Understanding Core Database Concepts

Database

- A ***database (db)*** is an organized collection of data, typically stored in electronic format.
  - It allows you to input data, organize the data and retrieve the data quickly.
  - Traditional databases are organized by fields, records, and files.

Database Files

- Microsoft SQL server uses three types of files to store the database:
  - Primary data files, with an .mdf extension, which contain user-defined objects, such as tables and views, as well as system tables.
  - Secondary data files, with an .ndf extension, on separate physical hard disks to give your database more room.
  - Transaction log files use an .ldf extension and don’t contain any objects such as tables or views.

Database Management System (DBMS)

- Most users do not access the databases directly, Instead, users use a ***database management system (DBMS)*** to access the databases indirectly.
- DBMS is a collection of programs that enables you to enter, organize, and select data in a database.

Types of Databases

- A ***flat type database*** are considered flat because they are two dimensional *tables* consisting of rows and columns.
- A **hierarchical database** design is similar to a tree structure (such as a family tree).
  - Each parent can have multiple children, but each child can have only one parent.
- A **relational database** is similar to a hierarchical database in that data is stored in tables and any new information is automatically added into the table without the need to reorganize the table itself.
  - Different from hierarchical database, a table in a relational database can have multiple parents.

Database Servers

- Databases are often found on ***database servers*** so that they can be accessed by multiple users and to provide a high-level of performance.
- A popular database server is Microsoft SQL Server.

Web Resource

- https://docs.microsoft.com/en-us/sql/relational-databases/tables/tables?view=sql-server-ver15
- https://database.guide/how-to-create-a-table-in-sql-server/
- https://docs.microsoft.com/en-us/sql/t-sql/lesson-1-creating-database-objects?view=sql-server-ver15

Data Types

- A ***data type*** is an attribute that specifies the type of data that an object can hold and it also specifies how many bytes each data type takes up.
- SQL Server 2008's built-in data types are organized by these general categories:
  - Exact Numbers
  - Approximate Numbers
  - Date and Time
  - Character Strings
  - Unicode Character Strings
  - Binary Strings
  - Other Data Types
  - CLR Data Types
  - Spatial Data Types

Data Types

- Money (Numeric) - This numeric data type is used in places where you want money or currency.
- Datetime - The datetime date and time data type is used to store date and time data in many different formats
- Integer - The int numeric data type is used to store mathematical computations and is used when you do not require a decimal point output.

Data Types

- Varchar - This character string data type is commonly used in databases where you are supporting English attributes
  - nvarchar – Used for non-English languages
- Boolean - Otherwise known as a bit data type.
- Float - This numeric data type is commonly used in the scientific community and is considered an approximate-number data type.

Data Types

Data Type	*Use / Description*	*Storage*
Exact Numerics:		
Bit	Integer with either a 1 or 0 value. 9 up to 16-bit columns are stored as 2 bytes. The storage size increases as the number of bit columns used increases.	1 byte
Tinyint	Integer data from 0 to 255	1 byte
Smallint	Integer data from -2^15(-32,768) to 2^15-1(32,767)	2 bytes
Int	Integer data from -2^31(-2,147,483,648) to 2^31-1(2m147m483m647)	4 bytes
Bigint	Integer data from -2^63(-9,223,372,036,854,775,808) to 2^63-1(9,223,372,036,854,775,807).	8 bytes
Numeric	Fixed precision and scale. Valid values are from -10^38+1 through 10^38-1.	Varies
Decimal	Fixed precision and scale. Valid values are from -10^38+1 through 10^38-1	Varies
Smallmoney	Monetary or currency values from -214,748.3648 to 214,748.3647	4 bytes
Money	Monetary or currency values from -922,337,203,685,477.508 to 922,337,203,685,477.5807	8 bytes

Implicit and Explicit Conversions

- SQL Server supports implicit conversions, which can occur without specifying the actual callout function (cast or convert).
- Explicit conversions actually require you to use the functions cast or convert specifically.

Syntax

```
cast(source-value AS destination-type)
--
Therefore, to convert the count variable to a float, you would
 use the following command:
cast(count AS float)
--The syntax of the convert function is:
CONVERT ( data_type [ ( length ) ], expression [,style ] )
--
where you can specify how many digits or characters the value
will be. For example:
CONVERT(nvarchar(10), OrderDate, 101)
--
This will convert the OrderDate, which is a DateTime data type
 to nvarchar value.
--The 101 style represents USA date with century. mm/dd/yyyy.
```

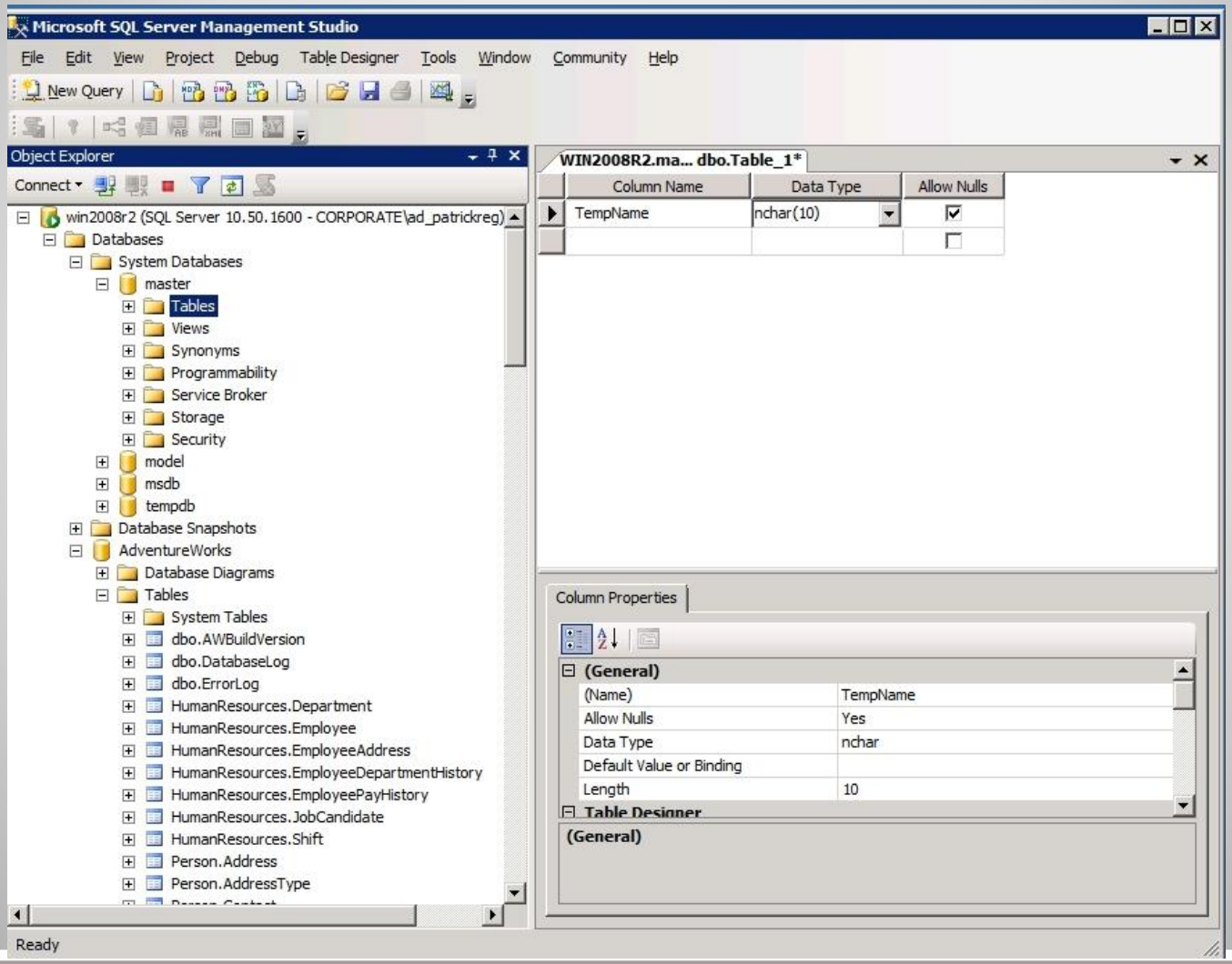
More Details

- https://docs.microsoft.com/en-us/sql/t-sql/functions/cast-and-convert-transact-sql?view=sql-server-ver15

SQL Server Management Studio (SSMS)

- The central feature of SSMS is the Object Explorer, which allows the user to browse, select and manage any of the objects within the server.

SQL Server Management Studio (SSMS)



Data Definition Language (DDL)

- *Data Definition Language (DDL)* is a subset of the Transact-SQL language.
- It deals with creating database objects like tables, constraints, and stored procedures.
- Some DDL commands include:
  - USE: Changes the database context.
  - CREATE: Creates a SQL Server database object (table, view or stored procedure)
  - ALTER: Changes an existing object
  - DROP: Removes an object from the database

Create/Drop Database

- Syntax

```
CREATE DATABASE database_name
    [ ON [ PRIMARY ] [ < filespec >]]
[ LOG ON [ < filespec >   ]]

    < filespec > ::=
     ( [ NAME = logical_file_name , ]    FILENAME = 'os_fil
e_name'     [ , SIZE = size ]   [ , MAXSIZE = { max_size | U
NLIMITED } ]     [ , FILEGROWTH = growth_increment ] ) [ ,.
..n ]
```

- Example

```
CREATE DATABASE Personnel
DROP DATABASE Personnel
```

System Tables

- System views belong to the sys schema. Some of these system tables include:
  - sys.Tables
  - sys.Columns
  - sys.Databases
  - sys.Constraints
  - sys.Views
  - sys.Procedures
  - sys.Indexes
  - sys.Triggers
  - sys.Objects

USE

- The USE command
- changes the database context to the specified database or database snapshot.

```
USE TESTDB
```

CREATE TABLE

The CREATE statement allows you to create a variety of database objects, including tables, views, and stored procedures.

```
USE [AdventureWorks]
GO
CREATE TABLE [dbo].[Planets](
    [IndvidualID] [int] NOT NULL,
    [PlanetName] [varchar](50) NULL,
    [PlanetType] [varchar](50) NULL,
    [Radius] [varchar](50) NULL,
    [TimeCreated] [datetime] NULL
) ON [PRIMARY]
GO
```

You can use the following statement to view the structure of `Planets` table:

```
sp_help 'dbo.Planets'
```

Data Integrity

- Is enforced to ensure that the data in a database is accurate, consistent, and reliable.
- Is broadly classified into the following categories:
  - ✓Entity integrity
  - ✓Domain integrity
  - ✓Referential integrity
  - ✓User-defined integrity

Data Integrity

- When creating tables, SQL Server allows you to maintain integrity by:
  - Applying constraints.
  - Applying rules.
  - Using user-defined types.

Constraints

- Define rules that must be followed to maintain consistency and correctness of the data.
- Can be either created while creating a table or added later.
- Check the existing data if added after the creation of the table.
- Can be created by using either of the following statements:
  - CREATE TABLE statement
  - ALTER TABLE statement

Constraint Syntax

```
CREATE TABLE table_name
(
    column_name CONSTRAINT constraint_name constraint_type
    [,CONSTRAINT constraint_name constraint_type]
)
```

Constraint can be divided into 4 types

- Primary key constraint
- Unique constraint
- Foreign key constraint
- Check constraint
- Default constraint

Primary Key Constraint

- Is defined on a column or a set of columns whose values uniquely identify all the rows in a table.
- Ensures entity integrity.
- Syntax:

```
CREATE TABLE table_name
        (
    col_name [CONSTRAINT constraint_name  PRIMARY KEY
        [CLUSTERED|NONCLUSTERED]
          col_name [, col_name [, col_name [, …]]]
        )
```

- Example

```
CREATE TABLE dbo.Persons
(
  PersonId int CONSTRAINT pkProjectCode PRIMARY KEY,
  …
  …
)
```

Composite Key

- A ***composite primary key*** occurs when you define more than one column as your primary key. Although many database administrators do not use them (nor are they sometimes even aware of them), they play an integral part in the designing of a good, solid data model.

```
CREATE TABLE CUSTOMER
(
    FirstName varchar(50) not null,
    LastName varchar(50) not null,
    CONSTRAINT pkCustomer PRIMARY KEY(FirstName,LastName)
)
```

Unique constraint:

- Is used to enforce uniqueness on non-primary key columns.
- Is similar to the primary key constraint except that it allows one NULL row.
- Syntax:

```
CREATE TABLE table_name
 (
col_name [CONSTRAINT constraint_name UNIQUE [CLUSTERED | NONCLU
STERED]
    (col_name [, col_name [, col_name [, …]]])
col_name [, col_name [, col_name [, …]]]
)
```

- Example

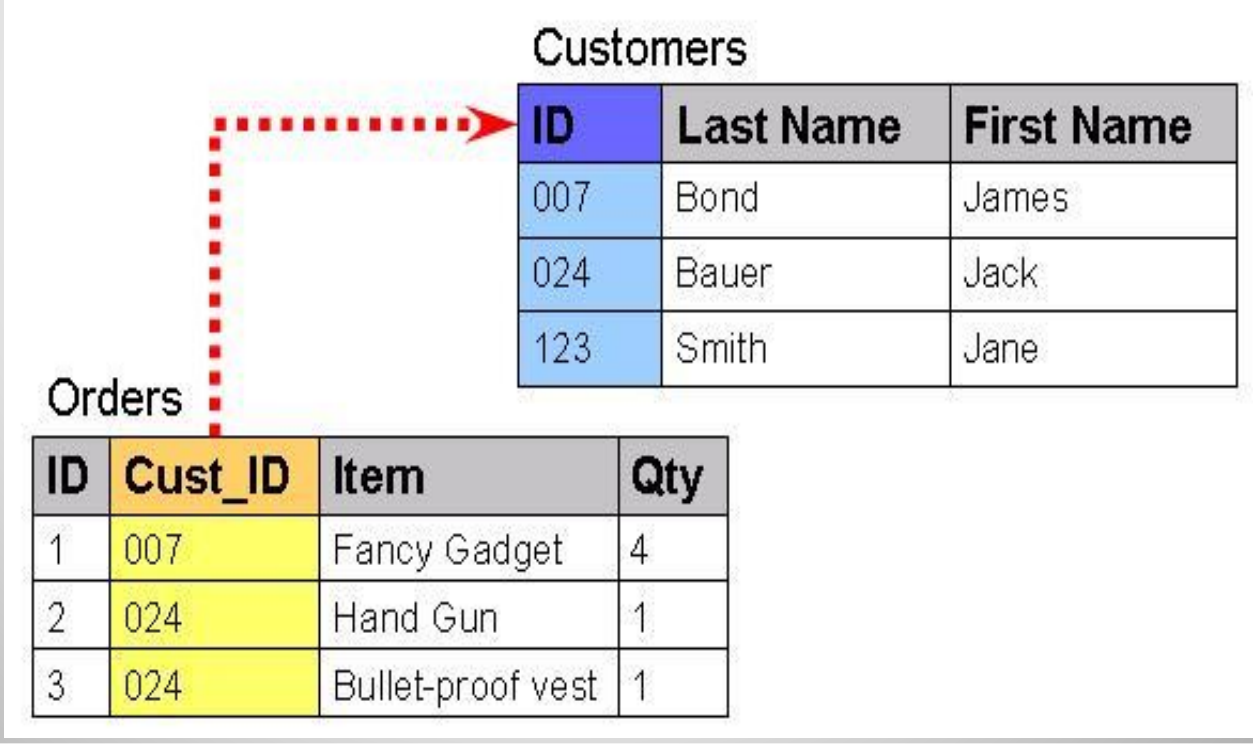
```
CREATE TABLE dbo.Persons
(
  PersonId int CONSTRAINT pkProjectCode PRIMARY KEY,
  PersonMobile varchar(30) constraint unMobile
  …
  …
)
```

Referential Integrity

- One of the most common mistakes of database manipulating is the accidental loss of entire tables.
- The best way to avoid this type of situation in the first place is to ensure your database is using ***referential integrity***.
- Referential integrity does not allow deletion of tables, unless they were actually at the end of the relationship.

Foreign key constraint:

- Removes the inconsistency in two tables when the data in one table depends on the data in another table.
- Always refers the primary key column of another table, as shown in the following figure.



Foreign Key

- Syntax

```
CREATE TABLE table_name
    (
    col_name [CONSTRAINT constraint_name FOREIGN KEY (co
l_name [, col_name [, …]])
    REFERENCES table_name (column_name [, column_name [,
 …]])]
    (col_name [, col_name [, col_name [, …]]])
    col_name [, col_name [, col_name [, …]]]
    )
```

Foreign Key Example

```
CREATE TABLE dbo.Jobs
(
    JobId int CONSTRAINT pkJobId PRIMARY KEY,
    PersonId int CONSTRAINT fkPersonId
    FOREIGN KEY REFERENCES dbo.Persons(PersonId)
    ..
)
```

Check constraint:

- Enforces domain integrity by restricting the values to be inserted in a column.
- Syntax

```
CREATE TABLE table_name
(
col_name [CONSTRAINT constraint_name] CHECK (expression)
(col_name [, col_name [, …]])
.
)
```

- Can be specified by using the following keywords:
  - IN
  - LIKE
  - BETWEEN

In

```
CREATE TABLE dbo.Jobs
(
    JobId int CONSTRAINT pkJobId PRIMARY KEY,
    PersonId int CONSTRAINT fkPersonId FOREIGN KEY REFERENCES
    dbo.Persons(PersonId),
    [Location] varchar(50) chklocation CHECK(Location IN('Ahm
edabad','Mumbai','Calcutta','Delhi','Pune')),
    ..
)
```

Like

- Is used to ensure that the values entered in specific columns are of a certain pattern.

```
CREATE TABLE dbo.Persons
(
  PersonId int CONSTRAINT pkProjectCode PRIMARY KEY,
  PersonMobile varchar(30) constraint unMobile,
  DeptCode char(4) CHECK (DeptCode LIKE '[0-9][0-9][0-9][0-9]')
  …
  …
)
```

Between

- Is used to specify a range of constant expressions by using the BETWEEN keyword.

```
CREATE TABLE dbo.Jobs
(
    JobId int CONSTRAINT pkJobId PRIMARY KEY,
    PersonId int CONSTRAINT fkPersonId FOREIGN KEY REFERENCES
 dbo.Persons(PersonId),
    [Location] varchar(50) chklocation CHECK(Location IN('Ahm
edabad','Mumbai','Calcutta','Delhi','Pune')) ,
    SalaryRange int CHECK (Salary BETWEEN 30000 AND 60000)
    ..
)
```

Default constraint:

- Can be used to assign a constant value to a column, and the user need not insert values for such a column.
- Syntax

```
    CREATE TABLE table_name
    (
    col_name [CONSTRAINT constraint_name] DEFAULT (constant_expr
ession | NULL)
    (col_name [, col_name [, …]])
    .
    )
```

Default Constraint Example

```
CREATE TABLE dbo.Jobs
(
    JobId int CONSTRAINT pkJobId PRIMARY KEY,
    PersonId int CONSTRAINT fkPersonId FOREIGN KEY REFERENCES
    dbo.Persons(PersonId),
    [Location] varchar(50) chklocation CHECK(Location IN('Ahmeda
bad','Mumbai','Calcutta','Delhi','Pune'))
    CONSTRAINT chkDefLocation DEFAULT 'Ahmedabad',
    SalaryRange int CHECK (Salary BETWEEN 30000 AND 60000)
    ..
)
```

Creating Rule

- Help in enforcing domain integrity for columns or user-defined data types.
- Can be applied to the column or the user-defined data type before an INSERT or UPDATE statement is issued.
- Are used to implement business-related restrictions or limitations.
- Can be created by using the CREATE RULE statement.
- Syntax:

```
CREATE RULE [ schema_name . ] rule_name
AS condition_expression
[ ; ]
```

- https://docs.microsoft.com/en-us/sql/t-sql/statements/create-rule-transact-sql?view=sql-server-ver15

Binding Rule

- you need to activate the rule by using a stored procedure, sp_bindrule.

```
sp_bindrule [ @rulename = ] 'rule' , [ @objname = ]
'object_name' [ , [ @futureonly = ] 'futureonly_flag' ]
```

- https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-bindrule-transact-sql?view=sql-server-ver15

User-defined data types:

- Are custom data types defined by the users with a custom name.
- Are basically named objects with the following additional features:
  - Defined data type and length
  - Defined nullability
  - Predefined rules that may be bound to the user-defined data types
  - Predefined default value that may be bound to the user-defined data types

Create Type

- Can be created by using the CREATE TYPE statement.
- Syntax:

```
CREATE TYPE [ schema_name. ] type_name { FROM
base_type [ ( precision
[ , scale ] ) ] [ NULL | NOT NULL ] } [ ; ]
```

- Example

```
CREATE TYPE DSCRP
FROM varchar(100) NOT NULL ;
```

Alter Statement

- The ALTER statement changes an existing object; you can use it to add or remove columns from a table, as shown in the following example.

```
ALTER TABLE Shirt
ADD Price Money;
GO
```

```
ALTER TABLE Persons
ALTER COLUMN DateOfBirth year;
Go
```

```
ALTER TABLE Persons
DROP COLUMN DateOfBirth;
```

- When working with these statements, be careful that you don't confuse ALTER with UPDATE.
- Remember, ALTER changes the object definition, but UPDATE changes the data in the table.

DROP

- The DROP statement actually removes an object from a database, but if other objects are dependent on the object you are attempting to remove, this statement will fail and an error will be raised.

```
DROP VIEW Size;
GO
DROP TABLE Person.Contact
```

- Remember to not confuse DROP, which removes an object from the database, with DELETE,which deletes data from within a table.

Truncate and Delete

- The DELETE statement is used to delete rows from a table, but it does not free the space containing the table.
- In comparison, the SQL TRUNCATE command is used to both delete the rows from a table and free the space containing the table.

```
--
Thus, to delete all rows from a table named User, you would ente
r the following command:
DELETE FROM user;
--
Similarly, to delete an employee with the identification number
200 from the User table, you
--would enter the following command:
DELETE FROM user WHERE id = 200;
```

Data Manipulation Language (DML)

- *Data Manipulation Language* (DML) is the language element which allows you to use the core statements:
  - SELECT: Retrieves rows from the database and enables the selection of one or many rows or columns from one or many tables in SQL Server.
  - INSERT: Adds one or more new rows to a table or a view in SQL Server.
  - UPDATE: Changes existing data in one or more columns in a table or view.
  - DELETE: Removes rows from a table or view.
  - MERGE: Performs insert, update, or delete operations on a target table based on the results of a join with a source table.

INSERT INTO Statement

- It is possible to write the INSERT INTO statement in two ways.
- The first way specifies both the column names and the values to be inserted:

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

- If you are adding values for all the columns of the table, you do not need to specify the column names in the SQL query

```
INSERT INTO table_name
VALUES (value1, value2, value3, ...);
```

INSERT Multiple Rows

- To add multiple rows to a table at once, you use the following form of the INSERT statement:

Syntax

```
INSERT INTO table_name
(column_list)
VALUES
    (value_list_1),
    (value_list_2),
    ...
    (value_list_n);
```

Example

```
--table structure
create table Employee
(
Id int primary key,
Name varchar(50)
)
--insert muliple rows
INSERT INTO Employee
(Id,Name)
VALUES
    (1,'John'),
    (2,'Rita'),
    (3,'Meena')
```

Update Command

- Changes existing data in a table or view in SQL Server

```
USE AdventureWorks2012;
UPDATE Person.Address
SET ModifiedDate = GETDATE();
--Updating multiple columns
--basic update
UPDATE Sales.SalesPerson
SET Bonus = 6000, CommissionPct = .10, SalesQuota = NULL;
--Limiting the Rows that Are Updated
UPDATE Production.Product
SET Color = N'Metallic Red'
WHERE Name LIKE N'Road-250%' AND Color = N'Red';
--Specifying a computed value
UPDATE Production.Product
SET ListPrice = ListPrice * 2;
--Specifying a compound operator
DECLARE @NewPrice INT = 10;
UPDATE Production.Product
SET ListPrice += @NewPrice
WHERE Color = N'Red';
```

Update command

- https://docs.microsoft.com/en-us/sql/t-sql/queries/update-transact-sql?view=sql-server-ver15#OtherTables

Merge

- Suppose, you have two table called source and target tables, and you need to update the target table based on the values matched from the source table. There are three cases:
  - The source table has some rows that do not exist in the target table. In this case, you need to insert rows that are in the source table into the target table.
  - The target table has some rows that do not exist in the source table. In this case, you need to delete rows from the target table.
  - The source table has some rows with the same keys as the rows in the target table. However, these rows have different values in the non-key columns. In this case, you need to update the rows in the target table with the values coming from the source table.

Merge syntax

```
MERGE target_table USING source_table
ON merge_condition
WHEN MATCHED
    THEN update_statement
WHEN NOT MATCHED
    THEN insert_statement
WHEN NOT MATCHED BY SOURCE
    THEN DELETE;
```

Merge Example

sales.category_staging

category_id	category_name	amount
1	Children Bicycles	15000.00
3	Cruisers Bicycles	13000.00
4	Cyclocross Bicycles	20000.00
5	Electric Bikes	10000.00
6	Mountain Bikes	10000.00

INSERT

sales.category

category_id	category_name	amount
1	Children Bicycles	15000.00
2	Comfort Bicycles	25000.00
3	Cruisers Bicycles	13000.00
4	Cyclocross Bicycles	10000.00

DELETE

UPDATE

category_id	category_name	amount
1	Children Bicycles	15000.00
3	Cruisers Bicycles	13000.00
4	Cyclocross Bicycles	20000.00
5	Electric Bikes	10000.00
6	Mountain Bikes	10000.00

Merge Example

```
MERGE sales.category t
    USING sales.category_staging s
ON (s.category_id = t.category_id)
WHEN MATCHED
    THEN UPDATE SET
        t.category_name = s.category_name,
        t.amount = s.amount
WHEN NOT MATCHED BY TARGET
    THEN INSERT (category_id, category_name, amount)
         VALUES (s.category_id, s.category_name, s.amount)
WHEN NOT MATCHED BY SOURCE
    THEN DELETE;
```

DQL (Select)

- The SELECT statement is used for accessing and retrieving data from a database.
- Syntax:

```
SELECT [ALL | DISTINCT] select_column_list
    [INTO [new_table_name]]
    [FROM {table_name | view_name}
    [WHERE search_condition]
```

- To display all the details of the employee

```
SELECT * FROM HumanResources.Employee
```

DQL (Select)

- To view specific details, you can specify the column names in the SELECT statement, as shown in the following query:

```
SELECT EmployeeID, ContactID, LoginID, Title
FROM HumanResources.Employee
```

Customizing the display:

- When report should contain column headings different from those given in the table

```
SELECT 'Department Number'= DepartmentID, ' Department Name'=
Name  FROM HumanResources.Department
--or
SELECT DepartmentID 'Department Number', Name 'Department Name
' FROM HumanResources.Department
--or
SELECT DepartmentID AS 'Department Number', Name AS 'Departmen
t Name' FROM HumanResources.Department
```

Literals & Concatenate

- Literals are string values that are enclosed in single quotes and added to the SELECT statement.

```
--literals
SELECT EmployeeID, 'Designation: ', Title
FROM HumanResources.Employee
--concatenate
SELECT 'snow' + 'ball'
```

- The following SQL query concatenates the data of the Name and GroupName columns of the Department table into a single column:

```
SELECT Name + ' department comes under ' + GroupName + '
 group' AS Department FROM HumanResources.Department
```

Calculating column values:

- Arithmetic operators are used to perform mathematical operations, such as addition, subtraction, division, and multiplication, on numeric columns or on numeric constants.
- SQL Server supports the following arithmetic operations:
  - + (for addition)
  - - (for subtraction)
  - / (for division)
  - * (for multiplication)
  - % (for modulo)

```
SELECT EmployeeID, Rate,
Per_Day_Rate = 8 * Rate
FROM HumanResources.EmployeePayHistory
```

Derived Table

- Derived tables are result sets used as table sources in a query.

```
--Select 2nd Highest salary from employee table
--Ans :*/

SELECT salary
FROM
(SELECT DENSE_RANK() OVER (ORDER BY salary DESC) [d_rank], *
FROM Employee
) [tbl_temp]
WHERE d_rank = 2;

--join
select FirstName, dept.DepartmentName from tbEmployee inner
join (select DepartmentID, DepartmentName from tbDepartment)
as dept  on dept.DepartmentID = tbEmployee.DepartmentID
```

Common Table Expression

- A Common Table Expression, also called as CTE in short form, is a temporary named result set that you can reference within a SELECT, INSERT, UPDATE, or DELETE statement. The CTE can also be used in a View.
- The CTE query starts with a “With” and is followed by the Expression Name. We will be using this expression name in our select query to display the result of our CTE Query and be writing our CTE query definition.
- Syntax

```
--syntax
WITH expression_name [ ( column_name [,...n] ) ]
AS
( CTE_query_definition )
--To view the CTE result we use a Select query with
the CTE expression name.
Select [Column1,Column2,Column3 …..] from
expression_name
--or
Select * from expression_name
```

CTE Example

```
WITH Sales_CTE (SalesPersonID, NumberOfOrders)
AS
(
    SELECT SalesPersonID, COUNT(*)
    FROM Sales.SalesOrderHeader
    WHERE SalesPersonID IS NOT NULL
    GROUP BY SalesPersonID
)
SELECT AVG(NumberOfOrders) AS "Average Sales Per Person"
FROM Sales_CTE;
```

CTE/Derived Table Web Resource

- https://www.sqlshack.com/sql-server-common-table-expressions-cte/
- https://docs.microsoft.com/en-us/sql/t-sql/queries/with-common-table-expression-transact-sql?view=sql-server-ver15
- https://docs.microsoft.com/en-us/sql/ssms/visual-db-tools/add-derived-tables-to-queries-visual-database-tools?view=sql-server-ver15
- https://www.mssqltips.com/sqlservertip/6038/sql-server-derived-table-example/

Where

- Selected rows can be retrieved using the WHERE clause in the SELECT statement.

```
SELECT * FROM HumanResources.Department
WHERE GroupName = 'Research and Development'
```

Comparison Operator/Logical Operator(and/or/not)

```
--comparison operator
SELECT EmployeeID, NationalIDNumber, Title, VacationHours
FROM HumanResources.Employee
WHERE VacationHours < 5
```

```
--Logical Operator
SELECT * FROM HumanResources.Department
WHERE GroupName = 'Manufacturing'
OR GroupName = ' Quality Assurance'
```

Range operator:

- Retrieves data based on a range.
- Are of the following types:
  - BETWEEN
  - NOT BETWEEN

```
SELECT EmployeeID, VacationHours
FROM HumanResources.Employee
WHERE VacationHours BETWEEN 20 AND 50
```

in/not in keyword

- In,Selects values that match any one of the values in a list
- Not in Restricts the selection of values that match any one of the values in a list.

```
SELECT EmployeeID, Title, LoginID FROM  HumanRe
sources.Employee
WHERE Title IN ('Recruiter', 'Stocker')
```

The LIKE keyword

- Is used to search a string by using the following wildcard characters:
  - %
  - _
  - []
  - [^]
- Matches the given character string with the specified pattern.

```
SELECT * FROM HumanResources.Department
WHERE Name LIKE 'Pro%'
```

NULL values:

- Can be retrieved by using the IS NULL keyword with the SELECT statement.

```
SELECT EmployeeID, EndDate
FROM HumanResources.EmployeeDepartmentHistory
WHERE EndDate IS NULL
```

ORDER BY clause:

- Can be used with the SELECT statement to display records in a specific order.
- Displays record in ascending or descending order.
- Syntax

```
SELECT select_list
FROM table_name
[ORDER BY order_by_expression [ASC|DESC]
[, order_by_expression [ASC|DESC]…]
```

- Example

```
SELECT GroupName, DepartmentID, Name
FROM HumanResources.Department
ORDER BY GroupName, DepartmentID
```

TOP keyword

- Can be used with the SELECT statement to retrieve only the first set of rows from the top of a table.
- Syntax

```
SELECT [TOP n [PERCENT]] column_name
[,column_name…]
FROM table_name
WHERE search_conditions
[ORDER BY [column_name[,column_name…]
```

```
SELECT TOP 10 * FROM HumanResources.Employee
```

DISTINCT keyword

- Eliminates the duplicate rows from the result set.
- Syntax:

```
SELECT [ALL|DISTINCT] column_names
FROM table_name
WHERE search_condition
```

- Example

```
SELECT DISTINCT Title FROM
HumanResources.Employee
WHERE Title LIKE 'PR%'
```

UNION Clause

- The ***UNION*** clause allows you to combine the results of any two or more queries into a resulting single set that will include all the rows which belong to the query in that union.

```
SELECT first_name, last_name
FROM employees
WHERE department = 'shipping'
UNION
SELECT first_name, last_name
FROM employees
WHERE hire_date BETWEEN '1-Jan-1990' AND '1-Jan-2000'
```

EXCEPT and INTERSECT Clauses

- The *EXCEPT* clause returns any of those distinct values from the left query which are not also found on the right query.
- The *INTERSECT* clause returns any distinct values not returned by both the query on the left and right sides of this operand.

Functions

- String functions
- Date functions
- Mathematical functions
- Ranking functions
- System functions

String Function

- Are used to manipulate the string values in the result set.
- Are used with the char and varchar data types.
- Syntax

```
SELECT function_name (parameters)
```

String Function List

Function	Description	Example
ASCII	Returns the ASCII value for the specific character	SELECT ASCII('A') –output: 65
CHAR	Returns the character based on the ASCII code	SELECT CHAR(68) –output: D
CHARINDEX	Returns the position of a substring in a string	SELECT CHARINDEX ('S','MICROSOFT SQL SERVER 2012' ,4)
CONCAT	Adds two or more strings together	SELECT CONCAT('John',' ','Smith')
FORMAT	Formats a value with the specified format	DECLARE @d DATETIME = '01/01/2012'; SELECT FORMAT (@d, 'd', 'en-US')
LEFT	Extracts a number of characters from a string (starting from left)	SELECT LEFT ('MICROSOFT SQL SERVER 2012',4)
LEN	Returns the length of a string	SELECT LEN ('John Smith')–ouput: 10
LOWER	Converts a string to lower-case	SELECT LOWER('HELP EVERYONE') –output help everyone
LTRIM	Removes leading spaces from a string	SELECT LTRIM (' John Smith')–output John Smith
PATINDEX	Returns the position of a pattern in a string	SELECT PATINDEX('%SER%','SQL SERVER')–output
REPLACE	Replaces all occurrences of a substring within a string, with a new substring	Select REPLACE ('Dont Respect Girls','Dont','Always') –ouput 'Always Respect Girls'

String Function

Function	Description
RTRIM	Removes trailing spaces from a string
SOUNDEX	Returns a four-character code to evaluate the similarity of two strings
SPACE	Returns a string of the specified number of space characters
STR	Returns a number as string
STUFF	Deletes a part of a string and then inserts another part into the string, starting at a specified position
SUBSTRING	Extracts some characters from a string
TRANSLATE	Returns the string from the first argument after the characters specified in the second argument are translated into the characters specified in the third argument.
TRIM	Removes leading and trailing spaces (or other specified characters) from a string
UNICODE	Returns the Unicode value for the first character of the input expression
REPLICATE	Repeats a string a specified number of times
REVERSE	Reverses a string and returns the result
RIGHT	Extracts a number of characters from a string (starting from right)

String Function (Web Reference)

- https://docs.microsoft.com/en-us/sql/t-sql/functions/string-functions-transact-sql?view=sql-server-ver15

String Function Example

```
SELECT Name = Title + ' ' + left(FirstName,1) + '. ' +
LastName, EmailAddress
FROM Person.Contact
```

Date Function

- Date functions are used to manipulate datetime values
- The following table lists some of the date functions

Function namel	Parameters	Example	Description
dateadd	(date part, number, date)	SELECT dateadd(mm, 3,'2009-01-01')	Returns 2009-04-01, adds 3 months to the date
datename	(date part, date)	SELECT datename(month, convert(datetime,'2005-06-06'))	Returns June, date part from the listed date as a character value)
getdate	()	SELECT getdate()	Returns the current date and time
day	(date)	SELECT day('2009-01-05')	Returns 5, an integer, which represents the day

Date Example

```
SELECT EmployeeID, datename(mm, hiredate)+ ', ' + convert(varchar, datepart(yyyy,
hiredate)) as 'Joining'
FROM HumanResources.Employee

--Sample date : 2014-09-04
--Output : September 4, 2014
SELECT DATENAME(mm, GETDATE()) + SPACE(1) + CAST( DATEPART(dd, GETDATE()) AS
VARCHAR) + ', ' + CAST(DATEPART(YYYY, GETDATE()) AS VARCHAR)

 --Write a query to get the first name and hire date from employees table where
 -- hire date between '1987-06-01' and '1987-07-30'
--Ans : */

SELECT first_name, hire_date
FROM employees
WHERE hire_date >= '1987-06-01' AND hire_date <= '1987-07-30'

--Write a query to get first name, hire date and experience of the employees.
--Ans : */
SELECT first_name, hire_date, DATEDIFF(YEAR, hire_date, GETDATE()) [Experience]
FROM employee
```

Date Web Resource

- https://docs.microsoft.com/en-us/sql/t-sql/functions/date-and-time-data-types-and-functions-transact-sql?view=sql-server-ver15
- Diiferent Datepart

Math Function

- Mathematical functions are used to manipulate the numeric values in a result set.

Function name	Parameters	Example	Description
ceiling	(numeric_expression)	SELECT ceiling(14.45)	Returns 15, the smallest integer greater than or equal to the specified value
exp	(float_expression)	SELECT exp(4.5)	Returns 90.0171313005218, the exponential value of the specified value
floor	(numeric_expression)	SELECT floor(14.45)	Returns14, the largest integer less than or equal to the specified value
power	(numeric_expression, y)	SELECT power(4,3)	Returns 64, which is 4 to the value of 3
round	(numeric_expression, length)	SELECT round(15.789, 2)	Returns 15.790, a numeric expression rounded off to the length specified as an integer expression

Example

```
SELECT EmployeeID,'Hourly Pay Rate' = round(Rate,2)
FROM HumanResources.EmployeePayHistory
```

Ranking Functions

- Generate sequential numbers for each row or to give a rank based on specific criteria.
  - row_number()
  - rank()
  - dense_ rank()

Example

```
--Is used where consecutive ranking values need to be given
based on a specified criteria.
SELECT DENSE_RANK() OVER (ORDER BY MArks DESC) [d_rank], *
FROM ExamResult
--It gives the rank one for the first row and then increments
the value by one for each row
--We get SAME ranks for the row having similar values as well.
SELECT RANK() OVER (ORDER BY MArks DESC) [d_rank], *
FROM ExamResult
--It gives the rank one for the first row and then increments
the value by one for each row.
-- We get different ranks for the row having similar values as
well.
SELECT ROW_NUMBER() OVER (ORDER BY MArks DESC) [d_rank], *
FROM ExamResult
```

More Details

- https://www.sqlshack.com/overview-of-sql-rank-functions/
- https://docs.microsoft.com/en-us/sql/t-sql/functions/ntile-transact-sql?view=sql-server-ver15

System Functions

- Are used to query the system tables.
- Are used to access the SQL Server databases or user-related information.

```
--Displays the host ID of the terminal on
which you are logged onto
SELECT host_id() as 'HostID'
```

System Funciton

- https://docs.microsoft.com/en-us/sql/t-sql/functions/system-functions-transact-sql?view=sql-server-ver15

Aggregate Function

- An aggregate function performs a calculation on a set of values, and returns a single value.
- Except for COUNT(*), aggregate functions ignore null values.
- Aggregate functions are often used with the GROUP BY clause of the SELECT statement.
  - SUM
  - COUNT
  - AVG
  - MAX
  - MIN

```
SELECT SUM(MARKS),COUNT(*),AVG(MARKS) FROM MARKS
```

Group By

- A SELECT statement clause that divides the query result into groups of rows, usually for the purpose of performing one or more aggregations on each group.
- The SELECT statement returns one row per group.

Sales Table

Country	Region	Sales
Canada	Alberta	100
Canada	British Columbia	200
Canada	British Columbia	300
United States	Montana	100

Output Table

Country	Region	TotalSales
Canada	Alberta	100
Canada	British Columbia	500
United States	Montana	100

Group By Example

```
SELECT Country, Region, SUM(sales) AS
TotalSales
FROM Sales
GROUP BY Country, Region;
```

GROUP BY ROLLUP

- Creates a group for each combination of column expressions. In addition, it "rolls up" the results into subtotals and grand totals

```
SELECT Country, Region, SUM(Sales) AS TotalSales FROM Sales GROUP BY
ROLLUP (Country, Region)
```

Country	Region	TotalSales
Canada	Alberta	100
Canada	British Columbia	500
Canada	NULL	600
United States	Montana	100
United States	NULL	100
NULL	NULL	700

Group By

- GROUP BY CUBE ( )
  - GROUP BY CUBE creates groups for all possible combinations of columns. For GROUP BY CUBE (a, b) the results has groups for unique values of (a, b), (NULL, b), (a, NULL), and (NULL, NULL).
- GROUP BY GROUPING SETS ( )
  - The GROUPING SETS option gives you the ability to combine multiple GROUP BY clauses into one GROUP BY clause. The results are the equivalent of UNION ALL of the specified groups.
- Reference Link:
  - https://docs.microsoft.com/en-us/sql/t-sql/queries/select-group-by-transact-sql?view=sql-server-ver15

Having Clause

- Specifies a search condition for a group or an aggregate. HAVING can be used only with the SELECT statement. HAVING is typically used with a GROUP BY clause. When GROUP BY is not used, there is an implicit single, aggregated group.

```
SELECT SalesOrderID, SUM(LineTotal) AS
SubTotal
FROM Sales.SalesOrderDetail
GROUP BY SalesOrderID
HAVING SUM(LineTotal) > 100000.00
ORDER BY SalesOrderID ;
```

SELECT INTO

- SELECT...INTO creates a new table in the default filegroup and inserts the resulting rows from the query into it. To view the complete SELECT syntax,

```
SELECT * INTO ABC FROM ExamResult

SELECT * FROM ABC
```

0 %

Results | Messages

	StudentName	Subject	Marks
1	Lily	Science	80
2	Lily	english	70
3	Isabella	Maths	50
4	Isabella	Science	70
5	Isabella	english	90
6	Olivia	Maths	55
7	Olivia	Science	60
8	Olivia	english	89

JOIN Clause

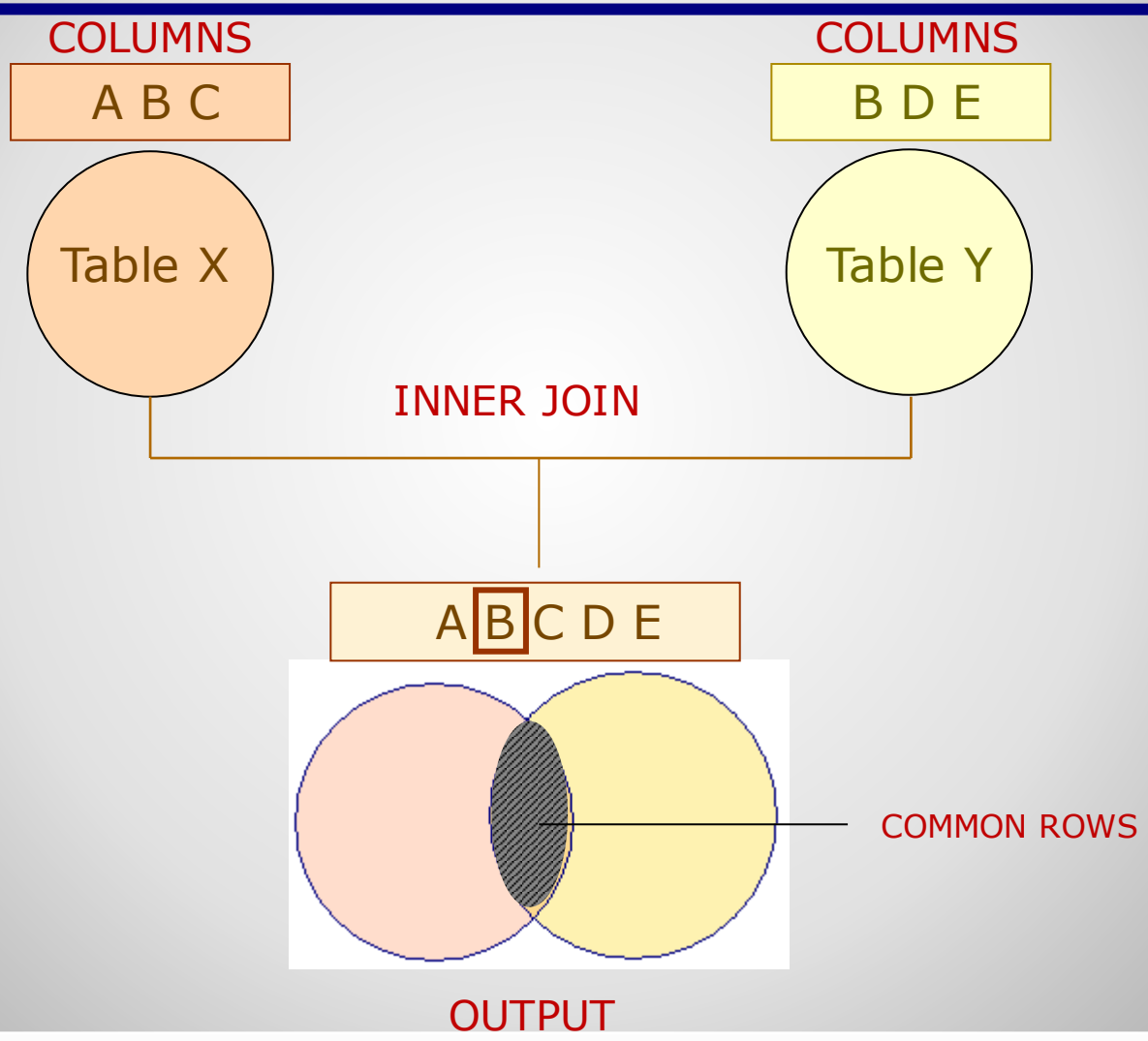
- The *JOIN* clause allows you to combine related data from multiple table sources
- There are three types of JOIN statements you should be aware of:
  - *INNER JOINS* allow you to match related records taken from different source tables
  - *OUTER JOINS* can include records from one or both tables you are querying which do not have any corresponding record(s) to be found in the other table.
  - *CROSS JOINS* return all rows from the one table with all rows from the other table. WHERE conditions should always be included.
  - *Self join*

Inner join:

- Retrieves records from multiple tables after comparing values present in a common column.
- Retrieves only those rows that satisfy the join condition.

```
SELECT column_name, column_name [,column_name]
FROM table1_name JOIN table2_name
ON table1_name.ref_column_name join_operator
table2_name.ref_column_name
```

Working of the Inner Join



Inner Join Example

```
SELECT e.EmployeeID,e.Title,
eph.Rate,eph.PayFrequency
FROM HumanResources.Employee e
JOIN HumanResources.EmployeePayHistory eph
ON e.EmployeeID = eph.EmployeeID
```

An inner join is the default join. Therefore, you can also apply an inner join by using the JOIN keyword.

Outer Join

- Displays the result set containing all the rows from one table and the matching rows from another table.
- Displays NULL for the columns of the related table where it does not find matching records.
- Is of the following types:
  - Left outer join
  - Right outer join
  - Full outer join

Left Outer Join

COLUMNS

A B C

Table X

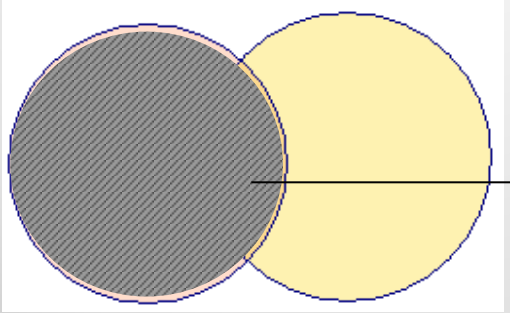
COLUMNS

B D E

Table Y

LEFT OUTER JOIN

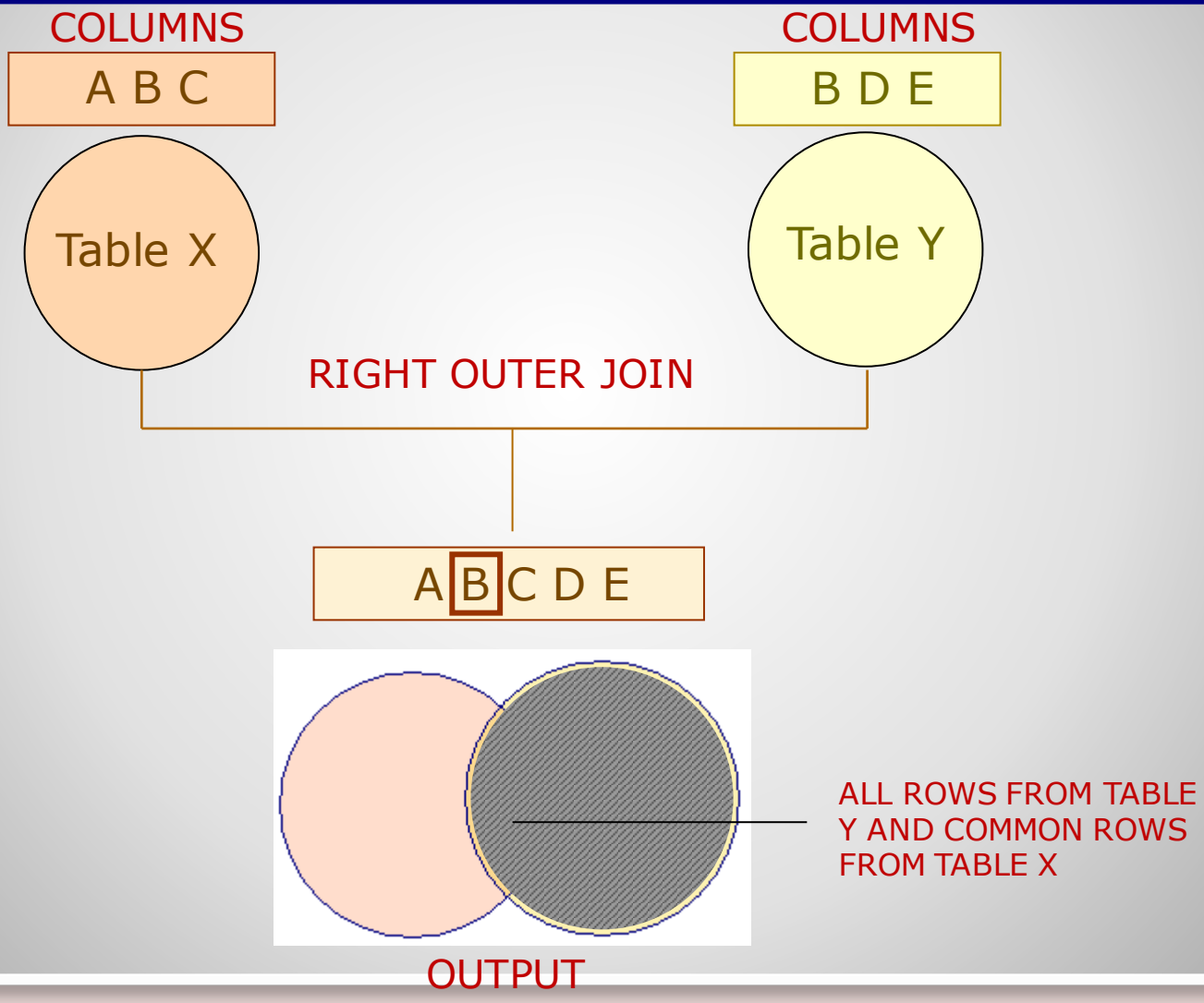
A B C D E



ALL ROWS FROM TABLE X
AND COMMON ROWS
FROM TABLE Y

OUTPUT

Right Outer Join



Full Outer Join

COLUMNS

A B C

Table X

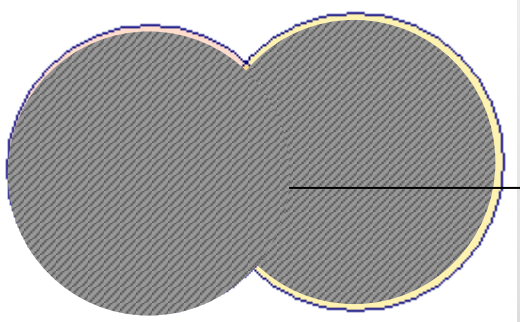
COLUMNS

B D E

Table Y

FULL OUTER JOIN

A B C D E



ALL ROWS FROM TABLE Y AND TABLE Y AND COMMON ROWS ONLY ONCE

OUTPUT

Syntax

```
SELECT column_name, column_name [,column_name]
FROM table1_name [LEFT | RIGHT| FULL] OUTER JOIN
table2_name
ON table1_name.ref_column_name join_operator
table2_name.ref_column_name
```

Left Outer Join

- Returns all rows from the table specified on the left side of the LEFT OUTER JOIN keyword and the matching rows from the table specified on the right side.
- Displays NULL for the columns of the table specified on the right side where it does not find matching records.
- Example

```
SELECT p.ProductID, p1.SalesOrderID, p1.UnitPrice
FROM Sales.SpecialOfferProduct p
LEFT OUTER JOIN [Sales].[SalesOrderDetail] p1
ON p. ProductID = p1.ProductID
WHERE SalesOrderID IS NULL
```

Right outer join

- Returns all the rows from the table specified on the right side of the RIGHT OUTER JOIN keyword.
- Returns the matching rows from the table specified on the left side.
- Example

```
SELECT e.Title, d.JobCandidateID FROM
HumanResources.Employee e
RIGHT OUTER JOIN HumanResources.JobCandidate d ON
e.EmployeeID=d.EmployeeID
```

Full outer join

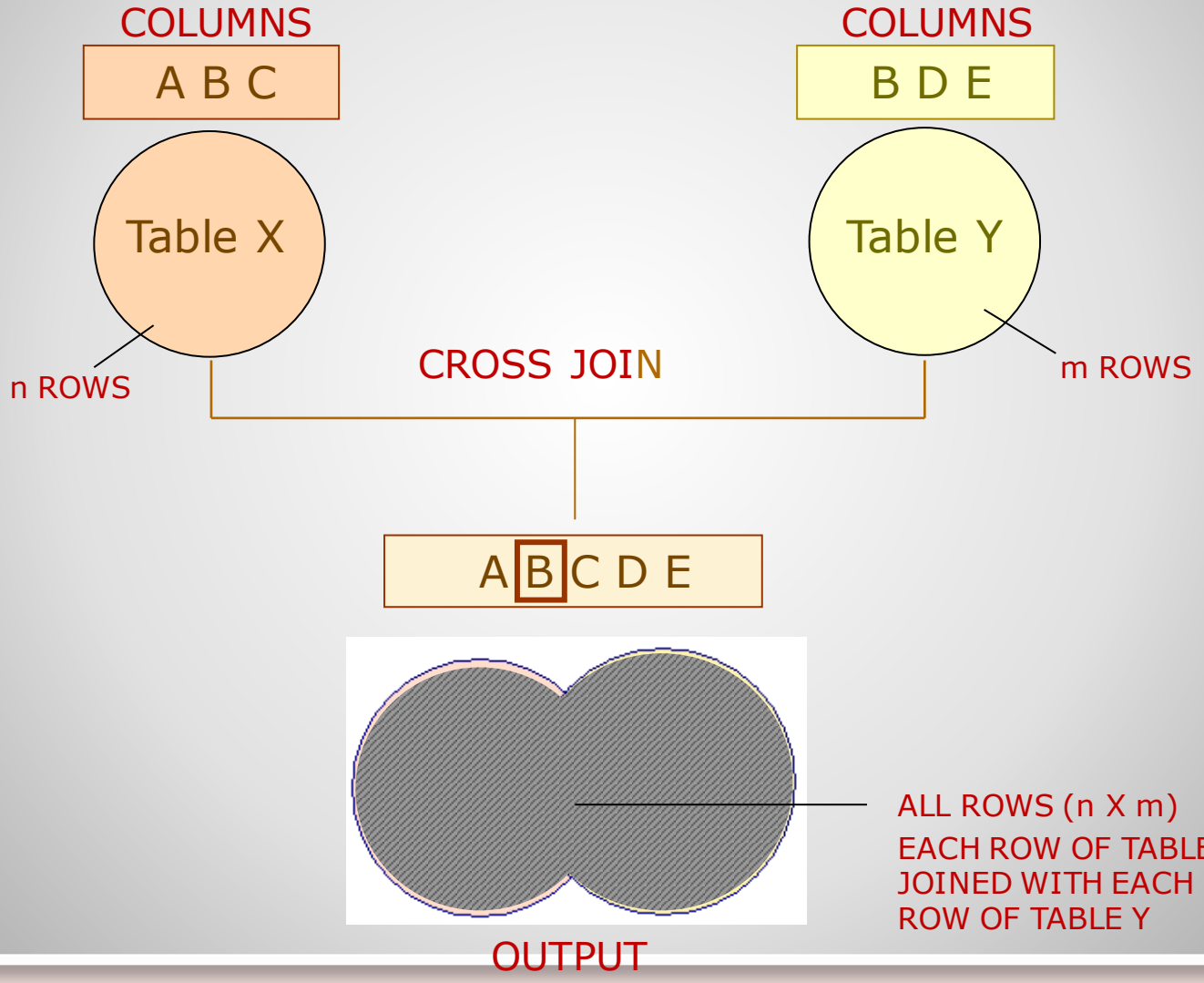
- Is a combination of left outer join and right outer join.
- Returns all the matching and non-matching rows from both the tables.

```
SELECT e.EmployeeID, e.EmployeeName,
ed.EmployeeEducationCode,
ed.Education
FROM Employee e FULL OUTER JOIN Education ed
ON e.EmployeeEducationCode = ed.EmployeeEducationCode
```

Cross join

- Is also known as the Cartesian Product.
- Joins each row from one table with each row of the other table.

Cross Join


COLUMNS
A B C
COLUMNS
B D E
Table X
Table Y
n ROWS
CROSS JOIN
m ROWS
A B C D E
ALL ROWS (n X m)
EACH ROW OF TABLE X
JOINED WITH EACH
ROW OF TABLE Y
OUTPUT

Cross Join Example

```
SELECT A.CompDescription, B.AddOnDescription,
A.Price + B.Price AS 'Total Cost'
FROM ComputerDetails A CROSS JOIN AddOnDetails B
```

Self join

- A table is joined with itself.
- One row in a table correlates with other rows in the same table.
- A table name is used twice by giving two alias names in the query.

Self Join

- To display the employee details along with their manager details, you can use a self join.

EmployeeID	Title	ManagerID
1	Production Technician - WC60	16
2	Marketing Assistant	6
3	Engineering Manager	12
4	Senior Tool Designer	3
5	Tool Designer	263
6	Marketing Manager	109
7	Production Supervisor - WC60	21
8	Production Technician - WC10	185
9	Design Engineer	3
10	Production Technician - WC10	185

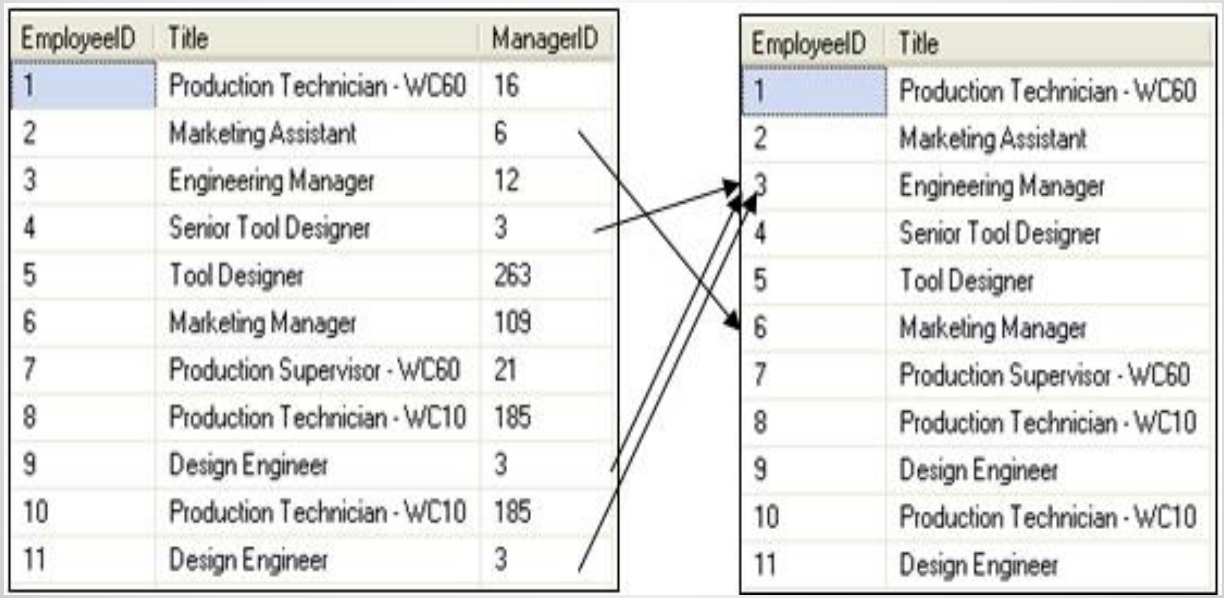
Self Join

Employee (emp)

EmployeeID	Title	ManagerID
1	Production Technician - WC60	16
2	Marketing Assistant	6
3	Engineering Manager	12
4	Senior Tool Designer	3
5	Tool Designer	263
6	Marketing Manager	109
7	Production Supervisor - WC60	21
8	Production Technician - WC10	185
9	Design Engineer	3
10	Production Technician - WC10	185
11	Design Engineer	3

Employee (mgr)

EmployeeID	Title
1	Production Technician - WC60
2	Marketing Assistant
3	Engineering Manager
4	Senior Tool Designer
5	Tool Designer
6	Marketing Manager
7	Production Supervisor - WC60
8	Production Technician - WC10
9	Design Engineer
10	Production Technician - WC10
11	Design Engineer



Emp.ManagerID = mgr.EmployeeID

Self Join Example

```
SELECT emp.EmployeeID, emp.Title AS Employee_Designation,
emp.ManagerID, mgr.Title AS Manager_Designation
FROM HumanResources.Employee emp, HumanResources.Employee mgr
WHERE emp.ManagerID = mgr.EmployeeID
```

SubQueries

- Is an SQL statement that is used within another SQL statement.
- Is nested inside the WHERE or HAVING clause of the SELECT, INSERT, UPDATE, and DELETE statements.

Problem Statement

- Find Names of the Employee whose Salary is more than ‘John’ Salary from the Employee table

```
SELECT EmployeeName from Employee
where Salary>(Select Salary from Employee
where EmpName='John')
```

SubQueries

- You can specify different kinds of conditions on subqueries by using the following keywords:
  - IN:
    - Is used to retrieve rows in a subquery based on the match of values given in a list.
  - EXISTS
    - Is used to check the existence of the data and returns true or false

Example

```
--IN
SELECT EmployeeID
FROM HumanResources.EmployeeAddress
WHERE AddressID IN
(SELECT AddressID FROM Person.Address WHERE City =
'Bothell')

--EXISTS
SELECT EmployeeID, Title
FROM HumanResources.Employee
WHERE EXISTS
(SELECT * FROM HumanResources.EmployeeDepartmentHistory
WHERE EmployeeID = HumanResources.Employee.EmployeeID AND
DepartmentID = 4)
```

Nested subqueries:

- Contain one or more subqueries.
- Can be used when the condition of a query is dependent on the result of another query.

```
SELECT DepartmentID FROM
HumanResources.EmployeeDepartmentHistory
WHERE EmployeeID =
(SELECT EmployeeID FROM HumanResources.Employee
WHERE ContactID =
(SELECT ContactID FROM Person.Contact WHERE
EmailAddress = 'taylor0@adventure-works.com')
)
```

Correlated subqueries

- Can be defined as a query that depends on the outer query for its evaluation.
- Uses the WHERE clause to refer to the table specified in the FROM clause.

```
SELECT * FROM EmployeeDetails e
WHERE Salary = (SELECT max(Salary)
FROM EmployeeDetails WHERE DeptNo = e.DeptNo)
```

SubQueries

- https://docs.microsoft.com/en-us/sql/relational-databases/performance/subqueries

Views

- A ***view*** is simply a virtual table consisting of different columns from one or more tables.
- Unlike a table, a view is stored in the database as a query object; therefore, a view is an object that obtains its data from one or more tables.

Views ensure the security of data by restricting access to the following data:

- Specific rows of tables
- Specific columns of tables
- Specific rows and columns of tables
- Rows obtained by using joins
- Statistical summaries of data in given tables
- Subsets of another view or subsets of views and tables

Two methods to define View

- By using SSMS
- By writing a Transact-SQL statement

using SSMS

- Expand the Views section by clicking the plus sign () next to Views.
- Right-click the Views folder in object explorer,then select New View.
- For details refer
- https://docs.microsoft.com/en-us/sql/ssms/visual-db-tools/query-and-view-designer-tools-visual-database-tools?view=sql-server-ver15

Using Query

```
USE AdventureWorks2012 ;
GO
CREATE VIEW HumanResources.EmployeeHireDate
AS
SELECT p.FirstName, p.LastName, e.HireDate
FROM HumanResources.Employee AS e JOIN Person.Person AS  p

ON e.BusinessEntityID = p.BusinessEntityID ;
GO
-- Query the view
SELECT FirstName, LastName, HireDate
FROM HumanResources.EmployeeHireDate
ORDER BY LastName;
```

Learning more about Views

- https://docs.microsoft.com/en-us/sql/t-sql/statements/create-view-transact-sql?view=sql-server-ver15

BATCH

- Is a group of SQL statements submitted together to SQL Server for execution.
- Compiles the statements as a single executable unit called an execution plan.
- Uses GO command at the end to send the SQL statements to the instance of SQL Server.

Variables

- Batch uses variables to store values during execution.
- A variable must be declared by using the DECLARE statement.
- A variable name is always preceded by the '@' symbol.
- The syntax of declaring a variable is:
- DECLARE @variable_name data_type

```
DECLARE @variable_name data_type
```

- Example

```
DECLARE @Rate int
SELECT @Rate = max(Rate)
FROM HumanResources.EmployeePayHistory
GO
```

Displaying Message

- A batch uses the PRINT statement to display user-defined messages and values of variables.
- For example:

```
PRINT @Rate
```

Programming Construct in SQL

- Programming constructs allow batches to perform the conditional execution of statements.
- A batch uses the following constructs to control the flow of statements:
  - IF...ELSE statement
  - CASE statement
  - WHILE statement

IF-ELSE

- The IF...ELSE statement:
- Is used to perform a particular action based on the result of the boolean expression.

```
IF boolean_expression {sql_statement |
statement_block} [ELSE boolean_expression
{sql_statement | statement_block}]
```

IF-ELSE EXAMPLE

```
DECLARE @Rate money
SELECT @Rate = Rate FROM
HumanResources.EmployeePayHistory
WHERE EmployeeID = 23
IF @Rate < 15
PRINT 'Review of the rate is required‘
ELSE
BEGIN
PRINT 'Review of the rate is not required‘
PRINT 'Rate ='
PRINT @Rate
END
GO
```

CASE STATEMENT

- Is used to evaluate a list of conditions and returns one of the possible results.
- Syntax

```
CASE
WHEN boolean_expression THEN expression
[[WHEN boolean_expression THEN expression]
[...]]
[ELSE expression]
END
```

- Example

```
SELECT EmployeeID, 'Marital Status' =
CASE MaritalStatus
        WHEN 'M' THEN 'Married'
        WHEN 'S' THEN 'Single'
        ELSE 'Not specified'
END
FROM HumanResources.Employee
GO
```

While Statement

- Is used to execute repeatedly as long as the given condition holds true.
- Is used to break or continue the loop by using the BREAK and CONTINUE statements.
- Syntax

```
WHILE  boolean_expression
{sql_statement | statement_block}
[BREAK]
{sql_statement | statement_block}
[CONTINUE]
```

Example

```
WHILE (SELECT AVG(ListPrice) FROM Production.Product) <
$300
BEGIN
   UPDATE Production.Product
      SET ListPrice = ListPrice * 2
   SELECT MAX(ListPrice) FROM Production.Product
   IF (SELECT MAX(ListPrice) FROM Production.Product) >
$500
      BREAK
   ELSE
      CONTINUE
END
PRINT 'Too much for the market to bear';
```

Exception Handling

- The errors that occur at run time are known as exceptions.
- Exceptions can be handled in the following ways:
  - By using the TRY-CATCH construct
  - By using the RAISERROR statement and handling the error in the application

Using TRY-CATCH

- TRY block encloses a group of T-SQL statements. If any error occurs in the statements of TRY block, control passes to the CATCH block.
- CATCH block encloses another group of statements, which gets executed when an error occurs in the TRY block
- Syntax

```
TRY
   <SQL statements>
   …
CATCH
    <SQL statements>
    …
END CATCH
```

Catch Block

- In the CATCH block, you can use the following system functions to determine the information about errors:
  - ERROR_LINE()
  - ERROR_MESSAGE()
  - ERROR_NUMBER()
  - ERROR_PROCEDURE
  - ERROR_SEVERITY():
  - ERROR_STATE()

Example

```
BEGIN TRY
INSERT INTO [AdventureWorks].[Person].[Contact]
VALUES (0, null, 'Robert', 'J', 'Langdon', NULL,'rbl@adventure-
works.com', 0, '1 (11) 500 555-0172' , '9E685955-ACD0-4218-AD7F-
60DDF224C452','2a31OEw=', NULL, newid(), getdate())
INSERT INTO [AdventureWorks].[HumanResources].[Employee]
VALUES ('AS01AS25R2E365W', 19979, 'robertl', 16, 'Tool Designer',
'1972-05-15', 'S', 'M', '1996-07-31', 0, 16, 20, 1, newid(),
getdate())
END TRY
BEGIN CATCH
SELECT 'There was an error! ' + ERROR_MESSAGE() AS ErrorMessage,
    ERROR_LINE()          AS ErrorLine,
    ERROR_NUMBER()        AS ErrorNumber,
    ERROR_PROCEDURE()     AS ErrorProcedure,
    ERROR_SEVERITY()      AS ErrorSeverity,
    ERROR_STATE()         AS ErrorState
END CATCH
GO
```

RAISERROR

- Is used to return messages back to the called applications.
- Uses the same message format as a system error or warning message generated by the SQL Server Database Engine.
- Can also return user-defined error messages.
- Syntax:
  - RAISEERROR('Message', Severity, State)

JSON Data in SQL Server

- JSON is a popular textual data format that's used for exchanging data in modern web and mobile applications.
- JSON is also used for storing unstructured data in log files or NoSQL databases such as Microsoft Azure Cosmos DB.
- Many REST web services return results that are formatted as JSON text or accept data that's formatted as JSON.

Convert JSON collections to a rowset

- OPENJSON
  - you can easily convert JSON data to rows and columns by calling the **OPENJSON** rowset function

```
DECLARE @json NVARCHAR(MAX);
SET @json = N'[
  {"id": 1, "Name":"john"},
  {"id": 2, "Name": "Rita"}
]';

SELECT *
FROM OPENJSON(@json)
  WITH (
    Id INT '$.id',
    Name NVARCHAR(50) '$.Name'
  )
```

Convert SQL Server data to JSON or export JSON

```
SELECT id, firstName AS "info.name", lastName AS
"info.surname", age, dateOfBirth AS dob
FROM People
FOR JSON PATH;
```

JSON Web Reference

- https://docs.microsoft.com/en-us/sql/relational-databases/json/json-data-sql-server?view=sql-server-ver15

Cursors

- Cursor is a database objects to retrieve data from a result set one row at a time, instead of the T-SQL commands that operate on all the rows in the result set at one time.
- We use cursor when we need to update records in a database table in singleton fashion means row by row.

Life Cycle of Cursor

- Declare Cursor
  - A cursor is declared by defining the SQL statement that returns a result set.
- Open
  - A Cursor is opened and populated by executing the SQL statement defined by the cursor.
- Fetch
  - When cursor is opened, rows can be fetched from the cursor one by one or in a block to do data manipulation.
- Close
  - After data manipulation, we should close the cursor explicitly.
- Deallocate
  - Finally, we need to delete the cursor definition and released all the system resources associated with the cursor.

Syntax to Declare Cursor

```
DECLARE cursor_name [ INSENSITIVE ] [ SCROLL ] CURSOR
     FOR select_statement
     [ FOR { READ ONLY | UPDATE [ OF column_name [ ,...n ] ] } ]
[;]
Transact-SQL Extended Syntax
DECLARE cursor_name CURSOR [ LOCAL | GLOBAL ]
     [ FORWARD_ONLY | SCROLL ]
     [ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]
     [ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]
     [ TYPE_WARNING ]
     FOR select_statement
     [ FOR UPDATE [ OF column_name [ ,...n ] ] ]
[;]
```

Cursor Example

- Suppose we have following table

Results | Messages

	EmpID	Name	Salary	Address
1	1	Mohan	12000	Noida
2	2	Pavan	25000	Delhi
3	3	Amit	22000	Dehradun
4	4	Sonu	22000	Noida
5	5	Deepak	28000	Gurgaon

- We want to read each row

Cursor Example

```
SET NOCOUNT ON
DECLARE @Id int,@name varchar(50),@salary int
DECLARE cur_emp CURSOR
STATIC FOR
SELECT EmpID,EmpName,Salary from Employee
OPEN cur_emp
IF @@CURSOR_ROWS > 0
BEGIN
FETCH NEXT FROM cur_emp INTO @Id,@name,@salary
WHILE @@Fetch_status = 0
BEGIN
PRINT 'ID : '+ convert(varchar(20),@Id)+', Name : '+@name+ ', Salary :
'+convert(varchar(20),@salary)
FETCH NEXT FROM cur_emp INTO @Id,@name,@salary
END
END
CLOSE cur_emp
DEALLOCATE cur_emp
SET NOCOUNT OFF
```

Cursor Web Resource

- https://docs.microsoft.com/en-us/sql/t-sql/language-elements/declare-cursor-transact-sql?view=sql-server-ver15
- https://www.mssqltips.com/sqlservertip/1599/cursor-in-sql-server/
- https://medium.com/analytics-vidhya/fetch-cursor-in-sql-server-2684d2c7d36

Stored Procedures

- A ***stored procedure*** is a previously written SQL statement which has been “stored” or saved into the database.
- One of the things that will save you time when running the same query over and over again is to create a stored procedure, which you can then execute from within the database’s command environment.
- Right-click Stored Procedures and choose New Stored Procedure

Creating Store Procedure

```
USE AdventureWorks2012;
GO
CREATE PROCEDURE HumanResources.uspGetEmployeesTest2
    @LastName nvarchar(50),
    @FirstName nvarchar(50)
AS

    SET NOCOUNT ON;
    SELECT FirstName, LastName, Department
    FROM HumanResources.vEmployeeDepartmentHistory
    WHERE FirstName = @FirstName AND LastName = @LastName

    AND EndDate IS NULL;
GO
```

Executing Store Procedure

```
EXECUTE HumanResources.uspGetEmployeesTest2 N'Ackerman', N'Pil
ar';
-- Or
EXEC HumanResources.uspGetEmployeesTest2 @LastName = N'Ackerma
n', @FirstName = N'Pilar';
GO
-- Or
EXECUTE HumanResources.uspGetEmployeesTest2 @FirstName = N'Pil
ar', @LastName = N'Ackerman';
GO
```

Return Values from Store Procedure

- By using the RETURN statement.
- By using the OUTPUT parameter.

Example

- Accepts the employee ID as an input parameter and returns the department name and shift ID as the output parameters

```
CREATE PROCEDURE prcGetEmployeeDetail @EmpId int, @DepName
char(50) OUTPUT, @ShiftId int OUTPUT
AS
BEGIN
IF EXISTS(SELECT * FROM HumanResources.Employee WHERE
EmployeeID = @EmpId)
BEGIN
SELECT @DepName = d.Name, @ShiftId = h.ShiftID
FROM HumanResources.Department d JOIN
HumanResources.EmployeeDepartmentHistory h
ON d.DepartmentID = h.DepartmentID
WHERE EmployeeID = @EmpId AND h.Enddate IS NULL
RETURN 0
END
```

Call a Procedure from Another Procedure

- Calls the prcGetEmployeeDetail procedure

```
CREATE PROCEDURE prcDisplayEmployeeStatus @EmpId int
AS
BEGIN
DECLARE @DepName char(50),@ShiftId int,@ReturnValue int
EXEC @ReturnValue = prcGetEmployeeDetail @EmpId,    @DepName OUTPUT,@ShiftId
OUTPUT
IF (@ReturnValue = 0)
BEGIN
PRINT 'The details of an employee with ID: ' + convert(char(10), @EmpId)
PRINT 'Department Name: ' + @DepName
PRINT 'Shift ID: ' + convert( char(1), @ShiftId)
  SELECT ManagerID, Title FROM
HumanResources.Employee
WHERE EmployeeID = @EmpID
END
ELSE
  PRINT 'No records found for the given employee'
END
```

Alter, rename, recompile, granting permission and Drop Procedure

- https://docs.microsoft.com/en-us/sql/relational-databases/stored-procedures/modify-a-stored-procedure?view=sql-server-ver15

Trigger

- Is a set of T-SQL statements that executes in response to certain actions, such as insert or delete.
- Is used to ensure data integrity.
- Triggers are of the following types
  - DML triggers
  - DDL triggers

DML Trigger

- Fire automatically in response to DML statements.
- Ensure data integrity.
- Execution of a DML trigger creates the following temporary tables, called magic tables:
  - Inserted table:
    - Contains a copy of all the records that are inserted in the trigger table.
  - Deleted table:
    - Contains a copy of all the records that have been deleted from the trigger table.

DML Trigger

- Depending on the operation that is performed, DML triggers can be further categorized as:
- Insert trigger:
  - Gets fired when a new record is added.
- Update trigger:
  - Gets fired when an existing record is modified.
- Delete trigger:
  - Gets fired when a record is deleted

DML Trigger

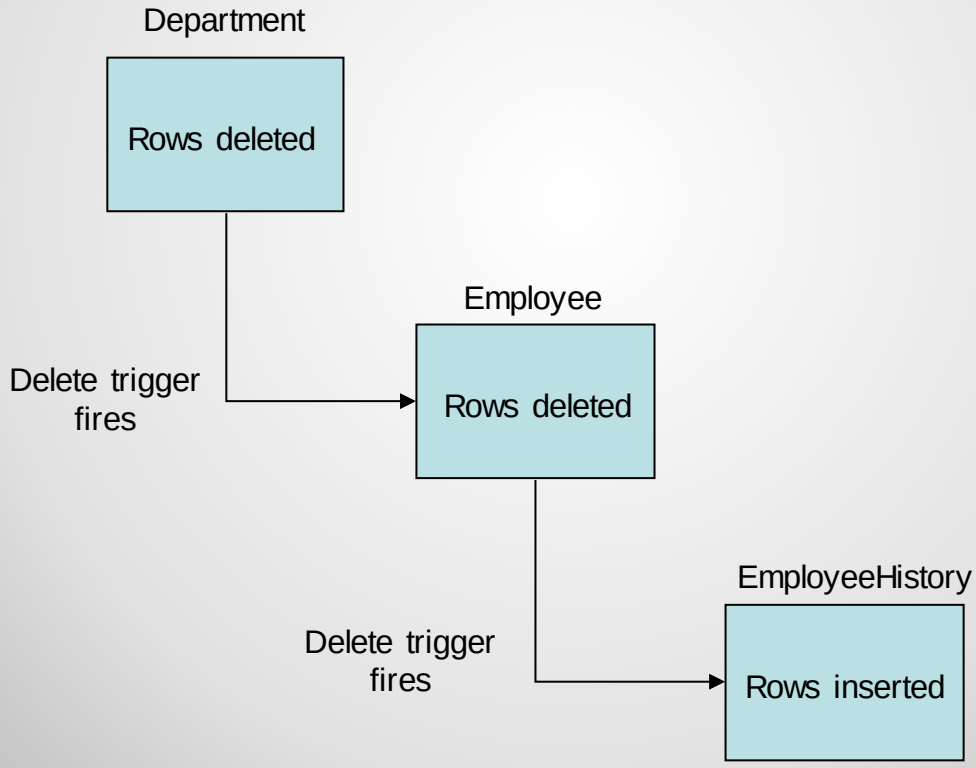
- Depending on the way the triggers are fired, DML triggers can be further categorized as:
- After trigger:
  - Fires after the execution of the DML operation for which the trigger is defined.
- Instead of trigger:
  - Executes instead of the events that cause the trigger to fire.
  - Can be created on both, a table as well as a view.

DDL trigger

- Is fired in response to DDL statements.
- Is used to perform administrative tasks.

Nested Trigger

- Triggers can be nested, as shown in the following figure.

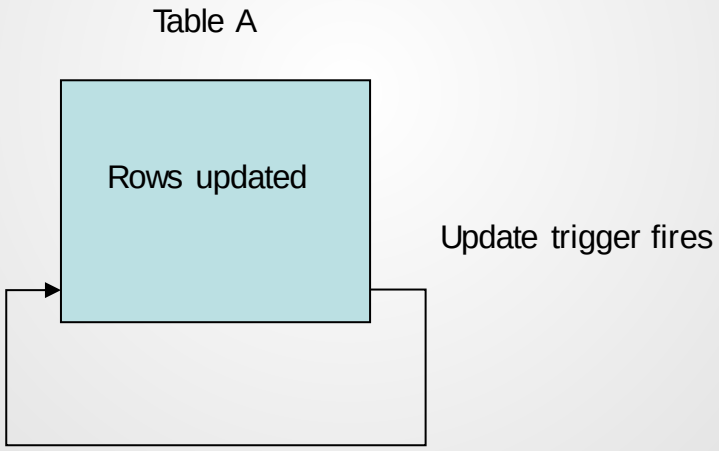


Recursive triggers:

- Eventually call itself.
- Are special cases of nested triggers.
- Can be of the following types:
  - Direct
  - Indirect

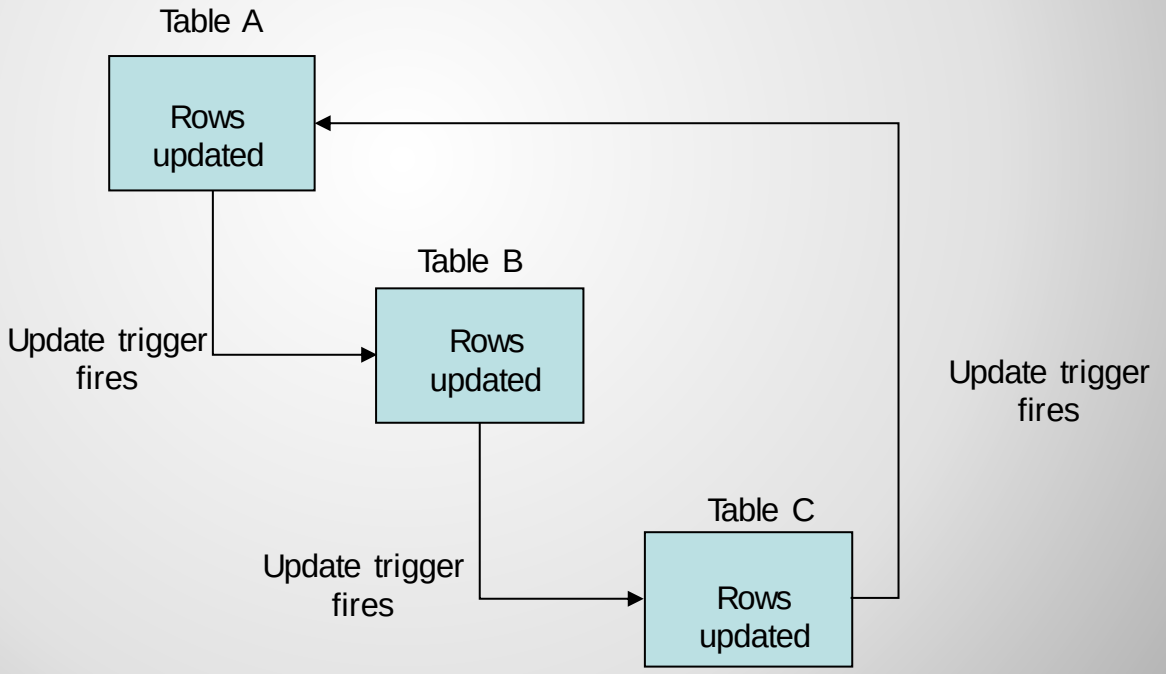
Direct recursive trigger:

- Fires and performs an action that causes the same trigger to fire again.



Indirect recursive trigger

- Fires a trigger on another table and eventually the nested trigger ends up firing the first trigger again.



Creation of Trigger

- Triggers are created using the CREATE TRIGGER statement.
  - Syntax

```
CREATE TRIGGER trigger_name
ON { OBJECT NAME }
{ FOR | AFTER | INSTEAD OF } { event_type [
,...n ] |
DDL_DATABASE_LEVEL_EVENTS }
{ AS
{ sql_statement [ ...n ] }
}
```

CREATION OF TRIGGER

- Gets fired at the time of inserting records in the Shift table
- Rolls back the INSERT statement if the modified date is not equal to the current date

```
CREATE TRIGGER trgInsertShift
ON HumanResources.Shift
FOR INSERT
AS
  DECLARE @ModifiedDate datetime
  SELECT @ModifiedDate = ModifiedDate FROM Inserted
  IF (@ModifiedDate != getdate())
  BEGIN
    PRINT 'The modified date should be the current     date.
Hence, cannot insert.'
    ROLLBACK TRANSACTION

  END
```

Creating a delete trigger:

- Creates a DELETE trigger on the Department table
- Prevents deletion in the Department table

```
CREATE TRIGGER trgDeleteDepartment
ON HumanResources.Department
FOR DELETE
AS
PRINT 'Deletion of Department is not   allowed'
ROLLBACK TRANSACTION
RETURN
```

Creating an update trigger:

- Gets fired while updating the Rate column in the EmployeePayHistory table

```
CREATE TRIGGER trgUpdateEmployeePayHistory
ON HumanResources.EmployeePayHistory
FOR UPDATE
AS
IF UPDATE (Rate)
BEGIN
DECLARE @AvgRate float
SELECT @AvgRate = AVG(Rate)
FROM HumanResources.EmployeePayHistory
IF(@AvgRate > 20)
BEGIN
PRINT 'The average value of rate cannot be more than 20'
IF(@AvgRate > 20)
BEGIN
PRINT 'The average value of rate cannot be      more than 20'
ROLLBACK TRANSACTION
END
END
```

Creating an after trigger

- Gets fired after the DELETE statement is executed on the Shift table to display a confirmation message

```
CREATE TRIGGER trgDeleteShift ON    HumanResources.Shift
AFTER DELETE
AS
PRINT 'Deletion successful'
```

Creating an instead of trigger

- Displays a message instead of deleting record from the Project table

```
CREATE TRIGGER trgDelete ON
HumanResources.Project
INSTEAD OF DELETE
AS
PRINT 'Project records cannot be deleted'
```

Creating a DDL trigger:

```
CREATE TRIGGER safety
ON DATABASE
FOR DROP_TABLE, ALTER_TABLE
AS
PRINT 'You must disable Trigger "safety" to drop  or
alter tables!'
ROLLBACK
```

Altering a trigger

- Syntax:

```
ALTER TRIGGER trigger_name
{ FOR | AFTER } { event_type [ ,...n ] |
DDL_DATABASE_LEVEL_EVENTS }
{ AS
{ sql_statement [ ...n ] }
}
```

Alter Trigger

```
ALTER TRIGGER HumanResources.trgInsertShift
ON HumanResources.Shift
FOR INSERT
AS
DECLARE @ModifiedDate datetime
SELECT @ModifiedDate = ModifiedDate FROM       Inserted
IF (@ModifiedDate != getdate())
BEGIN
RAISERROR ('The modified date is not the      current
date. The transaction cannot be     processed.',10, 1)
ROLLBACK TRANSACTION
END
RETURN
```

Deleting a trigger

- Syntax

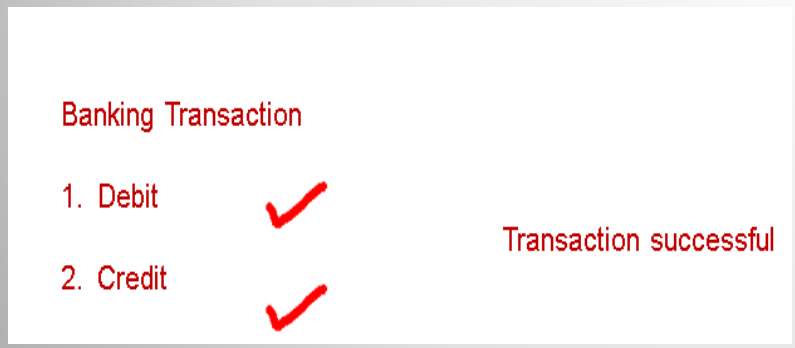
```
DROP TRIGGER { trigger }
```

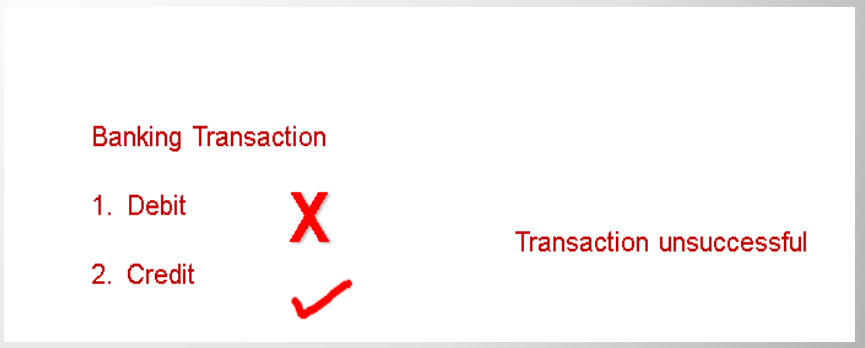
- Example

```
DROP TRIGGER
HumanResources.trgMagic
```

Transaction

- A transaction can be defined as a sequence of operations performed together as a single logical unit of work.
- Let us discuss about a banking transaction where the amount is debited from one account, and credited to another account





ACID Properties

- Atomicity
  - Either all the data modifications are performed or none of them are performed.
- Consistency
  - All the data is in a consistent state after a transaction is completed successfully.
- Isolation
  - Any data modification made by one transaction must be isolated from the other concurrent transactions.
- Durability
  - Any change in data by a completed transaction remains permanent in the system.

A transaction can be implemented in the following ways:

- Autocommit transaction
  - It is the default transaction management mode of SQL Server.
  - Each individual statement is a transaction.
- Implicit transaction
- A new transaction is implicitly started when the prior transaction completes, but each transaction is explicitly completed with a COMMIT or ROLLBACK statement.
- Explicit transaction
  - Each transaction is explicitly started with the BEGIN TRANSACTION statement and explicitly ended with a COMMIT or ROLLBACK statement.
- Batch-scoped transactions
  - Applicable only to multiple active result sets (MARS), a Transact-SQL explicit or implicit transaction that starts under a MARS session becomes a batch-scoped transaction. A batch-scoped transaction that is not committed or rolled back when a batch completes is automatically rolled back by SQL Server.

Implicit transaction:

- It does not define the start of the transaction.
- You are only required to commit or roll back the transaction.
- You need to turn on the implicit transaction mode to specify the implicit transaction.
- For example:

```
SET IMPLICIT_TRANSACTIONS ON;
INSERT INTO Emp VALUES ('Jack',
'Marketing');
INSERT INTO Emp VALUES ('Robert',
'Finance');
COMMIT TRANSACTION;
```

- You can turn off the Implicit transaction mode by using the following statement:

```
SET IMPLICIT_TRANSACTIONS OFF
```

Explicit transaction

- Can be created by using the following statements:
  - BEGIN TRANSACTION
    - Is used to set the starting point of a transaction
  - COMMIT TRANSACTION
    - Is used to save the changes permanently in the database
  - ROLLBACK TRANSACTION
    - Is used to undo the changes
  - SAVE TRANSACTION
    - Is used to establish save points that allow partial rollback of a transaction

Explicit Transaction

- You can define the beginning and end of a transaction, as shown in the following statements:

```
BEGIN TRAN myTran
UPDATE FixedDepositAccount
SET Balance = Balance - 25000
WHERE AccountName = 'John'

UPDATE SavingsAccount
SET Balance = Balance + 25000
WHERE AccountName ='John'
COMMIT TRAN myTran
```

Rollback Transaction

- Transactions are reverted:
- When the execution of transaction is in an invalid state.
- To maintain consistency.
- Using the ROLLBACK TRANSACTION and ROLLBACK WORK statements.

```
ROLLBACK [TRAN[SACTION] [transaction_name
|@tran_name_variable
|savepoint_name |
@savepoint_variable]]
```

Transaction Example

```
BEGIN TRANSACTION TR1
BEGIN TRY
UPDATE Person.Contact
SET EmailAddress='jolyn@yahoo.com'
WHERE ContactID = 1070
--Statement 1
UPDATE HumanResources.EmployeeAddress SET AddressID = 32533
WHERE EmployeeID = 1
COMMIT TRANSACTION TR1
--Statement 2
SELECT 'Transaction Executed'
END TRY
BEGIN CATCH
  ROLLBACK TRANSACTION TR1
  SELECT 'Transaction Rollbacked'
END CATCH
```

Locking

- SQL Server uses the concept of locking to ensure transactional integrity.
  - In the absence of locking, the following problems may occur:
    - Lost updates
      - Occurs when two or more transactions try to modify the same row
    - Uncommitted dependency (Dirty read)
      - Occurs when a transaction queries data from a table when the other transaction is in the process of modifying data
    - Inconsistent analysis
      - Occurs when the data is changed between simultaneous read by one user
    - Phantom reads
      - Occurs when new records inserted by a user are identified by transactions that started prior to the INSERT statement
      -

Isolation Levels

- Locking is controlled by the following types of isolation levels:
  - READ UNCOMMITTED
    - Allows another transaction to read uncommitted data.
  - READ COMMITED
    - Allows another transaction to read committed data.
  - REPEATABLE READ
    - A transaction cannot read the data that is being modified by the current transaction.
    - No other transaction can update the data read by the current transaction until the current transaction completes.
    - Other transactions can insert new rows.
  - SNAPSHOT
    - Other transactions can insert new rows.
  - SERIALIZABLE
    - No transaction can read, modify, or insert new data while the data is being read or updated by the current transaction.

Implement Isolation Level

```
SET TRANSACTION ISOLATION LEVEL
READ COMMITTED
BEGIN TRANSACTION TR
BEGIN TRY
UPDATE Person.Contact
SET EmailAddress='jolyn@yahoo.com'
WHERE ContactID = 1070
UPDATE HumanResources.EmployeeAddress SET AddressID = 32533
WHERE EmployeeID = 1
COMMIT TRANSACTION TR
PRINT 'Transaction Executed'
END TRY
BEGIN CATCH
ROLLBACK TRANSACTION TR
PRINT 'Transaction Rollbacked'
END CATCH
```

What is a deadlock?

- A deadlock is a situation where each transaction waits for a lock on each other's objects to be released.

Transaction Web Resource

- https://docs.microsoft.com/en-us/sql/t-sql/language-elements/transaction-isolation-levels?view=sql-server-ver15
- https://docs.microsoft.com/en-us/sql/t-sql/language-elements/transactions-transact-sql?view=sql-server-ver15
- https://docs.microsoft.com/en-us/sql/t-sql/language-elements/transactions-transact-sql?view=sql-server-ver15

Scenario 1-2: Designing a Relational Database

- You have been hired to create a relational database to support a car sales business. You need to store information on the business's employees, inventory, and completed sales. You also need to account for the fact that each salesperson receives a different percentage of their sales in commission. What tables and columns would you create in your relational database, and how would you link the tables?

Scenerio-1 Looking a Relational Database

- You just got hired as a DBA for an international company that is a holding company for many other companies. Your first task is to design a new database infrastructure. Therefore, you are told to think of your activities over the last several weeks. List at least one database that you have used directly or directly and describe how the database is most likely laid out.

Scenario 1-3: Using Help from SQL Server

- You recently graduated from school and were hired as a junior database administrator. One thing you've learned over your first few months on the job is that you don't have all the answers. Thankfully, Microsoft SQL Server 2008 has an extensive help system and examples.Say you want to display help regarding use of the CREATE statement so that you can create a table. What steps would you use to find that information

Creating Databases Using the SSMS Graphical Interface

- Your company, AdventureWorks, has decided to expand into interstellar travel. They have asked you to create a new database called Planets on the Microsoft SQL server using the SSMS graphical interface. What steps would you complete to create this database?