2020

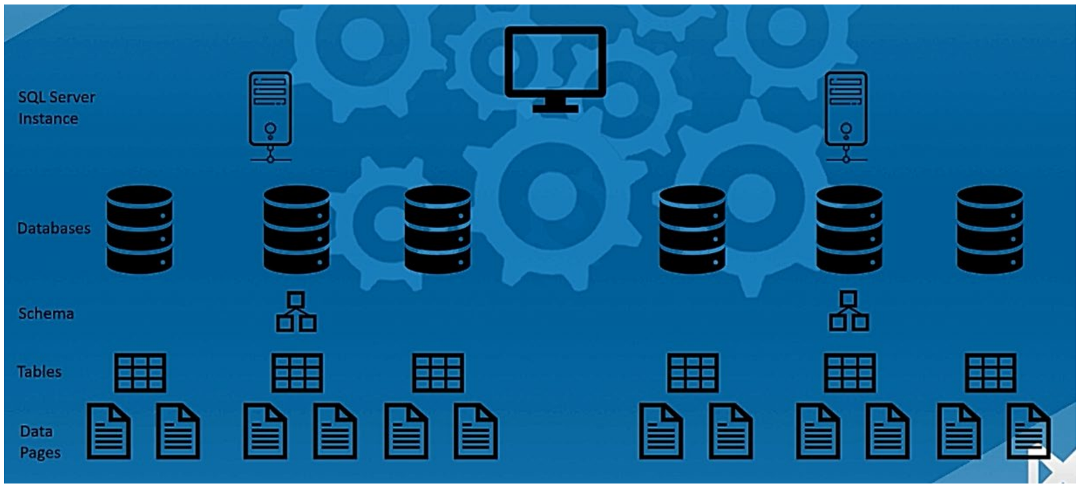
Microsoft SQL Server

PAYMAN FARAHANI

Section 4 - How is Data Stored in SQL Server

1. Simple Hierarchy & Structure

- At the top level you have a SQL Server "Instance"
- Within each instance you can have multiple "Databases"
- Each database can have multiple "Schemas"
    - Schema is a collection/container of database objects. It is associated with a username which is called the "Schema Owner". This schema owner is the owner of all the database objects within the schema
    - The default schema for a newly created database object is **dbo**, which is owned by the dbo user account.
- Each schema can contain:
    - Views
    - Sorted Procedures
    - Tables
    - And more…
- All of table data is organized in within 8K data pages



2. Why Relational

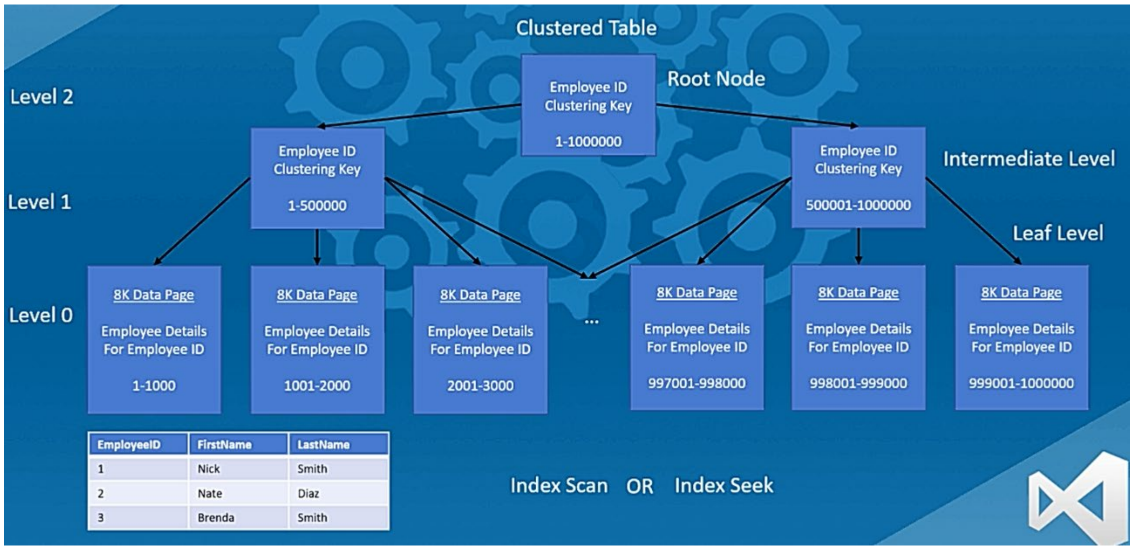
- Some of the available constraints:
    - **NOT NULL** – all columns by default can hold NULL values, but this constraint restricts any NULL values from being inserted into columns.
    - **UNIQUE** – ensures uniqueness for the columns just like "Primary Key" but you can have many UNIQUE constraints per table and only one "Primary Key".
    - **DEFAULT** – provides a default value for a column, if new record is entered and has no value for the column with DEFAULT constraint then the default value will be used.
    - **CHECK** – limits the value range that can be placed in a column.
- There are 3 types of relationships in SQL Server or in general relational database design:
    - **One-to-One**

- **One-to-Many** or **Many-to-One**
- **Many-to-Many**

3. Indexes

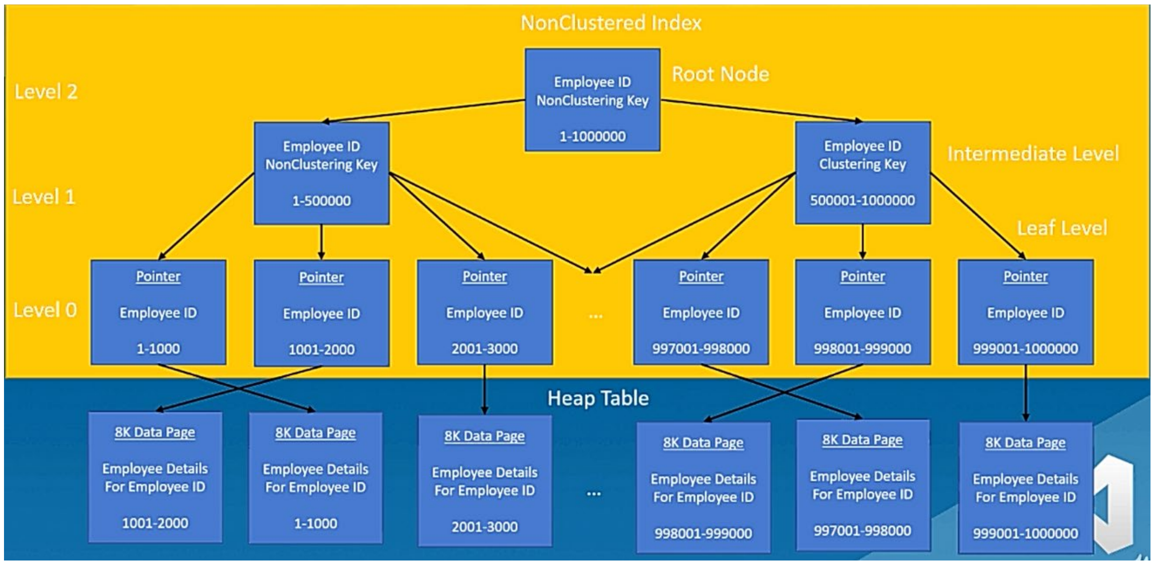
- When a table doesn't have an index – those 8K data pages are sorted in an unordered structure called a heap.
- We have 2 types of indexes:

  - **Clustered**
    - Clustered index orders the physical structure of the table itself, which is why we can have only one clustered index per table
    - Once the table has a clustered index it is called a "Clustered Table", since it is no longer a heap – aka stored in an unordered manner.


Clustered Table
Root Node
Level 2
Employee ID Clustering Key 1-1000000
Level 1
Employee ID Clustering Key 1-500000
Employee ID Clustering Key 500001-1000000
Intermediate Level
Leaf Level
Level 0
8K Data Page Employee Details For Employee ID 1-1000
8K Data Page Employee Details For Employee ID 1001-2000
8K Data Page Employee Details For Employee ID 2001-3000
...
8K Data Page Employee Details For Employee ID 997001-998000
8K Data Page Employee Details For Employee ID 998001-999000
8K Data Page Employee Details For Employee ID 999001-1000000
Index Scan OR Index Seek

EmployeeID	FirstName	LastName
1	Nick	Smith
2	Nate	Diaz
3	Brenda	Smith

  - **NonClustered**
    - NonClustered index creates the order in a separate structure from the data pages, so it doesn't change the structure of the table itself; instead, it points to it.
    - Unlike clustered index you can have many NonClustered indexes on the same table since the order is sorted in separate structures for each NonClustered index.
    - Clustered index gets created by default when adding a primary key constraint, but it is possible to manually set it so that a NonClustered index gets created instead.


NonClustered Index
Root Node
Level 2
Employee ID NonClustering Key 1-1000000
Level 1
Employee ID NonClustering Key 1-500000
Employee ID Clustering Key 500001-1000000
Intermediate Level
Leaf Level
Level 0
Pointer Employee ID 1-1000
Pointer Employee ID 1001-2000
Pointer Employee ID 2001-3000
...
Pointer Employee ID 997001-998000
Pointer Employee ID 998001-999000
Pointer Employee ID 999001-1000000
Heap Table
8K Data Page Employee Details For Employee ID 1001-2000
8K Data Page Employee Details For Employee ID 1-1000
8K Data Page Employee Details For Employee ID 2001-3000
...
8K Data Page Employee Details For Employee ID 998001-999000
8K Data Page Employee Details For Employee ID 997001-998000
8K Data Page Employee Details For Employee ID 999001-1000000

Section 5 - DDL in SQL Server

1. Data Definition Language Commands

- The main commands are:
  - **CREATE** – Allows you to create a new database or objects in the database
  - **DROP** – Allows you to delete a whole database or just a database object
  - **TRUNCATE** – Deletes all records from a table irreversibly and resets table identity to initial value
  - **ALTER** – Allows you to modify database options or database object properties

2. How to Create

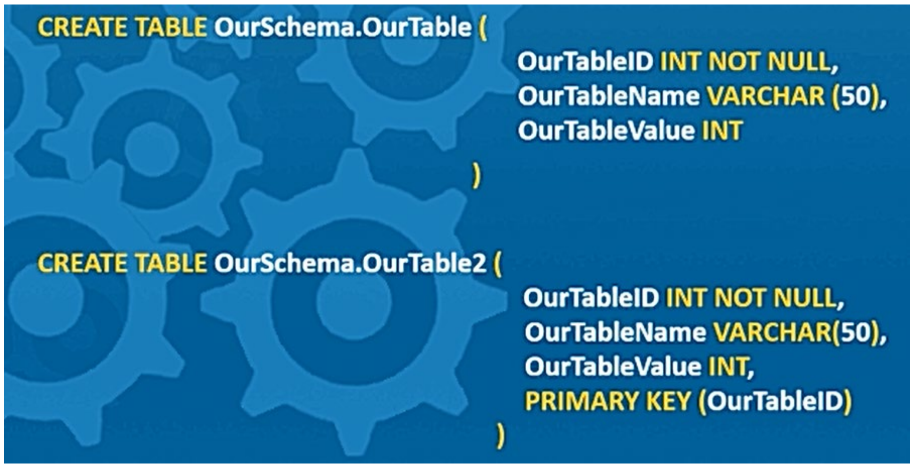
- Create a database:
  ```
  CREATE DATABASE OurDatabase
  ```

- Create a schema:
  ```
  CREATE SCHEMA OurSchema
  ```
  *Note: by default, we're the schema owner but if we want to give it a custom schema owner then we have to write this code:
  ```
  CREATE SCHEMA OurSchema2 AUTHORIZATION guest
  ```

- Most common data types in SQL Server:
  - **INT**
  - **DATETIME**
    - Use to store the date and time
  - **DECIMAL, FLOAT**

        - DECIMAL allows you to specify the number of decimal points to maintain and FLOAT does not
    - `BIT`
        - Stores a Boolean value and is often used as a flag or an indicator of some sort
    - `VARCHAR`(SIZE), `CHAR`(SIZE)
        - Both types are used to store non-Unicode (no national characters, other languages)
        - Both allocate 1 byte for each character so CHAR(20) or VARCHAR(20) means 20 bytes total allocated and 1 byte per character
        - CHAR is fixed size so if you only insert 4 characters and the total is 20 bytes you will end up with 16 bytes of wasted memory
        - VARCHAR is variable-length so if the total is 20 bytes but only 4 characters are inserted then it will only se 4 bytes and not waste the rest
    - `NVARCHAR`(SIZE), `NCHAR`(SIZE)
        - Both types are used to store Unicode and non-Unicode character values
        - Both allocated 2 bytes for each character
        - Rest of the properties are the same as above except the fact that 2 bytes are being allocated per character instead of 1 byte

- How to create a table:

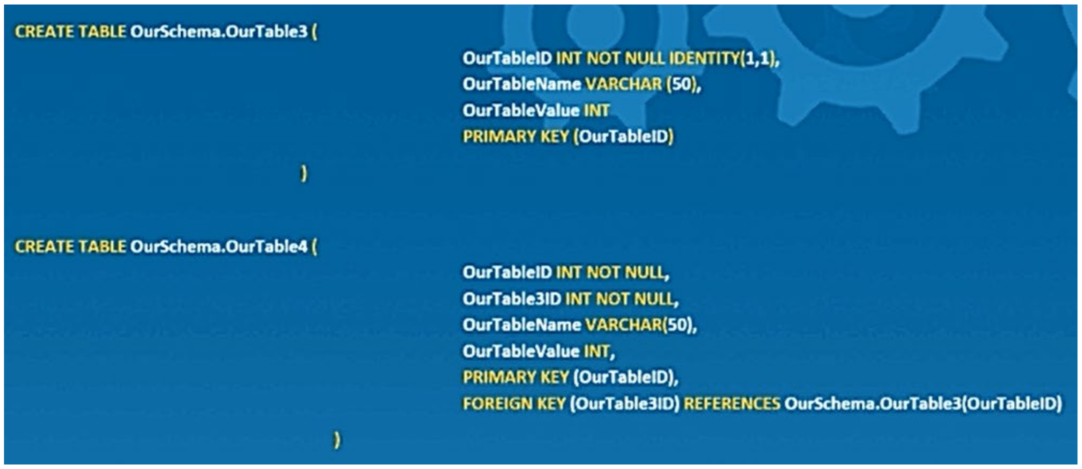


```
CREATE TABLE OurSchema.OurTable (
                                  OurTableID INT NOT NULL,
                                  OurTableName VARCHAR (50),
                                  OurTableValue INT
                                )

CREATE TABLE OurSchema.OurTable2 (
                                  OurTableID INT NOT NULL,
                                  OurTableName VARCHAR(50),
                                  OurTableValue INT,
                                  PRIMARY KEY (OurTableID)
                                 )
```

        - "OurTableID", "OurTableName", and "OurTableValue" are the names of our columns
        - "`NOT NULL`" is a constraint
        - "`PRIMARY KEY()`" is a constraint which guarantees uniqueness

        *Note: the problem with table 2 is that each time we insert a new record we have to figure out what is the unique ID that we have to generate in order to not violate the constraint.



```
CREATE TABLE OurSchema.OurTable3 (
        OurTableID INT NOT NULL IDENTITY(1,1),
        OurTableName VARCHAR (50),
        OurTableValue INT
        PRIMARY KEY (OurTableID)
)

CREATE TABLE OurSchema.OurTable4 (
        OurTableID INT NOT NULL,
        OurTable3ID INT NOT NULL,
        OurTableName VARCHAR(50),
        OurTableValue INT,
        PRIMARY KEY (OurTableID),
        FOREIGN KEY (OurTable3ID) REFERENCES OurSchema.OurTable3(OurTableID)
)
```

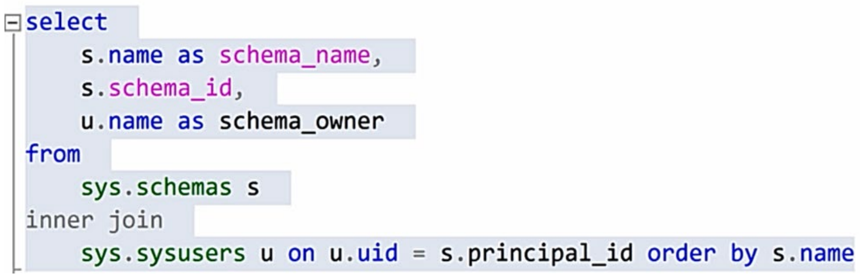
- To fix the problem with table 2, we use "`IDENTITY()`". Every time we add a new row, it generates a unique ID. In this example it says start the ID from 1 and increments by 1.
- In table 4 we're defining a foreign key. With "`FOREIGN KEY()`" we specify the table we're going to refer and with "`REFERENCES`" we specify which column it is connected to.

- Create a clustered index:
  ```
  CREATE CLUSTERED INDEX OurClusterIndex ON OurSchema.OurTable
  (OurTableID)
  ```

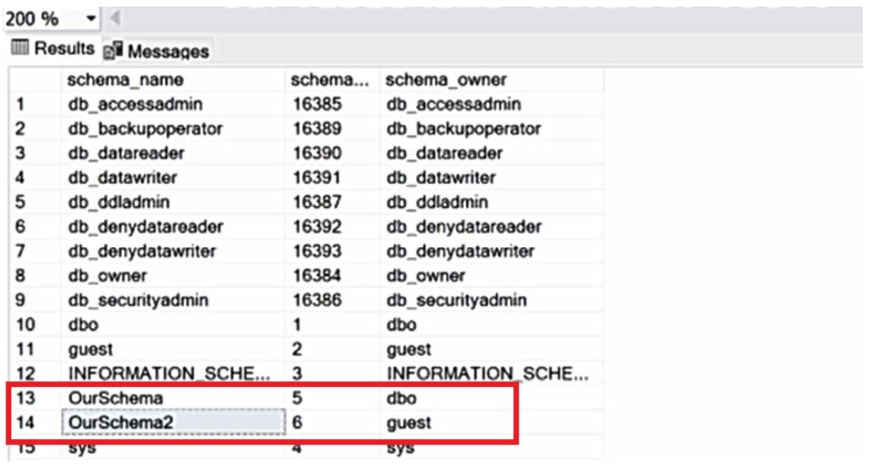
- Create a clustered index:
  ```
  CREATE NONCLUSTERED INDEX OurNonClusterIndex ON
  OurSchema.OurTable (OurTableID)
  ```

- There are 2 ways to create a comment:
  1. Type - - before the line
  2. Use shortcuts:
     - **Ctrl + K + C** to comment the line
     - **Ctrl + K + U** to uncomment the line
  3. For longer comments use them between /* */

- To know if schemas are assigned to the correct users:



```
select
    s.name as schema_name,
    s.schema_id,
    u.name as schema_owner
from
    sys.schemas s
inner join
    sys.sysusers u on u.uid = s.principal_id order by s.name
```


200 %
Results Messages

	schema_name	schema...	schema_owner
1	db_accessadmin	16385	db_accessadmin
2	db_backupoperator	16389	db_backupoperator
3	db_datareader	16390	db_datareader
4	db_datawriter	16391	db_datawriter
5	db_ddladmin	16387	db_ddladmin
6	db_denydatareader	16392	db_denydatareader
7	db_denydatawriter	16393	db_denydatawriter
8	db_owner	16384	db_owner
9	db_securityadmin	16386	db_securityadmin
10	dbo	1	dbo
11	guest	2	guest
12	INFORMATION_SCHE...	3	INFORMATION_SCHE...
13	OurSchema	5	dbo
14	OurSchema2	6	guest
15	sys	4	sys

- In the results we can see OurSchema is assigned to the default user (dbo) And OurSchema2 is assigned to the user named "guest"

3. How to Modify

- Add a column with ALTER TABLE:

```
    ALTER TABLE OurSchema.OurTable
        ADD NewColumn INT
```

- If you would like to add multiple columns:

```
    ALTER TABLE OurSchema.OurTable
        ADD NewColumn INT,
        AnotherColumn INT
```

- Modify a column with ALTER TABLE (changing the type of our column):

```
ALTER TABLE OurSchema.OurTable
        ALTER COLUMN NewColumn VARCHAR (50)
```

- Drop a column with ALTER TABLE (remove our column):

```
ALTER TABLE OurSchema.OurTable
        DROP COLUMN NewColumn
```

4. How to Truncate

- Truncate a table:

```
TRUNCATE TABLE OurSchema.OurTable
```

5. How to Drop

- Drop indexes (clustered and non-clustered)

```
    DROP INDEX OurClusterIndex ON OurSchema.OurTable
    DROP INDEX OurNonClusterIndex ON OurSchema.OurTable
```

- Drop a table:

```
DROP TABLE OurSchema.OurTable
```

- Drop a schema:

```
DROP SCHEMA OurSchema
```

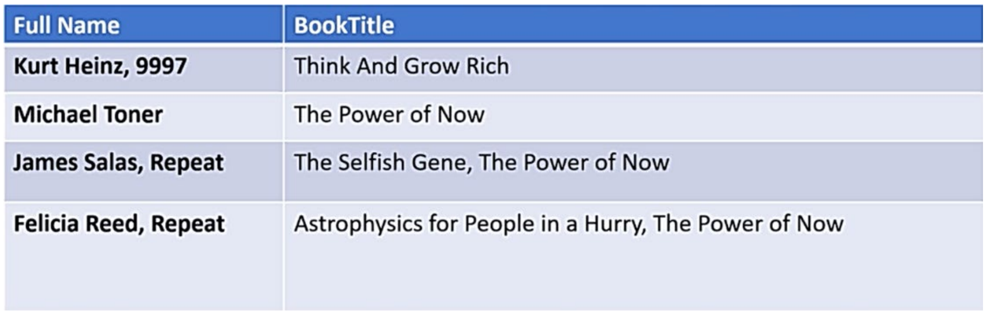
*Note: if the schema contains objects, the operation fails. Once we drop all the objects at first, the schema will be dropped successfully.

- Drop a database:

```
DROP DATABASE OurDatabase
```

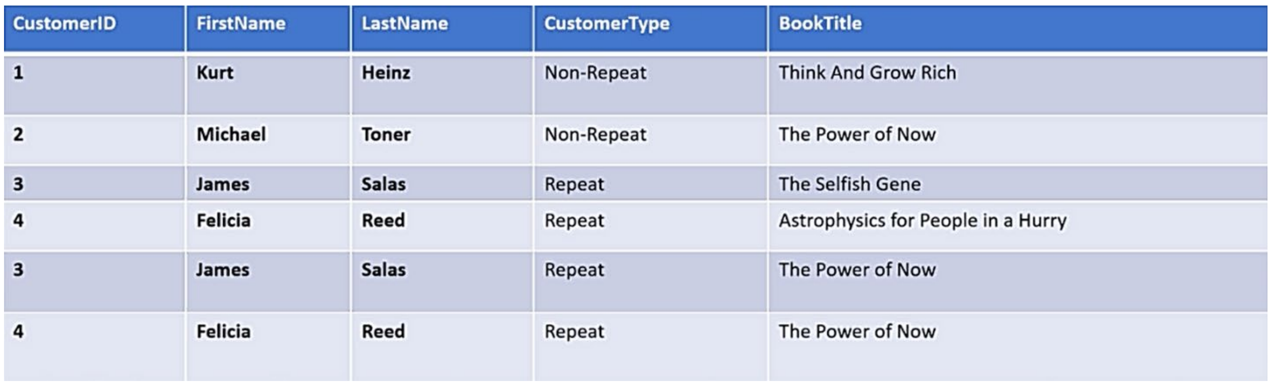
6. Normalization

- Normalization is a database technique where we build and organize the table in such way that it reduces redundancy and dependency of data
- It divides larger tables to smaller ones and uses the relational features of RDBMS to link those tables together
- There are 4 “phases” to the database normalization process. In most practical applications normalization achieves its best result in the 3rd phase.
    - First Normal Form (1NF)
    - Second Normal form (2NF)
    - Third Normal Form (3NF)
    - Boyce-code Normal Form (BCNF)
- Table is in First Normal Form if:
    1. Each cell of a table has only one value
    2. All of the data in a column must mean the same thing
    3. Each row of a table must be unique
    4. A table should not have any repeating groups



Full Name	BookTitle
Kurt Heinz, 9997	Think And Grow Rich
Michael Toner	The Power of Now
James Salas, Repeat	The Selfish Gene, The Power of Now
Felicia Reed, Repeat	Astrophysics for People in a Hurry, The Power of Now

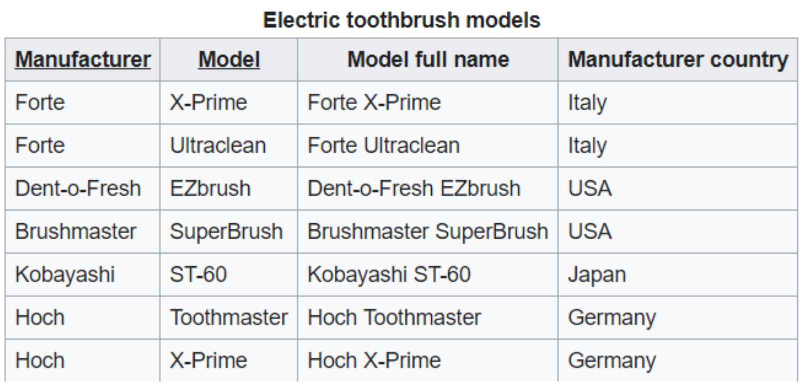
- The above table is not normalized and everything is saved in the same table
- The table bellow is normalized in 1NF.



CustomerID	FirstName	LastName	CustomerType	BookTitle
1	**Kurt**	**Heinz**	Non-Repeat	Think And Grow Rich
2	**Michael**	**Toner**	Non-Repeat	The Power of Now
3	**James**	**Salas**	Repeat	The Selfish Gene
4	**Felicia**	**Reed**	Repeat	Astrophysics for People in a Hurry
3	**James**	**Salas**	Repeat	The Power of Now
4	**Felicia**	**Reed**	Repeat	The Power of Now

*Note: The problem with the first normal form is that there is a lot of redundancy and if you change one record it has other ones which are related to it you would need to make the same type of change. Otherwise they will be wrong and over time this can get very cumbersome. So, if we change a customer type for James for example in the previous table, we would have to do it twice. If we had him there millions of times, we would have to perform to change millions of times. This is known as "**Functional Dependency**" in the current setup.

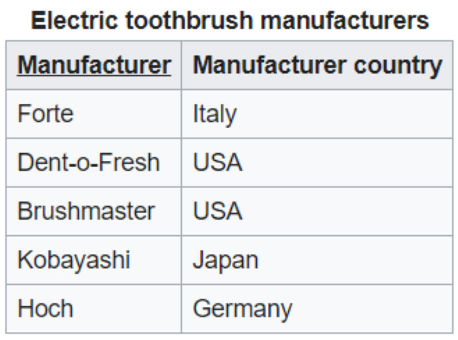
- Table is in Second Normal Form if:
    1. It has to be already in First Normal Form
    2. Each non key field must be functionally dependent on the primary key (Every non key fields are directly related to the primary key)



Electric toothbrush models

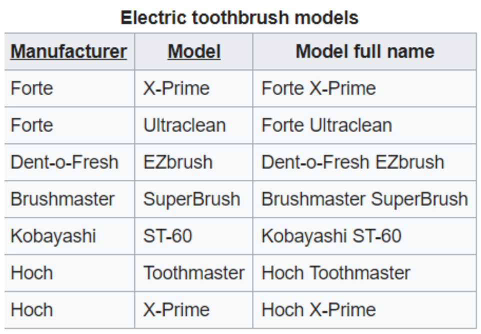
Manufacturer	Model	Model full name	Manufacturer country
Forte	X-Prime	Forte X-Prime	Italy
Forte	Ultraclean	Forte Ultraclean	Italy
Dent-o-Fresh	EZbrush	Dent-o-Fresh EZbrush	USA
Brushmaster	SuperBrush	Brushmaster SuperBrush	USA
Kobayashi	ST-60	Kobayashi ST-60	Japan
Hoch	Toothmaster	Hoch Toothmaster	Germany
Hoch	X-Prime	Hoch X-Prime	Germany

- The above relation does not satisfy 2NF because:
  {Manufacturer country} is functionally dependent on {Manufacturer}
  {Manufacturer country} is not part of a candidate key, so it is a non-prime attribute
  {Manufacturer} is a subset of {Manufacturer, Model} candidate key
  Therefore {Manufacturer country} is a non-prime attribute functionally dependent on a part of a candidate key, and is in violation of 2NF
- Even if the designer has specified the primary key as {Model full name}, the relation is not in 2NF because of the other candidate keys. {Manufacturer, Model} is also a candidate key, and Manufacturer country is dependent on a proper subset of it: Manufacturer.



Electric toothbrush manufacturers

Manufacturer	Manufacturer country
Forte	Italy
Dent-o-Fresh	USA
Brushmaster	USA
Kobayashi	Japan
Hoch	Germany



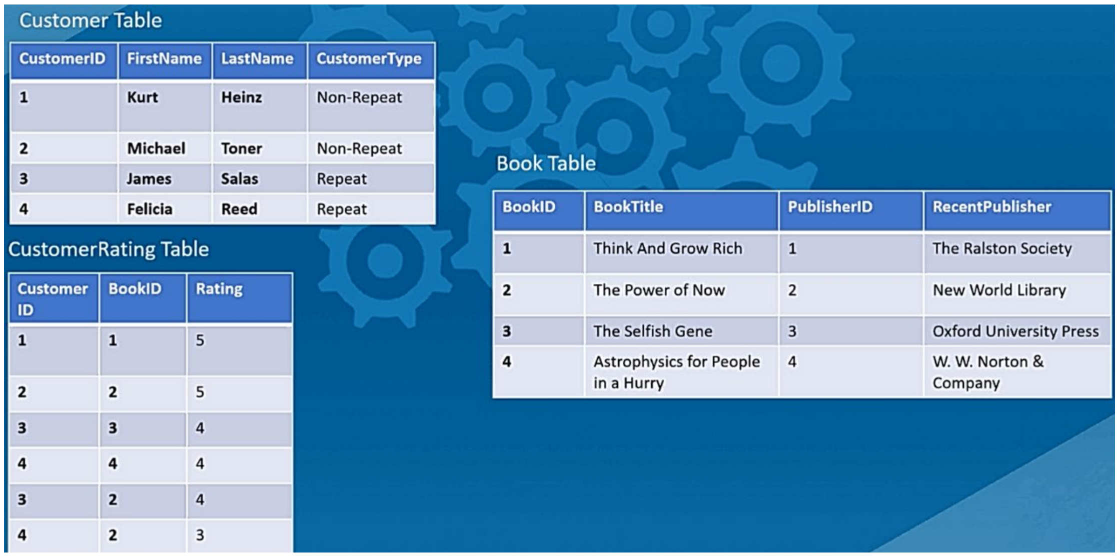
Electric toothbrush models

Manufacturer	Model	Model full name
Forte	X-Prime	Forte X-Prime
Forte	Ultraclean	Forte Ultraclean
Dent-o-Fresh	EZbrush	Dent-o-Fresh EZbrush
Brushmaster	SuperBrush	Brushmaster SuperBrush
Kobayashi	ST-60	Kobayashi ST-60
Hoch	Toothmaster	Hoch Toothmaster
Hoch	X-Prime	Hoch X-Prime

- To make the design conform to 2NF, it is necessary to have two relations

- Table is in Third Normal Form if:

1. Table is already in Second Normal Form
2. There is no other non-key attribute that you would need to change in a table if you changed a non-key attribute (**Transitive Dependency**). In other words, 3NF doesn't have Transitive Dependency.



Customer Table

CustomerID	FirstName	LastName	CustomerType
1	Kurt	Heinz	Non-Repeat
2	Michael	Toner	Non-Repeat
3	James	Salas	Repeat
4	Felicia	Reed	Repeat

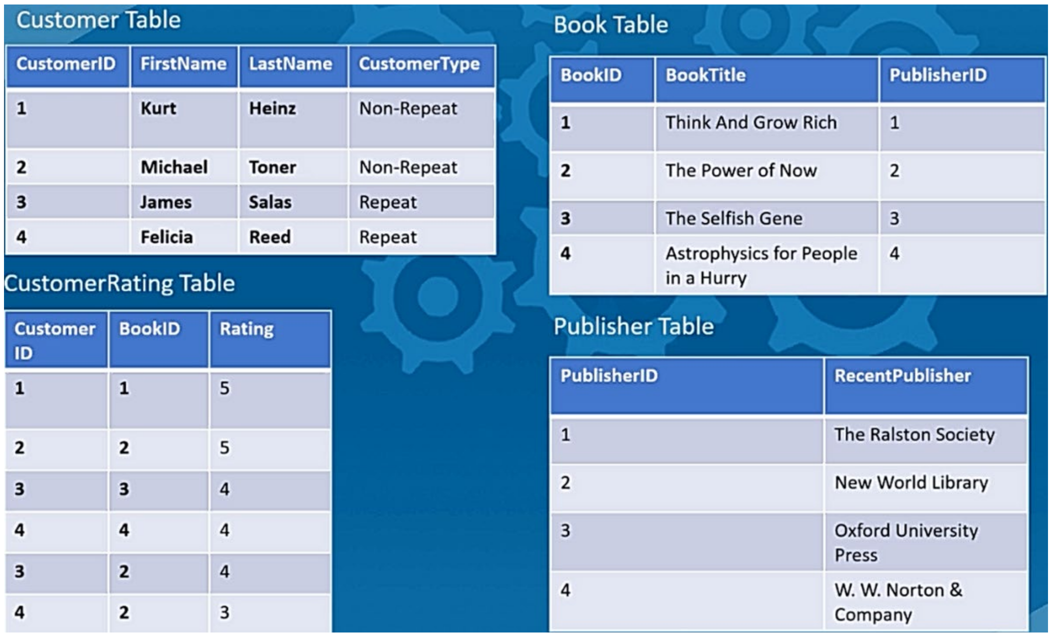
CustomerRating Table

Customer ID	BookID	Rating
1	1	5
2	2	5
3	3	4
4	4	4
3	2	4
4	2	3

Book Table

BookID	BookTitle	PublisherID	RecentPublisher
1	Think And Grow Rich	1	The Ralston Society
2	The Power of Now	2	New World Library
3	The Selfish Gene	3	Oxford University Press
4	Astrophysics for People in a Hurry	4	W. W. Norton & Company

- In the above table {PublisherID} and {RecentPublisher} are both dependent to {BookID} i.e. they are "Functionally dependent" (which is good). But they also are dependent to each other so, if we want to change {PublisherID} then we have to change {RecentPublisher} because they are "Transitive dependent".
- In order to fix that, we change the tables into the tables bellow. We separate publishers' info into Publisher Table.



Customer Table

CustomerID	FirstName	LastName	CustomerType
1	Kurt	Heinz	Non-Repeat
2	Michael	Toner	Non-Repeat
3	James	Salas	Repeat
4	Felicia	Reed	Repeat

CustomerRating Table

Customer ID	BookID	Rating
1	1	5
2	2	5
3	3	4
4	4	4
3	2	4
4	2	3

Book Table

BookID	BookTitle	PublisherID
1	Think And Grow Rich	1
2	The Power of Now	2
3	The Selfish Gene	3
4	Astrophysics for People in a Hurry	4

Publisher Table

PublisherID	RecentPublisher
1	The Ralston Society
2	New World Library
3	Oxford University Press
4	W. W. Norton & Company

Section 6 - DML in SQL Server

1. Data Manipulation Language Commands

- The main commands are:
  - **SELECT –** Allows you to retrieve data from a database
  - **INSERT –** Allows you to insert data into a table
  - **UPDATE –** Allows you to update data within an existing table
  - **DELETE –** Allows you to delete records from a table, can be used with conditional statements unlike `TRUNCATE` (with `DELETE` you only remove the subset of a table not the entire table)
    - Main differences between `TRUNCATE` and `DELETE` are that `TRUNCATE` is all or nothing, it is much faster, it resets the `IDENTITY` column and `DELETE` does not

2. Begin with Retrieval

- Simple SELECT statement that will retrieve everything from the Person table:
  ```
  SELECT * FROM Person.Person
  ```
  *Note: "*****" means select every objects.
  *Note: the first Person in "`Person.Person`" is the name of the schema and the second one is the name of the table.

- To select a specific column(s):
  ```
  SELECT BusinessEntityID, FirstName, LastName, Suffix FROM
  Person.Person
  ```
  *Note: If there was a space between column names, we have to write it between square brackets. For example: `[First Name], [Last Name], Suffix`

- When we're retrieving a column list we are not just limited to the specific columns on the table. We can perform modifications and change how we would like the column list to be displayed (This is called a **Calculated Field**)
  ```
  SELECT SalesOrderDetailID, OrderQty*UnitPrice FROM Person.Person
  ```

- …and if we want to give those calculated fields an alias:
  ```
  SELECT
          SalesOrderDetailID,
          OrderQty*UnitPrice AS [TotalPrice]
  FROM
          Person.Person
  ```

- We can also have table aliases
  ```
  SELECT SalesOrderDetailID, OrderQty*UnitPrice FROM Person.Person AS
  SOD
  ```

  *Note: To put a space between two concatenated fields, we use: `+' '+`

3. Scalar Functions

- Scalar function just means that a function is going to be performed on a single column
- Main categories of scalar functions:
  - String Functions
  - Date & Time Functions
  - Arithmetic Functions
  - Conversion Functions

- Some of the main String Functions are:
  - **CONCAT** – Similar to using "+"
    ```
    SELECT BusinessEntityID, CONCAT(FirstName, ' ', LastName) FROM
    Person.Person
    ```
  - **SUBSTRING** – Allows us to get part of a string; first parameter is name of the column to use, second parameter is the beginning of where to start, and the third parameter is the length of how much to cut.
    ```
    SELECT BusinessEntityID, SUBSTRING(FirstName,1,4) FROM
    Person.Person
    ```
  - **UPPER**
  - **LOWER**
    ```
    SELECT BusinessEntityID, UPPER(FirstName), LOWER(LastName)
    FROM Person.Person
    ```
  - **LTRIM** – Removes a trailing space from the left side; it comes with `SELECT` and without `FROM`
    ```
    SELECT LTRIM(' Peyman ')
    ```
  - **RTRIM** – Removes a trailing space from the right side
    ```
    SELECT RTRIM(' Peyman ')
    ```
  - **TRIM** – Removes all trailing spaces from both sides
    ```
    SELECT TRIM(' Peyman ')
    ```

- Date and time functions allows us to manipulate date and time values. Some of the main date and time functions are:
  - **GETDATE** – Returns the current date
    ```
    SELECT GETDATE()
    ```
  - **DATENAME** – Gets you back to a specified part of a date; it has the parts "`YEAR`", "`MONTH`", "`DAY`"
    ```
    SELECT DATENAME(YEAR,'2019-10-06')
    ```

- **DATEDIFF** – Returns the difference between two dates on the specified part of the date
  `SELECT DATEDIFF(DAY,'2019-9-7','2019-9-10')`

- **DATEADD** – Allows you to add to a specified part of the date and returns you the calculated date result
  `SELECT DATEADD(DAY,1,'2019-9-10')`

- Arithmetic functions allow us to manipulate numeric values. Some of the main functions are:
  - **ABS**
    `SELECT ABS(-5)`

  - **RAND**
    `SELECT RAND()`

  - **CEILING** – Rounds up the number
    `SELECT CEILING(5.55)`

  - **FLOOR**
    `SELECT FLOOR(5.55)`

- Conversion functions allows us to convert from one data type to another and have ways to deal with NULL values. Some of the main conversion functions are:
  - **CAST** – Allows you to convert from one data type to another. The first parameter is the one to be converted and then you put `AS` and then the data type you want it to be converted to.
    `SELECT CAST('10-01-2019' AS DATETIME)`

  - **ISNULL** – Allows you to specify a fallback result in case the first parameter happens to be NULL
    ```
    SELECT ISNULL('MyName', 'MyBackupName')
    SELECT ISNULL(NULL, 'MyBackupName')
    ```

  - **COALESCE** – It does the same thing with some differences; it allows more than one input and the return type will depend on the type which it falls back to. Another difference between `ISNULL` and `COALESCE` is that if all of the fallback values of `COALESCE` are NULL it will throw an error however if all of the fallback values of `ISNULL` is NULL it will just return `NULL`
    ```
    SELECT COALESCE(NULL, 'MyBackupName')
    SELECT COALESCE(NULL, NULL, 'MyBackupName',
    'MyLastBackupName')
    ```

4. Sorting

- We have a way of returning our data back sorted:
  `SELECT [your column(s)] FROM [your table] ORDER BY [your column(s)]`

- The default is ascending order but we can change it:
```
SELECT BusinessEntityID, FirstName, LastName FROM Person.Person
ORDER BY FirstName DESC

SELECT BusinessEntityID, FirstName, LastName FROM Person.Person
ORDER BY FirstName ASC, LastName DESC
```

5. Column & Row Logic

- To perform conditional logic on a column(s) we need to use the **CASE** statement with associated reserved keywords
- Column logic allows us to manipulate our columns based on a given condition before we get back the result set
```
SELECT
      CASE FirstName
                    WHEN ‘ken’ THEN ‘Ken Found’
                    WHEN ‘Miller’ THEN ‘Miller Found’
                    ELSE FirstName
      END AS [FirstNameWithLogic]
FROM
Person.Person
```
  - `END` states that the condition column is finished
  - In this code we used `AS` to give an alias to our conditional column

- Column logic in search format
```
SELECT
      CASE
           WHEN FirstName=‘ken’ THEN ‘Ken Found’
           WHEN LastName=‘Miller’ THEN ‘Miller Found’
           ELSE FirstName
      END AS [FirstNameOrLastNameWithLogic]
FROM
Person.Person
```

- Row logic allows us to retrieve only certain rows based on a given condition:
```
SELECT [your column(s)] FROM [your table] WHERE [row logic] ORDER BY
[your column(s)]
```

- Example:
```
SELECT BusinessEntityID, FirstName, LastName FROM Person.Person
WHERE Suffix=’Jr.’
```

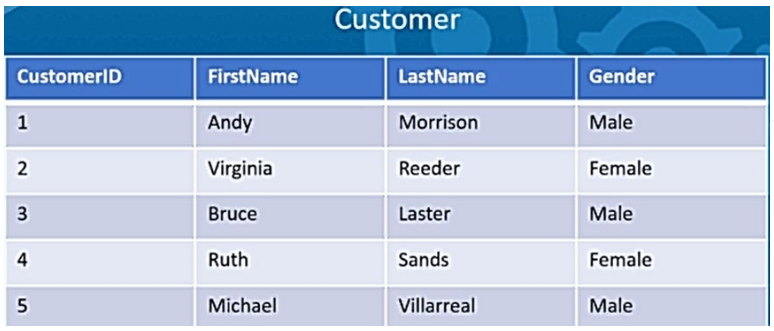
- To select only the amount of records we want:
```
SELECT TOP 10 BusinessEntityID, FirstName, LastName FROM
```

```
Person.Person WHERE Suffix='Jr.'
```

- Other reserved keys that comes with `WHERE`:
  - AND
    ```
    SELECT TOP 10 BusinessEntityID, FirstName, LastName FROM
    Person.Person WHERE Suffix='Jr.' AND FirstName='Dan'
    ```
  - OR
    ```
    SELECT TOP 10 BusinessEntityID, FirstName, LastName FROM
    Person.Person WHERE Suffix='Jr.' OR FirstName='Dan'
    ```
  - IN
    ```
    SELECT TOP 10 BusinessEntityID, FirstName, LastName FROM
    Person.Person WHERE FirstName IN('Eric', 'Adam' )
    ```
  - IS
  - NOT
    ```
    SELECT TOP 10 BusinessEntityID, FirstName, LastName FROM
    Person.Person WHERE Suffix IS NOT NULL
    ```
  - EXISTS

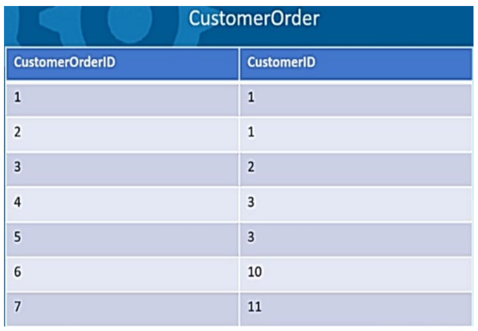
6. Joins

- There are 4 types of joining along with a feature of Self-Joining:
  - INNER JOIN (or JOIN)
  - LEFT OUTER JOIN (or LEFT JOIN)
  - RIGHT OUTER JOIN (or RIGHT JOIN)
  - FULL OUTER JOIN (or FULL JOIN)
  - Self Joining



Customer

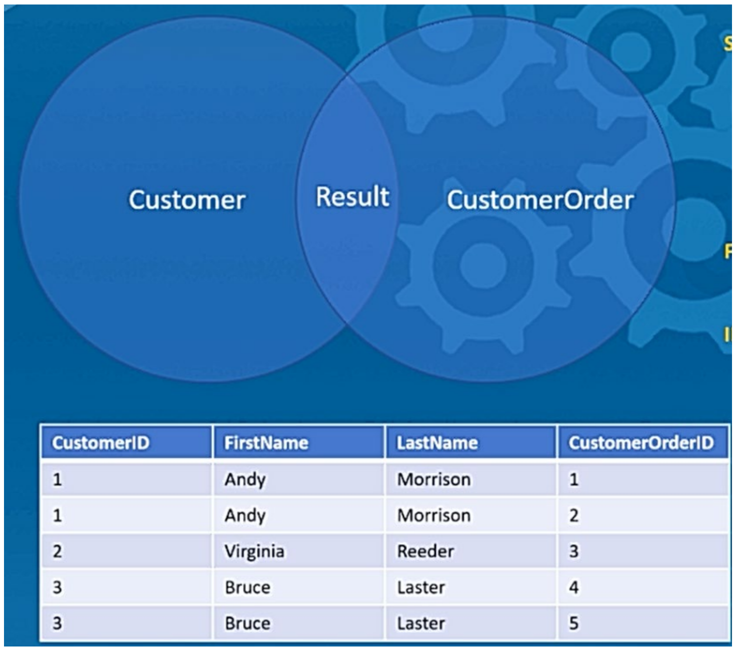
CustomerID	FirstName	LastName	Gender
1	Andy	Morrison	Male
2	Virginia	Reeder	Female
3	Bruce	Laster	Male
4	Ruth	Sands	Female
5	Michael	Villarreal	Male



CustomerOrder

CustomerOrderID	CustomerID
1	1
2	1
3	2
4	3
5	3
6	10
7	11

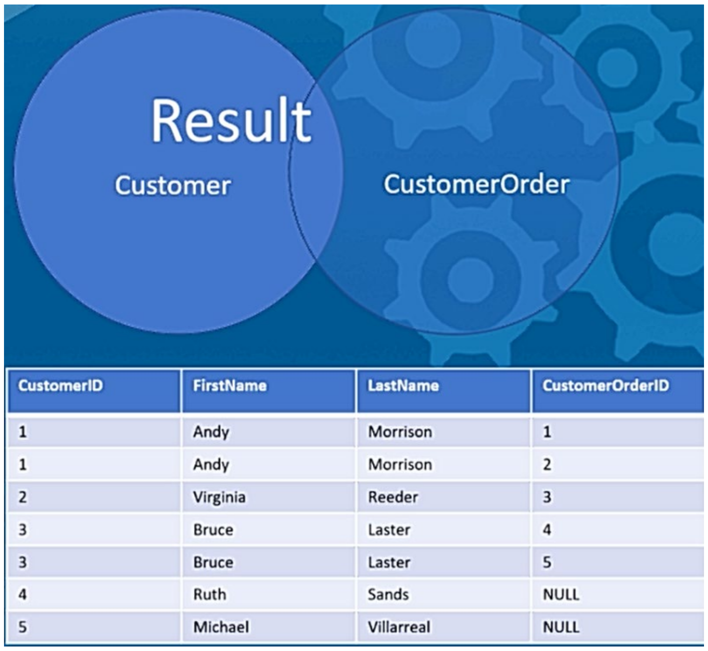
- The result of Inner Join is only the matching rows from {Customer} and {CustomerOrder} – anything that doesn't match will be ignored:
  ```
  SELECT
          c.CustomerID,
          c.FirstName,
          c.LastName,
          co.CustomerOrderID
  ```



CustomerID	FirstName	LastName	CustomerOrderID
1	Andy	Morrison	1
1	Andy	Morrison	2
2	Virginia	Reeder	3
3	Bruce	Laster	4
3	Bruce	Laster	5

- The result of a Left Join is the matching rows from {Customer} and {CustomerOrder} including all the nonmatching rows from {Customer} as well

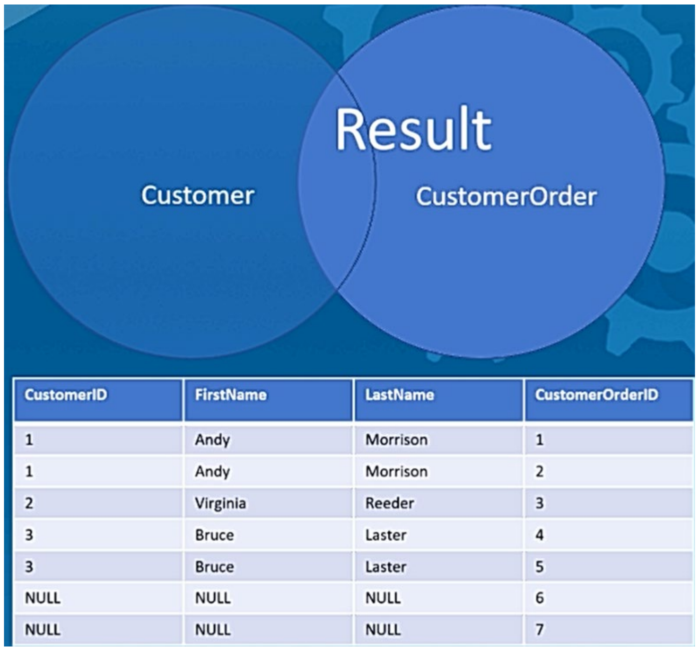
```
SELECT
         c.CustomerID,
         c.FirstName,
         c.LastName,
         co.CustomerOrderID
FROM
         dbo.Customer AS c
LEFT JOIN
         dbo.CustomeOrder co ON c.CustomerID = co.CustomerID
```



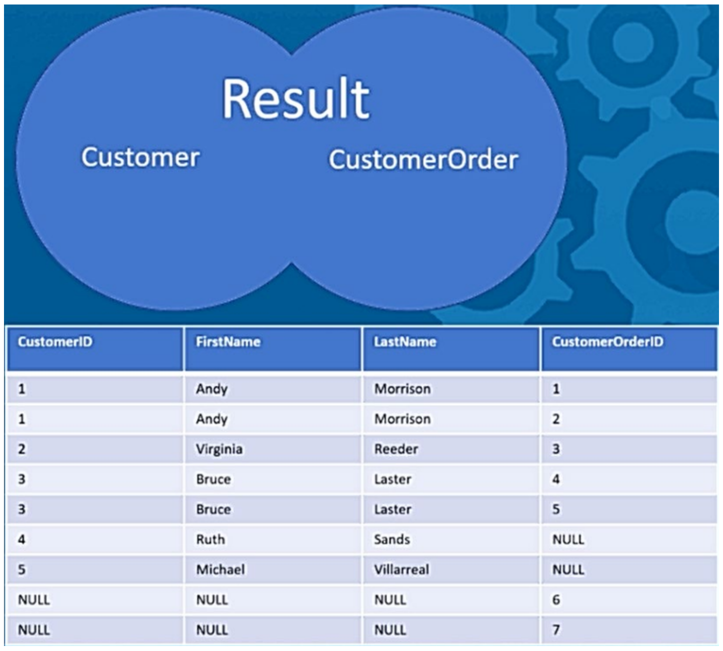
CustomerID	FirstName	LastName	CustomerOrderID
1	Andy	Morrison	1
1	Andy	Morrison	2
2	Virginia	Reeder	3
3	Bruce	Laster	4
3	Bruce	Laster	5
4	Ruth	Sands	NULL
5	Michael	Villarreal	NULL

- The result of a Right Join is the matching rows from {Customer} and {CustomerOrder} including all the nonmatching rows from {CustomerOrder} as well

```
SELECT
         c.CustomerID,
         c.FirstName,
         c.LastName,
         co.CustomerOrderID
FROM
         dbo.Customer AS c
RIGHT JOIN
         dbo.CustomeOrder co ON c.CustomerID = co.CustomerID
```



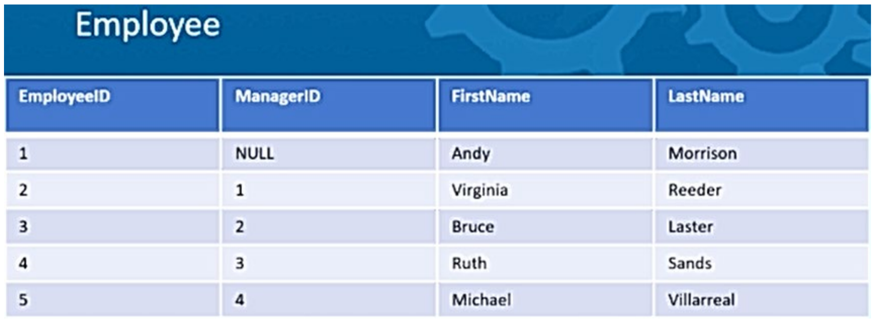
CustomerID	FirstName	LastName	CustomerOrderID
1	Andy	Morrison	1
1	Andy	Morrison	2
2	Virginia	Reeder	3
3	Bruce	Laster	4
3	Bruce	Laster	5
NULL	NULL	NULL	6
NULL	NULL	NULL	7


Result
Customer
CustomerOrder

CustomerID	FirstName	LastName	CustomerOrderID
1	Andy	Morrison	1
1	Andy	Morrison	2
2	Virginia	Reeder	3
3	Bruce	Laster	4
3	Bruce	Laster	5
4	Ruth	Sands	NULL
5	Michael	Villarreal	NULL
NULL	NULL	NULL	6
NULL	NULL	NULL	7

- Self-Joining is a technique which we're able to join same table with itself and to do that we can use any one of our joins

```
SELECT
        CONCAT(c.FirstName, c.LastName) [EmployeeName],
        CONCAT(m.FirstName, m.LastName) [ManagerName]
FROM
        Employee e
INNER JOIN
        Employee m ON e.ManagerID = m.EmployeeID
```



Employee

EmployeeID	ManagerID	FirstName	LastName
1	NULL	Andy	Morrison
2	1	Virginia	Reeder
3	2	Bruce	Laster
4	3	Ruth	Sands
5	4	Michael	Villarreal

7. Distinct Data

- Whichever columns you provide in the **SELECT DISTINCT** statement will be used to check for uniqueness:

```
SELECT DISTINCT [your column(s)]
FROM [your table]
WHERE [condition(s)]
ORDER BY [your column(s)]
```

- Example:

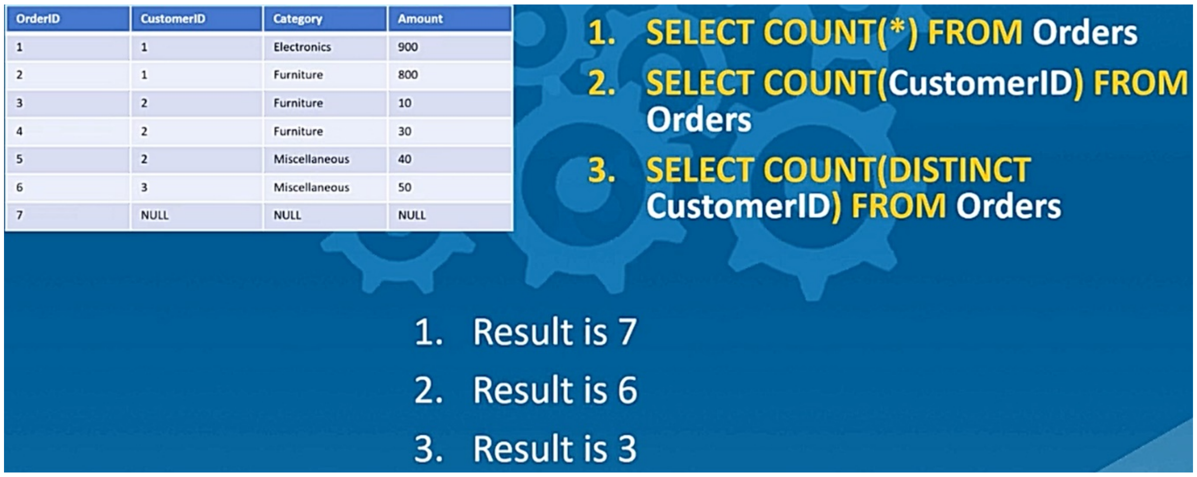
```
SELECT DISTINCT JobTitle FROM HumanResources.Employee

SELECT DISTINCT Gender, JobTitle FROM HumanResources.Employee
```

8. Aggregate Functions

- The most commonly used aggregate functions which are pre-defined in SQL for us:
  - **AVG**
  - **MAX**
  - **MIN**
  - **SUM**
  - **COUNT**

    ```
    SELECT COUNT (*) FROM Orders
    ```



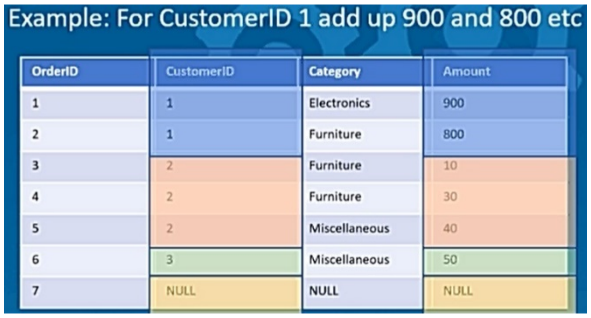
OrderID	CustomerID	Category	Amount
1	1	Electronics	900
2	1	Furniture	800
3	2	Furniture	10
4	2	Furniture	30
5	2	Miscellaneous	40
6	3	Miscellaneous	50
7	NULL	NULL	NULL

1. SELECT COUNT(*) FROM Orders
2. SELECT COUNT(CustomerID) FROM Orders
3. SELECT COUNT(DISTINCT CustomerID) FROM Orders

1. Result is 7
2. Result is 6
3. Result is 3

- **GROUP BY** allows us to separate our data into groups based on columns:

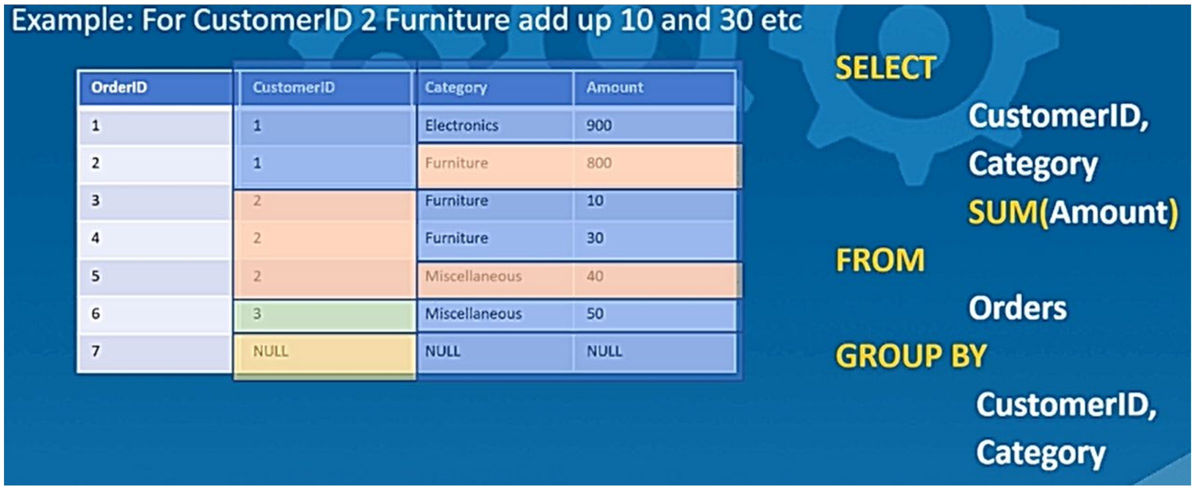
```
SELECT
        CustomerID,
        SUM(Amount)
FROM
        Orders
GROUP BY
        CustomerID
```



Example: For CustomerID 1 add up 900 and 800 etc

OrderID	CustomerID	Category	Amount
1	1	Electronics	900
2	1	Furniture	800
3	2	Furniture	10
4	2	Furniture	30
5	2	Miscellaneous	40
6	3	Miscellaneous	50
7	NULL	NULL	NULL

- We can group results by more than one column:



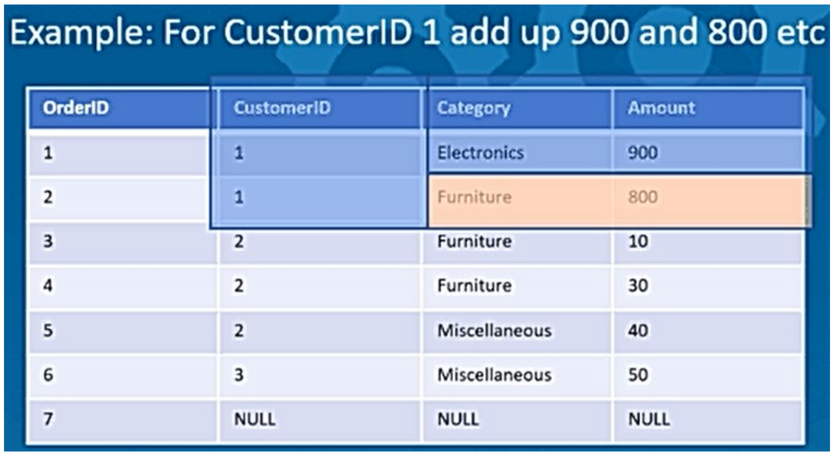
Example: For CustomerID 2 Furniture add up 10 and 30 etc

OrderID	CustomerID	Category	Amount
1	1	Electronics	900
2	1	Furniture	800
3	2	Furniture	10
4	2	Furniture	30
5	2	Miscellaneous	40
6	3	Miscellaneous	50
7	NULL	NULL	NULL

```
SELECT
        CustomerID,
        Category
        SUM(Amount)
FROM
        Orders
GROUP BY
        CustomerID,
        Category
```

- Just the way we can select which rows to get back with the `WHERE` reserved keyword, we can decide to get back with a new reserved keyword called **HAVING**.

```
SELECT
        CustomerID
        Category,
        SUM(Amount)
FROM
        Orders
GROUP BY
        CustomerID,
        Category
HAVING
        SUM(Amount) > 100
```



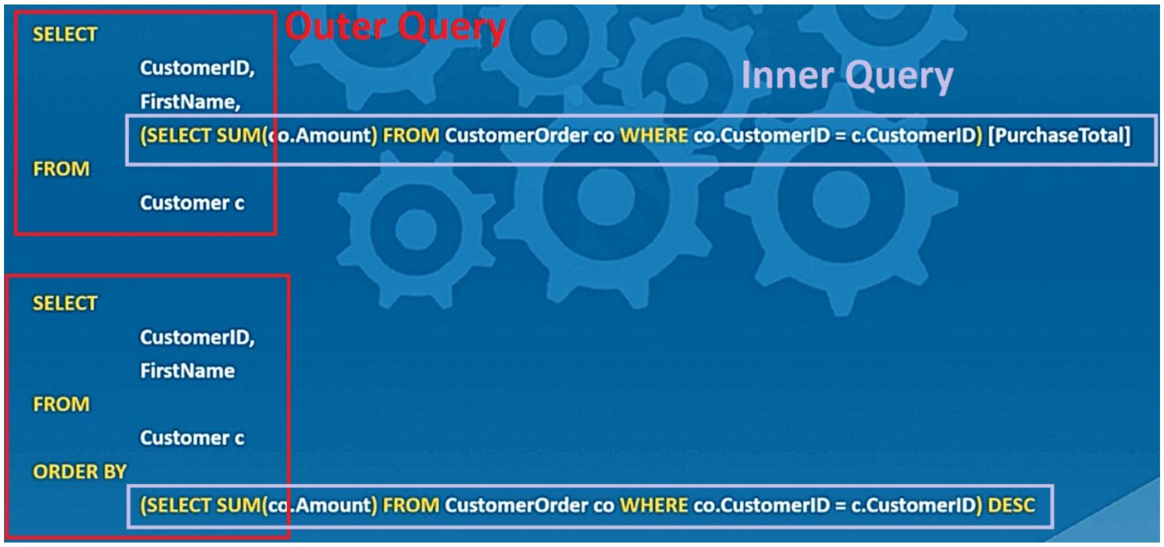
Example: For CustomerID 1 add up 900 and 800 etc

OrderID	CustomerID	Category	Amount
1	1	Electronics	900
2	1	Furniture	800
3	2	Furniture	10
4	2	Furniture	30
5	2	Miscellaneous	40
6	3	Miscellaneous	50
7	NULL	NULL	NULL

9. Subqueries

- A subquery is a `SELECT` statement which is included in another `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement or in another subquery entirely
- A subquery can be included in any part of the `SELECT` statement except `GROUP BY`
- The only difference is the way it's placed; part of the column list, tables, or conditions:
  - When used as part of the columns we are talking about `SELECT` and `ORDER BY`
  - When used as part of the tables we are talking about the `FROM`
  - When used as part of the conditions we are talking about `WHERE` and `HAVING`
- We have two types of queries:
  - **Inner Query** – is the inner one which is nested
  - **Outer Query** – is the one outside which uses the inner query

*Note: We can select run our Inner Query independently


Outer Query
Inner Query

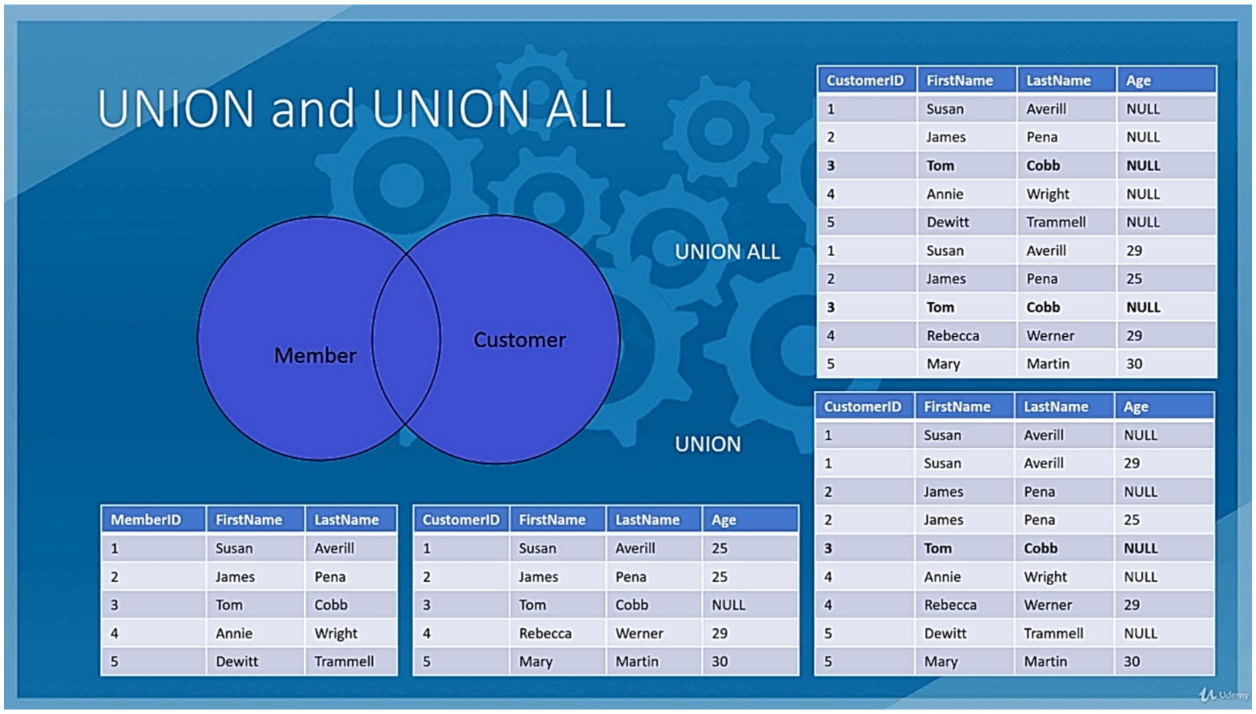
```
SELECT
        CustomerID,
        FirstName,
        (SELECT SUM(co.Amount) FROM CustomerOrder co WHERE co.CustomerID = c.CustomerID) [PurchaseTotal]
FROM
        Customer c

SELECT
        CustomerID,
        FirstName
FROM
        Customer c
ORDER BY
        (SELECT SUM(co.Amount) FROM CustomerOrder co WHERE co.CustomerID = c.CustomerID) DESC
```

10. Set Operations

- Set operations allow us to combine `SELECT` queries together

- Unlike subqueries we are not nesting SELECT statements within each other – we have two or more separate SELECT queries which get their own results and the via set operations those results get combined
- We have the following set operations available to us in T-SQL:
  - **UNION** – Combines result sets of two or more SELECT queries
  - **UNION ALL** - The difference between UNION and UNION ALL is that in UNION, if two or more records are completely similar, just one of them will be returned however, in UNION ALL, all of them will be returned
  - **INTERSECT** – It allows you to see only the common matches between queries
  - **EXCEPT** – It allows you to get the result set which is in only one set and completely not in the other



UNION and UNION ALL

MemberID	FirstName	LastName
1	Susan	Averill
2	James	Pena
3	Tom	Cobb
4	Annie	Wright
5	Dewitt	Trammell

CustomerID	FirstName	LastName	Age
1	Susan	Averill	25
2	James	Pena	25
3	Tom	Cobb	NULL
4	Rebecca	Werner	29
5	Mary	Martin	30

UNION ALL

CustomerID	FirstName	LastName	Age
1	Susan	Averill	NULL
2	James	Pena	NULL
3	**Tom**	**Cobb**	**NULL**
4	Annie	Wright	NULL
5	Dewitt	Trammell	NULL
1	Susan	Averill	29
2	James	Pena	25
3	**Tom**	**Cobb**	**NULL**
4	Rebecca	Werner	29
5	Mary	Martin	30

UNION

CustomerID	FirstName	LastName	Age
1	Susan	Averill	NULL
1	Susan	Averill	29
2	James	Pena	NULL
2	James	Pena	25
3	**Tom**	**Cobb**	**NULL**
4	Annie	Wright	NULL
4	Rebecca	Werner	29
5	Dewitt	Trammell	NULL
5	Mary	Martin	30

```
SELECT
        MemberID[CustomerID],
        FirstName,
        LastName,
        NULL[Age]
FROM
        Member
UNION
SELECT
        CustomerID,
        FirstName,
        LastName,
        Age
```

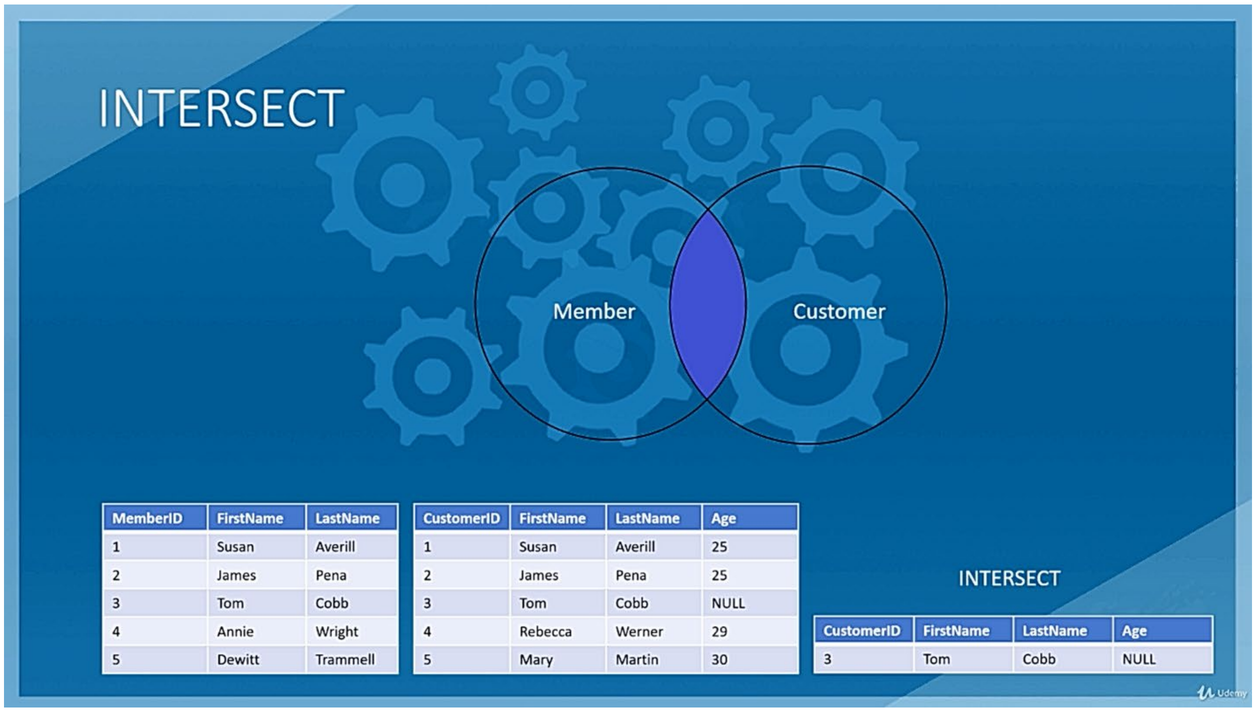
```
SELECT
        MemberID[CustomerID],
        FirstName,
        LastName,
        NULL[Age]
FROM
        Member
UNION ALL
SELECT
        CustomerID,
        FirstName,
        LastName,
        Age
```

```
FROM
        Customer
```

```
FROM
        Customer
```

*Note: There are three rules to follow for these statements to work (these rules apply for all set operations):

1. The columns of all queries being combined, need to be into the same exact order.
2. There needs to be the same number of columns and all queries
3. All columns in queries being combined need to be compatible or have the exact data type



INTERSECT

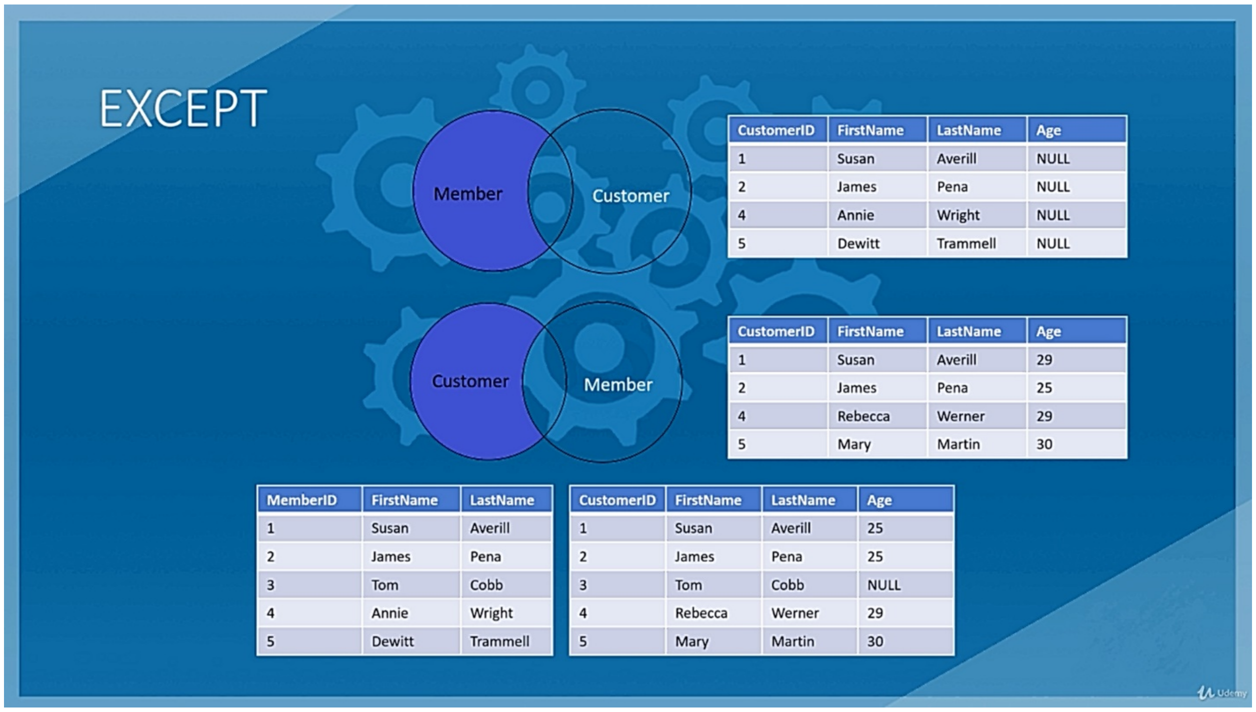
MemberID	FirstName	LastName
1	Susan	Averill
2	James	Pena
3	Tom	Cobb
4	Annie	Wright
5	Dewitt	Trammell

CustomerID	FirstName	LastName	Age
1	Susan	Averill	25
2	James	Pena	25
3	Tom	Cobb	NULL
4	Rebecca	Werner	29
5	Mary	Martin	30

INTERSECT

CustomerID	FirstName	LastName	Age
3	Tom	Cobb	NULL

```
SELECT
        MemberID[CustomerID],
        FirstName,
        LastName,
        NULL[Age]
FROM
        Members
INTERSECT
SELECT
        CustomerID,
        FirstName,
        LastName,
        Age
FROM
        Customer
```



EXCEPT

CustomerID	FirstName	LastName	Age
1	Susan	Averill	NULL
2	James	Pena	NULL
4	Annie	Wright	NULL
5	Dewitt	Trammell	NULL

CustomerID	FirstName	LastName	Age
1	Susan	Averill	29
2	James	Pena	25
4	Rebecca	Werner	29
5	Mary	Martin	30

MemberID	FirstName	LastName
1	Susan	Averill
2	James	Pena
3	Tom	Cobb
4	Annie	Wright
5	Dewitt	Trammell

CustomerID	FirstName	LastName	Age
1	Susan	Averill	25
2	James	Pena	25
3	Tom	Cobb	NULL
4	Rebecca	Werner	29
5	Mary	Martin	30

11. Manipulate Data

- Manipulate Data – Insert:

```
INSERT INTO [Person].[PhoneNumberType]
(
   [Name],
   [ModifiedDate]
)
```

```
VALUES
(
   ‘Emergency Phone Number’,
   GETDATE()
)
```

- Manipulate data – Update:
```
UPDATE [Person].[PhoneNumberType]
SET [Name] = ‘Emergency’
WHERE PhoneNumberTypeID = 4
```

- Manipulate data – Delete:
```
DELETE FROM [Person].[PhoneNumberType]
WHERE PhoneNumberTypeID = 4
```

Section 7 - DCL in SQL Server

1. Data Control Language Commands

- This part of SQL language is used to enforce database security in a multiple user database environment
- Privileges in SQL Server are access rights given to the database user for a database object. There are two types of privileges:
    1. **System** privileges – These set of privileges allow the user to manipulate database objects by using `CREATE`, `ALTER` or `DROP` commands
    2. **Object** privileges – These set of privileges allow the user to manipulate data within the database objects or access the data by using `SELECT`, `INSERT`, `UPDATE`, `DELETE` and `EXECUTE`

- Roles are basically a group of privileges that can be assigned to many users

- When assigning a privilege to a user you only do it for that user but when there are many users for a given database it could become very difficult to manage all the privileges for each person

- A role could define a set of privileges for multiple users so that when you change a privilege on the role it automatically synchronizes it up to all the users attached to it

2. Grant & Revoke

- Two main commands are used to accomplish these tasks:
    - **GRANT**
    - **REVOKE**
    - Only the database administrator or owner of the database object can grant/revoke privileges on a database object

```
GRANT [Name Of The Privilege]
ON [Database Object]
TO [Username or PUBLIC or Role Name]

REVOKE [Name Of The Privilege]
ON [Database Object]
FROM [Username or PUBLIC or Role Name]
```