.

NOTE: You may want to add these to a `.sql` file with comments or a txt file and save them in a GitHub repo for future reference.

1. Add a new supplier.
2. Add a new product provided by that supplier.
3. List all products and their suppliers.
4. Raise the price of your new product by 15%.
5. List the products and prices of all products from that supplier.
6. Delete the new product.
7. Delete the new supplier.
8. List all products.
9. List all suppliers.

(OPTIONAL) EXERCISE 3

The `sqlbolt.com` web site has some nice tutorial and practice exercises that build up to more complex situations.

Anytime you finish an exercise early, go there and start working your way through the exercises.

Section 6–2

Creating a Database and Modifying the Schema

Creating a Database

- **You can create a database in SQL Server**
- **The CREATE DATABASE statement is used to create the database**

Syntax

```
CREATE DATABASE database_name;
```

Example: CREATE DATABASE

TASK: Create a new database to manage a collection of books

```
CREATE DATABASE bookshelf;
```

Creating a New Table

- **You can also create tables using SQL**

- **The CREATE TABLE statement is used to create a table**

Syntax

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
   ....
);
```

- **A list of MySQL data types can be found here:**

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

- **You can mark fields as not being nullable**

Syntax

```
CREATE TABLE table_name (
    column1 datatype PRIMARY KEY,
    column2 datatype,
    column3 datatype NOT NULL,
    ....
);
```

- **You can mark a field as a primary key using PRIMARY KEY**

Syntax

```
CREATE TABLE table_name (
    column1 datatype PRIMARY KEY,
    column2 datatype,
    column3 datatype,
    ....
);
```

- **You can also use IDENTITY(start,increment) mark fields as being an auto-increment field**
  - This happens frequently when you don't want the user to be responsible for generating unique primary keys
  - start represents the starting value, and increment specifies the value added to the previous value when a new row is created

Syntax

```
CREATE TABLE table_name (
    column1 datatype IDENTITY(1,1) PRIMARY KEY,
    column2 datatype,
    column3 datatype NOT NULL,
    ....
);
```

- **You can add constraints**
  - A constraint is a rule that is enforced by the database when data is manipulated
  - There are several types of constraints, including the PRIMARY KEY one above

- **The UNIQUE constraint ensures that the table has unique values for a single columns or a combination of columns**

Syntax

```
CREATE TABLE table_name (
    column1 datatype AUTO_INCREMENT PRIMARY KEY,
    column2 datatype,
    column3 datatype NOT NULL,
    ....
    CONSTRAINT constraint-name
    UNIQUE (column2, ...)
);
```

- **The CHECK constraint ensures value(s) meet a specified condition**

Syntax

```
CREATE TABLE table_name (
    column1 datatype IDENTITY(1,1) PRIMARY KEY,
    column2 datatype,
    column3 datatype NOT NULL,
    column4 datatype NOT NULL,
    ....
    CONSTRAINT constraint-name
    CHECK (condition(s)-that-can-include-AND-or-OR-or-BETWEEN)
);
```

Example: CREATE TABLE

TASK: Create a new table to track books

```
CREATE TABLE bookshelf.book (
    book_id BIGINT IDENTITY(1,1) PRIMARY KEY,
    title varchar(45) NOT NULL,
    author varchar(45) NOT NULL,
    publish_date DATE, -- Can be null because our DB may contain some unpublished books
    price DECIMAL(3, 2),   -- 3 digits plus 2 after the decimal, $999.99 is max price
    CONSTRAINT unique_author_title UNIQUE (author, title)
);
```

TASK: Create a new table to track books but specify the database first with a USE statement

```
USE bookshelf;

CREATE TABLE book (
    book_id BIGINT IDENTITY(1,1) PRIMARY KEY,
    title varchar(45) NOT NULL,
    author varchar(45) NOT NULL,
    publish_date DATE, -- Can be null because our DB may contain some unpublished books
    price DECIMAL(3, 2),   -- 3 digits plus 2 after the decimal, $999.99 is max price
    CONSTRAINT unique_author_title UNIQUE (author, title)
);
```

TASK: Create a new table to track books and make sure price is not negative

```
USE bookshelf;

CREATE TABLE book (
    book_id BIGINT IDENTITY(1,1) PRIMARY KEY,
    title varchar(45) NOT NULL,
    author varchar(45) NOT NULL,
    publish_date DATE, -- Can be null because our DB may contain some unpublished books
    price DECIMAL(3, 2),   -- 3 digits plus 2 after the decimal, $999.99 is max price
    CONSTRAINT unique_author_title UNIQUE (author, title),
    CONSTRAINT not_negative CHECK (price >= 0);
);
```

Dropping Tables or Databases

- **You can drop (delete) a table or even a database**

- **Make sure to use caution as it permanently deletes the all schema and data associated with the table or database**

Syntax

```
DROP TABLE table-name;
```

Syntax

```
DROP DATABASE database-name;
```

Example: USING DROP

TASK: Drop the book table

```
USE bookshelf;

DROP TABLE book;
```

TASK: Drop the bookshelf database

```
DROP DATABASE bookshelf;
```

Referential Integrity

- **As we have seen, frequently information in one table will reference information in another tables**
  - This is done through a 'Foreign Key' relationship

- **SQL Server uses the FOREIGN KEY constraint**
  - Best practice says the name of the foreign key column should match the name of the primary key column

Syntax

```
CREATE TABLE table1 (
   table1-primary-key-column datatype PRIMARY KEY,
   column2 datatype,
   column3 datatype,
   ....
);

CREATE TABLE table2 (
   table2-primary-key-column datatype PRIMARY KEY,
   table1-primary-key-column datatype, -- foreign key to table1
   column3 datatype,
   FOREIGN KEY (table1-foreign-key-column)
   REFERENCES table1(table1-primary-key-column)
   ....
);
```

- **The FOREIGN KEY REFERENCES clause creates a foreign key constraint which ensures 2 things:**
  - You can't add a record in table 2 with a foreign key value that doesn't exist as a primary key value in table 1
  - You can't a record in table 1 if there are still records in table 2 that reference that it via a foreign key

Example: FOREIGN KEY

TASK: Create a new review table for book reviews where book-to-review is a 1-to-M relationship.

```
USE bookshelf;

CREATE TABLE book (
    book_id BIGINT IDENTITY(1,1) PRIMARY KEY,
    title varchar(45) NOT NULL,
    author varchar(45) NOT NULL,
    publish_date DATE,
    price DECIMAL(3, 2),
    CONSTRAINT unique_author_title UNIQUE (author, title)
);

CREATE TABLE review (
    review_id BIGINT PRIMARY KEY IDENTITY(1,1),
    book_id BIGINT NOT NULL,
    rating INT,
    FOREIGN KEY (book_id) REFERENCES book(book_id),
    CONSTRAINT one_to_ten CHECK (rating BETWEEN 0 AND 10)
);
```

- **Referential integrity ensures:**
  - You can't add a record to review for a book_id that doesn't exist in book
  - You can't delete a book that still has a review for it

Altering a Table

- **You can use the ALTER TABLE statement to make changes to the table structure, including:**
  - Adding/modifying/dropping columns
  - Modifying constraints
- **The alteration only changes the structure or definition of the table, but the data stored in the table remains intact**
  - You may encounter data conversion or validation errors when the modification of a data type or constraint is incompatible with existing data

Syntax

```
ALTER TABLE table_name
ADD COLUMN define-column-here;
```

Syntax

```
ALTER TABLE table_name
MODIFY COLUMN define-column-here;
```

Syntax

```
ALTER TABLE table_name
DROP COLUMN column-name;
```

Syntax

```
ALTER TABLE table_name
ADD CONSTRAINT define-constraint-here;
```

Syntax

```
ALTER TABLE table_name
DROP CONSTRAINT constraint-name;
```

Example: ALTER TABLE

TASK: Add a column to the book table to hold Hardback or Paperback

```
USE bookshelf;

ALTER TABLE book
ADD COLUMN format VARCHAR(9) NOT NULL DEFAULT 'Hardback';
```

TASK: Modify the price column to allow prices up to $9999.99

```
USE bookshelf;

ALTER TABLE book
ALTER COLUMN price DECIMAL(4,2) NOT NULL;
```

TASK: Replace a constraint that requires unique author/title records with one that allows a book to have an additional publication date (ex: 2nd edition)

```
USE bookshelf;

ALTER TABLE book
DROP CONSTRAINT unique_author_title;

-- replace with a constraint that allows a book to have an additional publication date
-- (ex: 2nd edition)
ALTER TABLE book
ADD CONSTRAINT unique_author_title_publish_date UNIQUE (author, title, publish_date);
```

Example: Script to Create and Seed a Database

```
CREATE DATABASE bookshelf;

-- Create the book table and add some data to it...

USE bookshelf;

CREATE TABLE book (
    book_id BIGINT IDENTITY(1,1) PRIMARY KEY,
    title varchar(45) NOT NULL,
    author varchar(45) NOT NULL,
    publish_date DATE,
    price DECIMAL(6, 2),
    CONSTRAINT unique_author_title UNIQUE (author, title)
);

INSERT INTO book VALUES ('Pride and Prejudice', 'Jane Austen',
'1813-01-28', 7.99);

INSERT INTO book (title, author, publish_date, price)
VALUES ('To Kill a Mockingbird', 'Harper Lee', '1960-07-11',
15.99);

INSERT INTO book (title, author, publish_date, price)
VALUES ('1984', 'George Orwell', '1949-06-08', 12.99);

INSERT INTO book (title, author, publish_date, price)
VALUES ('The Great Gatsby', 'F. Scott Fitzgerald', '1925-04-
10', 14.99);

INSERT INTO book (title, author, publish_date, price)
VALUES ('Moby-Dick', 'Herman Melville', '1851-10-18', 13.99);

INSERT INTO book (title, author, publish_date, price)
VALUES ('To the Lighthouse', 'Virginia Woolf', '1927-05-05',
11.99);
```

Other Features

- **There are many SQL and SQL Server features that we haven't discussed, including:**
  - creating and using views
  - creating and using stored procedures
  - working with triggers
- **It never hurts a developer to learn more SQL!**

Module 7

Project:

Creating, Seeding, Querying, and Updating a Database

Section 7–1

Project Description

Project Description

In this project, you will create a database of US States and the popular attractions in them. Name the database vacation_attractions. Keep all of the SQL statements in a single SQL script file.

Your database will have two tables: states and attractions. There is a one-to-many relationship between these tables. That is, one state can have many attractions in it, but each attraction is in exactly one state.

For the states table, you will minimally include columns for: state id (auto-increment/primary key), state name, state abbreviation, and capital city. For attractions, you will minimally include columns for: attraction id (auto-increment/primary key), attraction name, attraction description, price of admission (may be NULL), state abbreviation, and any other fields that you want.

Design a SQL script that:

- begins by dropping the database (in case it already exists)
- creates the database
- creates the states table
- creates the attractions table
- adds at least 5 rows to the states table
- adds at least 12 rows to the attractions table

Once you get your script working and the database built and seeded, create another script that runs the following queries:

- list all states
- list all attractions, but join with states so that you can display the state name (rather than abbreviation) of each attraction
- list all attractions that are free to enter (have 0 or NULL as a price)
- update the price on one of your parks that is free
- list all attractions that are free to enter (the one you changed the admission price should not appear)

- delete one of your attractions
- list all attractions(is the deleted one gone?)
- add a new attraction
- list all attractions(is the new one there?)

Module 8

Advanced Features

Section 8–1

Views

Views

- **Views are virtual tables created by storing a predefined SQL query**
  - They contain columns and rows like a regular table
  - They do NOT store data themselves – they represent data from one or more *other* tables

- **Views serve as reusable query templates, making it easier to re-run complex queries**

Syntax

```
CREATE VIEW view_name AS
SELECT some_column, some_other_column
FROM some_table
WHERE condition
```

- **Views can be queried, updated, inserted into, or deleted from (depending on the table's permissions)**

Example: Views

TASK: Create a view to retrieve most commonly used employee information

```
USE northwind;

CREATE VIEW EmployeeInfo AS
SELECT EmployeeID, FirstName, LastName, Title
FROM Employees;
```

TASK: Create a view that provides information about products and their suppliers

```
USE northwind;

CREATE VIEW ProductSupplierInfo AS
SELECT p.ProductID, p.ProductName, p.UnitPrice, s.CompanyName AS SupplierName
FROM Products p
JOIN Suppliers s ON p.SupplierID = s.SupplierID;
```

TASK: Create a view that shows the total number of orders place by each customer

```
USE northwind;

CREATE VIEW CustomerOrderCount AS
SELECT c.CustomerID, c.CompanyName, COUNT(o.OrderID) AS OrderCount
FROM Customers c
LEFT JOIN Orders o ON c.CustomerID = o.CustomerID
GROUP BY c.CustomerID, c.CompanyName;
```

Section 8–2

Triggers

Triggers

- **Triggers are a special type of stored procedure that are automatically executed in response to specific events that occur in the database**

- **They are associated with a particular table or view and fired off when certain operations such as INSERT, UPDATE, or DELETE are performed**

- **Triggers can be fired BEFORE or AFTER the triggering event**

Syntax

```
CREATE TRIGGER trigger_name
ON table_name
[ BEFORE | AFTER ] { INSERT, UPDATE, DELETE }
AS
BEGIN
-  Trigger logic goes here
END;
```

Example: Trigger

TASK: Create a new table called AuditLog. Create a trigger on the Employees that inserts a new row

- **First create the AuditLog table**

```
USE northwind;

CREATE TABLE AuditLog (
    LogID INT IDENTITY(1,1) PRIMARY KEY,
    EventDateTime DATETIME,
    EventType VARCHAR(50),
    TableName VARCHAR(50)
);
```

- **Then create the trigger on the Employees Table**
  - You can't add a record to review for a book_id that doesn't exist in book
  - You can't delete a book that still has a review for it

```
CREATE TRIGGER audit_trigger
ON Employees
AFTER INSERT, UPDATE, DELETE
AS
BEGIN
  -- Insert audit information into the AuditLog table
  INSERT INTO dbo.AuditLog (EventDateTime, EventType, TableName)
  VALUES (GETDATE(), 'INSERT/UPDATE/DELETE', 'Employee');
END;
```

  - Perform an INSERT/UPDATE/DELETE query on the Employees and check the AuditLog table for a new row to validate

Section 8–3

Materialized Query Tables

Materialized Query Tables

- **Materialized Query Tables (MQTs) are physical tables storing precomputed query results**
  - They serve as a cache for frequently accessed data
  - Stored results improve query performance
- **MQTs can help avoid expensive computations during query execution, reducing query response time**

Syntax

```
CREATE VIEW MaterializedQueryTable
WITH SCHEMABINDING
AS
SELECT
    /* Columns selected from joined tables */
FROM
    /* First table */
JOIN
    /* Second table */
    ON /* Join condition */
WHERE
    /* Optional WHERE clause */
GROUP BY
    /* Optional GROUP BY clause */
```

Example: Materialized Query Table

TASK: Create a materialized query table (indexed view) named SalesTotalByCategory in the Northwind database. Also, create a clustered index on this view to optimize performance.

```
CREATE VIEW SalesTotalByCategory
WITH SCHEMABINDING
AS
SELECT
    dbo.Categories.CategoryID,
    dbo.Categories.CategoryName,
    SUM(dbo.[Order Details].UnitPrice * dbo.[Order Details].Quantity * (1 - dbo.[Order Deta
ils].Discount)) AS TotalSales
FROM
    dbo.Categories
JOIN
    dbo.Products ON dbo.Categories.CategoryID = dbo.Products.CategoryID
JOIN
    dbo.[Order Details] ON dbo.Products.ProductID = dbo.[Order Details].ProductID
JOIN
    dbo.Orders ON dbo.[Order Details].OrderID = dbo.Orders.OrderID
WHERE
    dbo.Orders.ShippedDate IS NOT NULL
GROUP BY
    dbo.Categories.CategoryID, dbo.Categories.CategoryName;
```

TASK: Create a clustered index on the above view to optimize performance.

```
CREATE UNIQUE CLUSTERED INDEX IX_SalesTotalByCategory ON SalesTotalByCategory(CategoryID);
```