Contents

List of Figures

1 SQL (Structured Query Language)

- Is a language that allows you to communicate with the database
  - Write questions that a database can understand
  - Often in a series of smaller questions

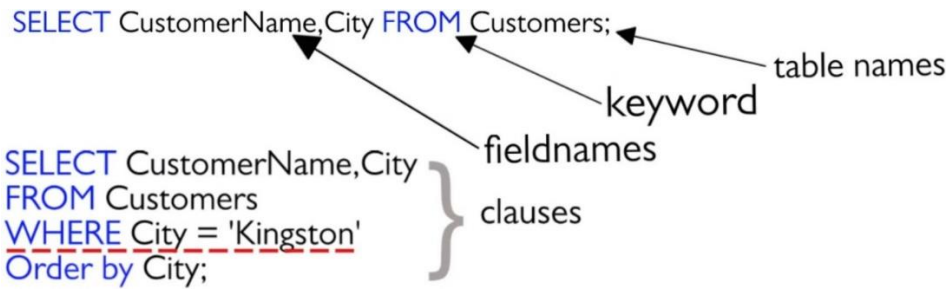


Figure 1: SQL statement

- A SQL statement is any SQL code that performs an action
- A SQL query is any statement that returns records using the SELECT keyword
- Types of SQL statements
  - DML (*Data Manipulation Language*)
    - Edit data in the database
    - Create, Read, Update or Delete (CRUD) records
  - DDL (Data Definition Language)
    - Edit the structure of the database
    - Add, change or remove database objects

2 DDL (Data Definition Language)

2.1 Create Database

- Best practices:
  - CamelCase the database name or use an underscore

```
-- Create the new database if it does not exist already
IF NOT EXISTS (
    SELECT [name]
        FROM sys.databases
        WHERE [name] = N'DatabaseName'
)
CREATE DATABASE DatabaseName
```

Figure 2: Create database

2.2 Drop Database

```
-- Drop the database if it exists
IF EXISTS (
    SELECT [name]
        FROM sys.databases
        WHERE [name] = N'DatabaseName'
)
DROP DATABASE DatabaseName
```

Figure 3: Drop database

2.3 Create/Drop Table

```
-- Drop the table if it already exists
USE DatabaseName
GO
-- Create the table in the specified schema
CREATE TABLE Employee
(
    Id INT PRIMARY KEY IDENTITY, -- Primary Key column
    FirstName VARCHAR(50) NOT NULL,
    LastName NVARCHAR(50) NOT NULL,
    DateOfBirth DATETIME2
);
GO
DROP TABLE Employee
GO
```

Figure 4: Create and drop table

2.4 Alter Table

```
ALTER TABLE Employee
ADD JobTitleId INT NOT NULL, FOREIGN KEY (JobTitleId) REFERENCES JobTitle(Id);
GO
ALTER TABLE Employee
ALTER COLUMN DateOfBirth DATE;
GO
ALTER TABLE Employee
DROP COLUMN DateOfBirth;
GO
ALTER TABLE Employee
ADD CONSTRAINT UQ_EmployeeId UNIQUE(EmployeeId);
GO
```

Figure 5: Alter table

```
ALTER TABLE Employee
ADD EmployeeId VARCHAR(10) UNIQUE, EmployementDate DATETIME2 DEFAULT GETDATE();
```

Figure 6: Constraints

3 DML (Data Manipulation Language)

3.1 Read Data

3.1.1 SELECT Statement

- The SELECT statement is used to select data from a database

```
-- Select all data from a table
SELECT * FROM Person.Person
```

Figure 7: SELECT statement

3.1.2 WHERE Clause

```
-- Select only persons who are Marketing Assistants
SELECT * FROM HumanResources.Employee
WHERE JobTitle = 'Marketing Assistant'

-- Select only the employee with the id 20
SELECT * FROM HumanResources.Employee
WHERE BusinessEntityID = 20
```

Figure 8: WHERE clause

3.1.3 SELECT TOP Clause

```
-- Select top 10 records from the table
SELECT TOP(10) * FROM Person.Person
```

Figure 9: SELECT TOP clause

3.1.4 ORDER BY Keyword

```
-- Select all employees sorted by hire date
SELECT * FROM HumanResources.Employee
ORDER BY HireDate DESC
```

Figure 10: ORDER BY keyword

3.1.5 LIKE Operator

```
--Select all records with the word "Marketing" in the job title
SELECT * FROM HumanResources.Employee
WHERE JobTitle LIKE '%Marketing%'    -- CONTAINS

SELECT * FROM HumanResources.Employee
WHERE JobTitle LIKE '%Marketing'     -- ENDS WITH

SELECT * FROM HumanResources.Employee
WHERE JobTitle LIKE 'Marketing%'     -- STARTS WITH
```

Figure 11: LIKE operator

3.1.6 ALIASES

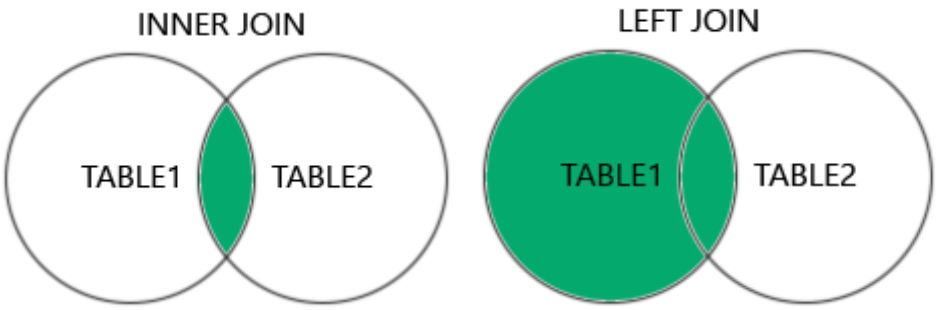
```sql
-- Select data with alised columns
SELECT
    BusinessEntityID AS Id,
    FirstName [First Name],
    LastName [Last Name]
FROM Person.Person
```

Figure 12: Aliases

3.1.7 INNER/LEFT JOIN Keyword

The INNER JOIN keyword selects records that have matching values in both tables. It is like an intersection.

The LEFT JOIN keyword returns all records from the left table and the matching records from the right table. It is like a complement.




```sql
-- Select work orders, the product and their scrap reason
SELECT
    p.Name [Product Name],
    p.ProductNumber [Product Number],
    wo.OrderQty [Order Qty],
    wo.StockedQty [Stocked Qty],
    sr.Name [Scrap Reason]
FROM [Production].[Product] p
INNER JOIN [Production].[WorkOrder] wo ON wo.ProductID = p.ProductID
LEFT JOIN [Production].[ScrapReason] sr ON sr.ScrapReasonID = wo.ScrapReasonID
```

Figure 13: INNER/LEFT JOIN keyword

3.1.8 UNION Operator

The UNION operator is used to combine the result-set of two or more SELECT statements.

- Every SELECT statement within UNION must have the same number of columns
- The columns must also have similar data types
- The columns in every SELECT statement must also be in the same order

The UNION operator selects only distinct values. To allow duplicate values, use UNION ALL.

```sql
-- Select all customers and employees into one data set
SELECT FirstName
FROM [HumanResources].[Employee] e
INNER JOIN [Person].[Person] p ON p.BusinessEntityID = e.BusinessEntityID
UNION
SELECT FirstName
FROM [Sales].[Customer] c
INNER JOIN [Person].[Person] p ON p.BusinessEntityID = c.PersonID
```

Figure 14: UNION Operator

3.1.9 DISTINCT/GROUP BY Statement

```sql
-- Eliminate all duplicated names from the Customer table
SELECT DISTINCT FirstName, LastName
FROM [Sales].[Customer] c
INNER JOIN [Person].[Person] p ON p.BusinessEntityID = c.PersonID

SELECT FirstName, LastName
FROM [Sales].[Customer] c
INNER JOIN [Person].[Person] p ON p.BusinessEntityID = c.PersonID
GROUP BY FirstName, LastName
```

Figure 15: DISTINCT/GROUP BY statement

3.1.10 Aggregate Functions

```sql
-- Find the number of customers with the same first name
SELECT FirstName, COUNT(FirstName)
FROM [Sales].[Customer] c
INNER JOIN [Person].[Person] p ON p.BusinessEntityID = c.PersonID
GROUP BY FirstName
HAVING COUNT(FirstName) > 1
ORDER BY FirstName

-- Find total, average, lowest and highest amounts for sales
SELECT
    SUM(TotalDue) [Total Sales Amount],
    AVG(TotalDue) [Average Sales Amount],
    MIN(TotalDue) [Lowest Sales Amount],
    MAX(TotalDue) [Highest Sales Amount]
FROM [Sales].[SalesOrderHeader]
```

Figure 16: Aggregate functions

3.1.11 String Manipulation

```sql
-- Find total, average, lowest and highest amounts for sales per salesperson
SELECT
    FirstName + ' ' + LastName [Full Name],
    COUNT(TotalDue) [Number of Sales],
    FORMAT(SUM(TotalDue), 'C') [Total Sales Amount],
    FORMAT(AVG(TotalDue), 'C') [Average Sales Amount],
    FORMAT(MIN(TotalDue), 'C') [Lowest Sales Amount],
    FORMAT(MAX(TotalDue), 'C') [Highest Sales Amount]
FROM [Sales].[SalesPerson] sp
INNER JOIN [Person].[Person] p ON p.BusinessEntityID = sp.BusinessEntityID
INNER JOIN [Sales].[SalesOrderHeader] soh ON soh.SalesPersonID = p.BusinessEntityID
GROUP BY FirstName, LastName
ORDER BY FirstName, LastName
```

Figure 17: String manipulation

Function	Description
ASCII	Returns the ASCII value for the specific character
CHAR	Returns the character based on the ASCII code
CHARINDEX	Returns the position of a substring in a string
CONCAT	Adds two or more strings together
Concat with +	Adds two or more strings together
CONCAT_WS	Adds two or more strings together with a separator
DATALENGTH	Returns the number of bytes used to represent an expression
DIFFERENCE	Compares two SOUNDEX values, and returns an integer value
FORMAT	Formats a value with the specified format
LEFT	Extracts a number of characters from a string (starting from left)
LEN	Returns the length of a string
LOWER	Converts a string to lower-case
LTRIM	Removes leading spaces from a string
NCHAR	Returns the Unicode character based on the number code
PATINDEX	Returns the position of a pattern in a string
QUOTENAME	Returns a Unicode string with delimiters added to make the string a valid SQL Server delimited identifier
REPLACE	Replaces all occurrences of a substring within a string, with a new substring
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)

Figure 18: Sting manipulation functions

3.1.12 Subqueries

```sql
-- Employees who have more vacation available than the average
SELECT BusinessEntityID, JobTitle, VacationHours
FROM [HumanResources].[Employee]
WHERE VacationHours > (SELECT AVG(VacationHours) FROM [HumanResources].[Employee])
```

Figure 19: Subqueries

3.1.13 CTE (Common Table Expressions)

CTE is like a temporary table and his scope is limited to the query that immediately follows it. Once the query is complete, the CTE is no longer accessible.

```sql
-- Find the total number of sales orders per year of each sales person
WITH Sales_CTE
AS
(
    SELECT SalesPersonID, SalesOrderID, YEAR(OrderDate) SalesYear
    FROM [Sales].[SalesOrderHeader]
    WHERE SalesPersonID IS NOT NULL
)
SELECT SalesPersonID, COUNT(SalesOrderID) TotalSales, SalesYear
FROM Sales_CTE
GROUP BY SalesPersonID, SalesYear
ORDER BY SalesPersonID, SalesYear
```

Figure 20: CTE

3.1.14 Window Functions

Window functions operate on a set of rows and return a single aggregated value for each row. The term Window describes the set of rows in the database on which the function will operate.

The Window (set of rows on which functions operates) is defined using an OVER() clause.

```
window_function ( [ ALL ] expression )
OVER ( [ PARTITION BY partition_list ] [
ORDER BY order_list] )
```

Figure 21: Window function syntax

Types of window functions:

- Aggregate window functions
  - SUM(), MAX(), COUNT()
- Ranking window functions
  - RANK(), DENSE_RANK(), ROW_NUMBER()
    - If there are three rows and there are two rows with the same value, the ranking will jump and appear 1-1-3
    - With the DENSE_RANK will appear 1-1-2
- Value window functions
  - LAG(), LEAD(), FIRST_VALUE()

```sql
-- Find the maximum sale price
SELECT
    SalesOrderID,
    OrderQty,
    UnitPrice,
    MAX(UnitPrice) OVER() MaxPrice
FROM [Sales].[SalesOrderDetail]

-- Find the maximum sale price per sales order
SELECT
    SalesOrderID,
    OrderQty,
    UnitPrice,
    MAX(UnitPrice) OVER(PARTITION BY SalesOrderID) MaxPrice
FROM [Sales].[SalesOrderDetail]

-- Find the sum sale price per sales order
SELECT
    SalesOrderID,
    OrderQty,
    UnitPrice,
    SUM(UnitPrice) OVER(PARTITION BY SalesOrderID) Total
FROM [Sales].[SalesOrderDetail]
```

Figure 22: Aggregate window functions

```sql
-- Assign a number two each row of the result
-- Rank employee by vacation hours in the department
SELECT
    BusinessEntityID,
    JobTitle,
    VacationHours,
    ROW_NUMBER() OVER(ORDER BY JobTitle) RowNumber,
    RANK() OVER(PARTITION BY JobTitle ORDER BY VacationHours) Rank,
    DENSE_RANK() OVER(PARTITION BY JobTitle ORDER BY VacationHours) DenseRank
FROM [HumanResources].[Employee]
```

Figure 23: Ranking window functions

```sql
-- Find the next/previous stock level that will be coming up
SELECT
    ProductID,
    [Name],
    ProductNumber,
    SafetyStockLevel,
    LEAD(SafetyStockLevel, 1, 0) OVER(ORDER BY ProductID) NextStockLevel,
    LAG(SafetyStockLevel, 1, 0) OVER(ORDER BY ProductID) PrevStockLevel
FROM Production.Product
```

Figure 24: Value window functions

3.1.15 ISNULL/COALESCE

In ISNULL, the type doesn't matter, but the size does (it allows you to put an integer in a varchar, but if the size is greater than allowed, the value is truncated).

In COALESCE the type is important, but the size is not.

```sql
-- Replace null values
SELECT TOP (10)
    ISNULL([Title], '') Title,
    [FirstName],
    COALESCE([MiddleName], 'N/A') MiddleName,
    [LastName]
FROM [Person].[Person]
```

Figure 25: ISNULL / COALESCE

3.2 Insert Data

3.2.1 Full Insert

```sql
-- Insert several crypto currency symbols
INSERT INTO [Sales].[Currency]
VALUES
(
    'ETH', 'Ethereum', GETDATE()
),
(
    'BCH', 'Bitcoin Cash', GETDATE()
),
(
    'BNB', 'BNB', GETDATE()
);
```

Figure 26: Full INSERT

3.2.2 Partial Insert

```sql
-- Insert new currency - Litecoin
INSERT INTO [Sales].[Currency]
(
    [CurrencyCode],
    [Name]
)
VALUES
(
    'LTC',
    'Litecoin'
);
```

Figure 27: Partial INSERT

3.2.3 SELECT INTO

Useful for copying tables and for backup purposes.

```sql
-- Create a backup copy of PurchaseOrderDetail table
SELECT * INTO [Purchasing].[PurchaseOrderDetailBackup]
FROM [Purchasing].[PurchaseOrderDetail];
```

Figure 28: SELECT INTO

3.3 Update Data

```sql
-- Update the Title and MiddleName of a person
UPDATE [Person].[Person]
SET
    [Title] = 'Mr.', MiddleName = 'Jackson'
WHERE BusinessEntityID = 10;
GO
```

Figure 29: Update data

```sql
-- Set the EmailPromotion = 2 to everyone of Bothell
UPDATE [Person].[Person] SET EmailPromotion = 2
FROM [Person].[Person] p
INNER JOIN [Person].[BusinessEntityAddress] bea ON bea.BusinessEntityID = p.BusinessEntityID
INNER JOIN [Person].[Address] a ON a.AddressID = bea.AddressID
WHERE City = 'Bothell';
GO

-- Update sales person from US region with a bonus of 50%
WITH CTE
AS
(
    SELECT BusinessEntityID, Bonus, CountryRegionCode
    FROM [Sales].[SalesPerson] sp
    INNER JOIN [Sales].[SalesTerritory] st ON st.TerritoryID = sp.TerritoryID
    WHERE CountryRegionCode = 'US'
)
UPDATE CTE SET Bonus += Bonus * 0.5;
```

Figure 30: Update data using JOIN and CTE

3.4 Delete Data

TRUNCATE is used when all the rows in a table have to be deleted. You can also use DELETE, but truncate is much faster.

```sql
-- Delete all rows when a person's middle name is 'J'
DELETE FROM Person.PersonDEMO
WHERE MiddleName = 'J';

-- Delete all rows
TRUNCATE TABLE Person.PersonDEMO;

-- Delete the first 10 rows of a table
WITH CTE
AS
(
    SELECT TOP(10) * FROM Person.PersonDEMO
)
DELETE FROM CTE;
```

Figure 31: Delete data

To delete rows when there are relationships between tables, we need to configure foreign key for that.

```sql
ALTER TABLE [Tab1]
ADD CONSTRAINT FK_Tab1_Tab2
FOREIGN KEY (Tab2Id) REFERENCES Tab2(Id) ON DELETE CASCADE;
```

Figure 32: CASCADE on delete

4 Views

A view is an object that represents a select query that you don't want to write repeatedly.

```sql
CREATE OR ALTER VIEW Sales.vSalesPersonsTotal
AS
SELECT
    FirstName [First Name],
    LastName [Last Name],
    COUNT(TotalDue) [Number of Sales],
    FORMAT(SUM(TotalDue), 'C') [Total Sales Amount],
    FORMAT(AVG(TotalDue), 'C') [Average Sales Amount],
    FORMAT(MIN(TotalDue), 'C') [Lowest Sales Amount],
    FORMAT(MAX(TotalDue), 'C') [Highest Sales Amount]
FROM [Sales].[SalesPerson] sp
INNER JOIN [Person].[Person] p ON p.BusinessEntityID = sp.BusinessEntityID
INNER JOIN [Sales].[SalesOrderHeader] soh ON soh.SalesPersonID = p.BusinessEntityID
GROUP BY FirstName, LastName;
GO
DROP VIEW Sales.vSalesPersonsTotal
```

Figure 33: Manage a view

5 Variables

A variable is a temporary object that is used to store data during the batch execution period.

```sql
DECLARE @tempMessage VARCHAR(100) = 'Test';
-- or
SET @tempMessage = 'Test';
-- or
SELECT @tempMessage = 'Test';
PRINT @tempMessage;
```

Figure 34: Declare variables

6 Functions

6.1 Scalar Functions

A scalar function returns only one value.

```sql
-- Create a function that returns the average sales for an given year
CREATE FUNCTION ufnGetAverageSalesForYear
(
    -- Parameters
    @Year INT
)
RETURNS DECIMAL(10,2)
AS
BEGIN
    -- Declare the return variable
    DECLARE @AverageSalesAmount DECIMAL(10,2)

    -- T-SQL statement
    SELECT @AverageSalesAmount = AVG(TotalDue)
    FROM Sales.SalesOrderHeader
    WHERE YEAR(OrderDate) = @Year

    -- Return the result
    RETURN @AverageSalesAmount
END
```

Figure 35: Create a scalar function

```sql
DECLARE @Year INT = 2012;
SELECT dbo.ufnGetAverageSalesForYear(@Year) [Average Sales], TotalDue
FROM Sales.SalesOrderHeader
WHERE YEAR(OrderDate) = @Year;
```

Figure 36: Test the scalar function

6.2 Table Functions

A table function returns a tabular structure.

This type of function creates a wrapper that allows you to manipulate data in a way that probably wouldn't be possible with a normal view.

```sql
-- Create a function that returns the products with inventory greather than a given value
CREATE FUNCTION ufnGetProductsWithInventoryGreatherThan(@count INT)
RETURNS TABLE
AS
RETURN
    SELECT
        p.Name,
        pi.Quantity
    FROM Production.Product p
    INNER JOIN Production.ProductInventory pi ON pi.ProductID = p.ProductID
    WHERE pi.Quantity >= @count
```

Figure 37: Create a table function

```sql
-- For testing purposes
SELECT * FROM dbo.ufnGetProductsWithInventoryGreatherThan(800);
```

Figure 38: Test the table function

7 Control-of-Flow Statements

7.1 BEGIN… END

The begin keyword in SQL is used to mark the starting point of a logical block of statements, and the end keyword is used to specify the closing point of the logical block in SQL.

7.2 IF/ELSE

```
-- Evaluate a number
BEGIN
    DECLARE @ID INT = 10;

    IF @ID >= 10
    BEGIN
        PRINT('The number is greather than 10');

        IF @ID = 10
            PRINT('The number is 10');
    END
    ELSE
        PRINT('The number is not greather than 10');
END
```

Figure 39: IF/ELSE statements

7.3 WHILE

```
-- Example using WHILE
BEGIN
    DECLARE @num INT = 1;
    DECLARE @total INT = 0;

    WHILE @num <=10
    BEGIN
        SET @total += @num;
        SET @num += 1;
        PRINT @total;
    END
    PRINT @total;
END
```

Figure 40: WHILE statement

7.4 BREAK

```
-- Example of using BREAK to stop the cycle
DECLARE @number INT = 1;
DECLARE @result INT = 0;

WHILE @num <=10
BEGIN
    SET @total += @num;
    SET @num += 1;
    PRINT @total;
    IF @num = 5
        BREAK;
END
PRINT @total;
```

Figure 41: Using BREAK

7.5 CASE

```
-- Example using CASE to change the displayed value according to gender
SELECT
    JobTitle,
    BirthDate,
    HireDate,
    CASE Gender
        WHEN 'M' THEN 'MALE'
        WHEN 'F' THEN 'FEMALE'
        ELSE 'N/A'
    END AS Gender
FROM HumanResources.Employee;
```

Figure 42: CASE statement

8 Stored Procedures

Stored procedures are a type of database object in Microsoft SQL Server that allows you to encapsulate a sequence of SQL statements and procedural logic into a single unit.

Benefits of stored procedures:

- Improve performance
    - They are precompiled and stored in the database, reducing network traffic
    - The execution plan is cached, so when the stored procedure is executed, a cache plan is used
- Makes code reusable

When creating a stored procedure, make sure that it returns only the values you want in your query. This will significantly improve the performance of your query.

```sql
-- Create a stored procedure that retrieves the information of a person with a given BussinessEntityID and PersonType
CREATE OR ALTER PROCEDURE sp_PersonDetails
    @bussinessEntityID int,
    @personType VARCHAR(5)
AS
BEGIN
    SELECT BusinessEntityID, PersonType, FirstName, LastName FROM Person.Person
    WHERE BusinessEntityID = @bussinessEntityID AND PersonType = @personType
END
GO
EXECUTE sp_PersonDetails 200, 'EM'
GO

-- Create a stored procedure that retrieves the highest BussinessEntityID of a person
CREATE OR ALTER PROCEDURE sp_FindMaxPerson
    @result int OUTPUT
AS
BEGIN
    SELECT @result = MAX(BusinessEntityID) FROM Person.Person;
END
GO
DECLARE @result INT;
EXECUTE sp_FindMaxPerson @result OUTPUT;
PRINT(@result);
GO
```

Figure 43: Stored Procedures

9 Triggers

A trigger is a special type of stored procedure that is automatically executed in response to a specific database event such as insert, update or delete operation on a table.

Triggers are used to enforce data integrity rules, perform data validation and automate certain tasks.

Types of triggers:

- After triggers
    - AFTER INSERT
    - AFTER UPDATE
    - AFTER DELETE
- Instead of triggers
    - INSTEAD OF INSERT
    - INSTEAD OF UPDATE
    - INSTEAD OF DELETE
- DDL triggers
- Log on Triggers

```sql
CREATE [OR ALTER] TRIGGER [Schema_Name].Trig_Name
ON TABLE
AFTER INSERT | UPDATE | DELETE
AS
BEGIN
        -- Statements
        -- Insert, Update, Or Delete Statements
END
```

```sql
CREATE [OR ALTER] TRIGGER [Schema_Name].Trig_Name
ON TABLE | VIEW
INSTEAD OF INSERT | UPDATE | DELETE
AS
BEGIN
        -- Statements
        -- Insert, Update, Or Delete Statements
END
```

Figure 44: Trigger syntax

10 Cursors

SQL Server Cursors provides a way to iterate and process data row by row. However, they should be used judiciously due to their potential impact on performance.

It's crucial to evaluate alternative solutions and consider cursor usage as a last resort.

Types of cursors:

- Forward-Only Cursors
    - Allow only forward movement through the result set
    - Cannot move backward or reposition
    - Most efficient type of cursor
- Static Cursors
    - Create a temporary copy of the result set in TempDB
    - Changes in the underlying data are not reflected in the cursor
    - Allow forward and backward movement through the result set
- Dynamic Cursors
    - Reflect the changes made to the underlying data
    - Allow forward and backward movement
    - Performance can be slower compared to static cursors
- Keyset Cursors
    - Similar to dynamic cursors but with improved performance
    - Reflect the changes made to the underlying data
    - Allow forward and backward movement

Performance considerations:

- Minimize the use of cursors
- Fetch only necessary columns
- Use appropriate cursor type
- Keep transactions short

11 Transactions

A transaction is a logical unit of work. Either all the work is completed as a whole unit or none of it happens.

Transactions in SQL Server:

- All DML statements such as INSERT, UPDATE and DELETE
- All DDL statements such as CREATE TABLE and CREATE INDEX

To control the result of a transaction, we use:

- COMMIT
  - Marks the end of a successful transaction
- ROLLBACK
  - Rolls back a transaction to the beginning of the transaction, or to a save point inside the transaction

Properties of a transaction – ACID:

- Atomicity
  - Every transaction is an atomic unit of work
  - All database changes in the transaction succeed or none of them succeed
- Consistency
  - Every transaction, whether successful or not, leaves the database in a consistent state as defined by all objects and database constraints
  - If your transaction attempts to insert a row that is an invalid foreign key, SQL Server will detect the violation of a constraint and generate an error message
- Isolation
  - A transaction's changes are isolated from the activities of any other transaction.
  - If two transactions want to change the same data, one of them has to wait for the other to finish.
- Durability
  - Every transaction endures through an interruption of service. When service is restored, our committed transactions are rolled forward and all uncommitted transactions are rolled back.
  - SQL Server maintains transactional durability by using the database transaction log.

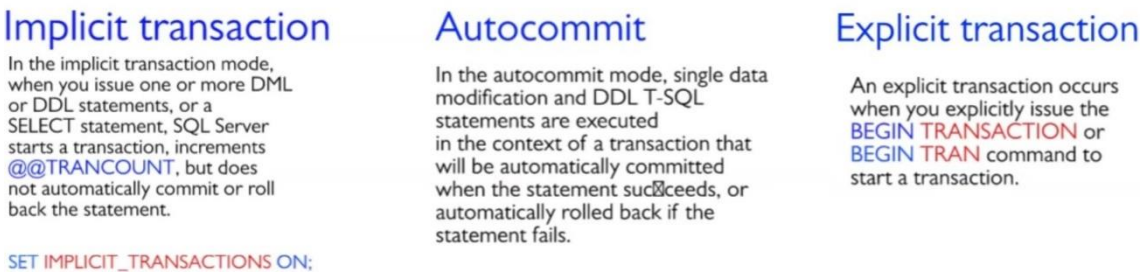


Figure 45: Transaction modes

To preserve the isolation of transactions, SQL Server implements a set of locking protocols. At the basic level, there are two general modes of locking:

- Shared locks
  - Used for sessions that read data
- Exclusive locks
  - Used for changes to data

Requested	Granted	
	Exclusive (X)	Shared (S)
Exclusive	No	No
Shared	No	Yes

Figure 46: Lock compatibility

Blocking – if two sessions request an exclusive lock on the same resource, and one is granted the request, then the other session must wait until the first releases its exclusive lock.

Deadlocking – a deadlock results from a mutual blocking between two or more sessions.

Reference Scripts

DDL

https://github.com/ChicoFinels/T-SQL/blob/main/Scripts/DDL.sql

DML – Read Data

https://github.com/ChicoFinels/T-SQL/blob/main/Scripts/DML%20-%20Read%20Data.sql

DML – Insert Data

https://github.com/ChicoFinels/T-SQL/blob/main/Scripts/DML%20-%20Insert%20Data.sql

DML – Update Data

https://github.com/ChicoFinels/T-SQL/blob/main/Scripts/DML%20-%20Update%20Data.sql

DML – Delete Data

https://github.com/ChicoFinels/T-SQL/blob/main/Scripts/DML%20-%20Delete%20Data.sql

Views

https://github.com/ChicoFinels/T-SQL/blob/main/Scripts/Views.sql

Variables

https://github.com/ChicoFinels/T-SQL/blob/main/Scripts/Variables.sql

Functions

https://github.com/ChicoFinels/T-SQL/blob/main/Scripts/Functions.sql

Control-of-Flow Statements

https://github.com/ChicoFinels/T-SQL/blob/main/Scripts/Control-of-Flow%20Statements.sql

Stored Procedures

https://github.com/ChicoFinels/T-SQL/blob/main/Scripts/Stored%20Procedures.sql

After Triggers

https://github.com/ChicoFinels/T-SQL/blob/main/Scripts/After%20Triggers.sql

Instead Triggers

https://github.com/ChicoFinels/T-SQL/blob/main/Scripts/Instead%20Triggers.sql

Cursors

https://github.com/ChicoFinels/T-SQL/blob/main/Scripts/Cursors.sql