Module “Data processing"

Submodule “T-SQL"

DDL, DML. CRUD. Store procedure language

UA Resource Development Unit

2020

Content

DDL
DML
CRUD
Indexers
User-definded functions
Procedures
Triggers

T-SQL

BD object	Description
Tables	Tables are database objects that contain all the data in a database.
Views	A view is a virtual table whose contents are defined by a query. Like a table, a view consists of a set of named columns and rows of data.
Indexes	Speed up data access operations, indexes are generated based on one or more fields
Triggers	Triggers is a special type of stored procedure that automatically takes effect when a DML event takes place that affects the table.
Stored procedures	A stored procedure in SQL Server is a group of one or more Transact-SQL statements
User-defined functions	User-defined functions are routines that accept parameters, perform an action, such as a complex calculation, and return the result of that action as a value.
Constraints	Constraints are used to specify rules for the data in a table. Constraints are used to limit the type of data that can go into a table.

DDL

Define DB Objects

Besides working with data, you can also work with database objects using CREATE, DROP and ALTER statements.
CREATE statement creates tables, relationships between tables and other database objects

```
create table category (
id_category integer not null,
name varchar(15),
constraint pk_category primary key (id_category));

create table product (
id_product integer not null primary key,
name varchar(15),
id_category integer,
constraint fk1_product foreign key (id_category) references
category(id_category));
```

Define DB Objects

Constrain declaration

A table can contain only one PRIMARY KEY constraint.

All columns defined within a PRIMARY KEY constraint must be defined as NOT NULL. If nullability is not specified, all columns participating in a PRIMARY KEY constraint have their nullability set to NOT NULL.

Constraints can be specified when the table is created with the CREATE TABLE statement, or after the table is created with the ALTER TABLE statement.

```
<foreign key constraint (table)> ::=
FOREIGN KEY (<column> [, < column >]...)
REFERENCES [<schema>.]<table> [(< column > [, <ccolumn >]...)]
[ ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
[ ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
```

Define DB objects

Foreign keys declaration

```
create table organization (
…
constraint pk_organization primary key (cod),
constraint fk1_organization
foreign key (codctr, codreg) references refreg (codctr, codreg)
on delete set null
on update cascade,

constraint fk2_organization
foreign key (codformorg) references refformorg (cod)
on delete set default
on update cascade
…
```

Define DB objects

ALTER statement modifies existing database objects: tables, relationships between tables, etc

- ✓ adding an attribute to a table:

```
alter table client add adress varchar(30);
```

- ✓ adding a foreign key to a table:

```
alter table product add constraint fk1_product foreign key
(id_category) references category (id_category);
```

Define DB objects

DROP statement deletes already existing database objects or their components: tables, relationships between tables, table fields, etc.

- ✓ removing foreign key:

```
alter alter table product drop constraint fk1_product;
```

- ✓ removing tables:

```
drop table orders;
drop table contracts, product, client, category;
```

Define DB objects

OBJECT_ID - Returns the database object identification number
The following example checks for the existence of a specified table by verifying that the table has an object ID. If the table exists, it is deleted. If the table does not exist, the DROP TABLE statement is not executed.

```
IF OBJECT_ID ('fk_Purchases1') IS NOT NULL
alter table purchases DROP Constraint fk_Purchases1;

IF OBJECT_ID ('fk_Purchases2') IS NOT NULL
alter table purchases DROP Constraint fk_Purchases2;

IF OBJECT_ID ('Goods') IS NOT NULL
DROP TABLE Goods;
```

Define DB objects

A view is a virtual table whose contents are defined by a query. Like a table, a view consists of a set of named columns and rows of data. The rows and columns of data come from tables referenced in the query defining the view and are produced dynamically when the view is referenced.

The query that defines the view can be from one or more tables or from other views in the current or other databases

You can modify the data of the base table through the view as long as the following conditions are met:

- ✓ Any changes, including UPDATE, INSERT, and DELETE statements, must refer to columns of only one base table.
- ✓ Columns to be modified in the view must directly reference the column data of the base table. Columns cannot be formed in any other way, including:
    - using an aggregate function: for example, AVG, COUNT, SUM, MIN, MAX , etc;
    - based on calculation that include several columns.
    - columns formed using the UNION, UNION ALL, CROSSJOIN and other similar operators

Define DB objects

Creating and usage view

```
-- Create the view
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 ;

-- Query the view
SELECT FirstName, LastName, HireDate
FROM HumanResources.EmployeeHireDate
ORDER BY LastName;
```

DML

Selecting and Filtering data

Data types in MS SQL Server

Numeric types (int – bigint – bit – decimal – smallint – money – smallmoney – tinyint float – real)
Strings (Character: char – varchar – text Unicode character: nchar – nvarchar - ntext)
Data and Time (Time (12:35:29. 1234567), Date   (2007-05-08), Smalldatetime (2007-05-08 12:35:00), Datetime (2007-05-08 12:35:29.123) etc.)
Other (cursor, table, timestamp and more)

https://docs.microsoft.com/en-us/sql/t-sql/data-types/numeric-types?view=sql-server-ver15
https://docs.microsoft.com/en-us/sql/t-sql/data-types/date-and-time-types?view=sql-server-ver15
https://docs.microsoft.com/en-us/sql/t-sql/data-types/string-and-binary-types?view=sql-server-ver15

Selecting and Filtering data

Supported data types:

Numeric

int – bigint – bit – decimal – smallint – money – smallmoney – tinyint

float – real

Date and time

date – time – datetime – datetime2 – datetimeoffset - smalldatetime

Binary strings

binary– varbinary - <u>image</u>

Strings

Character:

char – varchar – <u>text</u>

Unicode character:

nchar – nvarchar - <u>ntext</u>

Other

cursor – timestamp – table

…and more

Selecting and Filtering data

Data types in MS SQL Server

Char vs varchar:

```
declare @variable1 as varchar(5) = 'abc';
declare @variable2 as char(5) = 'abc';

select datalength(@variable1) as type1, datalength(@variable2)as
type2;
```

Type1 – 3
Type2 - 5

Selecting and Filtering data

Data types in MS SQL Server

If you use **char** or **varchar**, we recommend to:

- ✓ Use **char** when the sizes of the column data entries are consistent.
- ✓ Use **varchar** when the sizes of the column data entries vary considerably.
- ✓ Use **varchar**(**max**) when the sizes of the column data entries vary considerably, and the string length might exceed 8,000 bytes.
- ✓ If using a lower version of the SQL Server Database Engine, consider using the Unicode **nchar** or **nvarchar** data types to minimize character conversion issues.

Selecting and Filtering data

Data types in MS SQL Server

Data and Time:

```
declare @date date= '12-10-25';
declare @datetime datetime= @date;

select @date as '@date', @datetime as '@datetime';

--result
--@date      @datetime
------------ -----------------------
--2025-12-10 2025-12-10 00:00:00.000
```

Selecting and Filtering data

atribute names

source (table)

```
select CityID, CityName from Cities where CountryID > 2
```

Selecting statement

filtering statement

filtering statement

Selecting and Filtering data

The **SELECT** query reads the data and returns it:

Get the entire list of all cities in Ukraine:

Almost always, a SELECT query returns a dataset as a relation, and this is an important feature that should not be forgotten, since you can perform the same actions on a result set as on a regular table.

```
Select CityID,CityName From Cities
where CountryID = 1
```

CityID	CityName
1	Kyiv
2	Lviv
3	Odessa
4	Dnipro

Selecting and Filtering data

Instead of specifying the specific names of the fields that you want to select, you can specify an abbreviation:

```
Select * From Countries
```

CountryID	CountryName
1	Ukraine
2	Belarus
3	Spain
4	USA

Selecting and Filtering data

Using **DISTINCT**

In some cases, you may have duplication in the result set, for example, if you modify the previous statement:

```
Select case CountryID
    when 1 then 'Belarus'
    when 2 then 'Russia'
    when 3 then 'Ukraine'
    else 'Unknown'
   End
From Cities
```

	CountryName
1	Ukraine
2	Ukraine
3	Ukraine
4	Ukraine
5	Belarus
6	Belarus
7	Belarus
8	Belarus
9	Spain

Selecting and Filtering data

In order to remove duplication, you need to use the DISTINCT operator:

```
Select DISTINCT
case CountryID
when 1 then 'Ukraine'
…
```

	CountryName
1	Ukraine
2	Belarus
3	Spain

Selecting and Filtering data

Selection condition WHERE

Data tables can contain many records, millions of records. It is very rare to select all records with a regular **SELECT**.
In most cases, you will select data following some condition. The **WHERE** clause allows you to implement such a selection:

Select all departments of the city of Kyiv (ID = 1):

```
Select * from Departments where CityID = 1
```

Selecting and Filtering data

The Filter operations:

- ✓ **Arithmetic** +, -, *, /, %
- ✓ **Comparing** =, >, <, <> (not equal to), ! (not), >= , <=
- ✓ **Logical** AND, OR, NOT
- ✓ **Concatenation** +

```
SELECT EmployeeKey, LastName  FROM DimEmployee
WHERE (EmployeeKey <= 500) AND (LastName LIKE '%Smi%’) AND (FirstName
LIKE '%A%’);

SELECT EmployeeKey, LastName FROM DimEmployee WHERE (EmployeeKey = 1)
OR (EmployeeKey = 8) OR (EmployeeKey = 12);
```

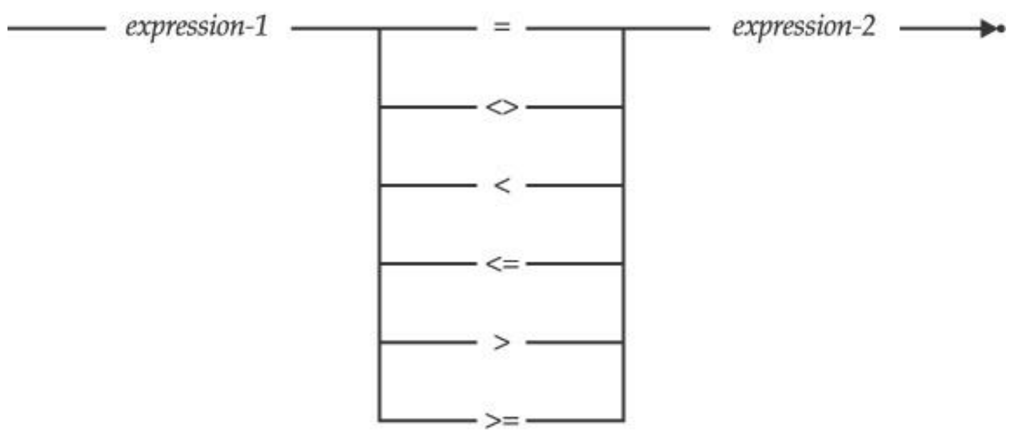
Selecting and Filtering data

SQL Server allows you to check various conditions:

- ✓ Comparison for compliance, more, less;
- ✓ Entering the range;
- ✓ Entering a specific set of values;
- ✓ Pattern matching;
- ✓ Checking for null
- ✓ Existence check.

Selecting and Filtering data

Matching comparison, more, less:

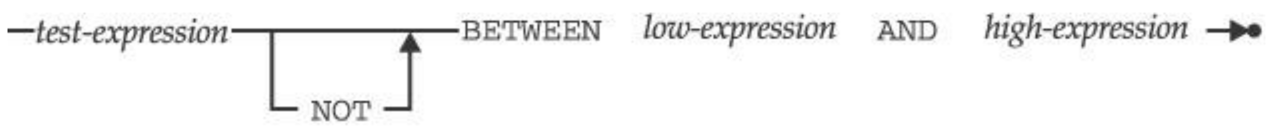


All departments opened before January 1, 2011:

```
Select * from Departments
where DateOfCreation < '2011-01-01'
```

Selecting and Filtering data

Range entry (BETWEEN):

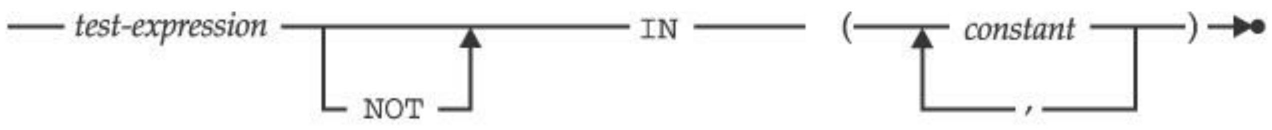


All departments opened from January to December 2010:

```
Select * from Departments
where DateOfCreation between '2010-01-01' and '2011-12-01'
```

Selecting and Filtering data

Entering a specific set (IN):



Find all departments in Kyiv, Dnipro:

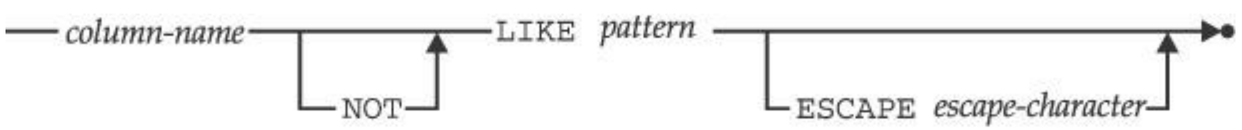
```
Select * from Departments
where CityID in (1,4)
```

Find all departments outside Kyiv:

```
Select * from Departments
where CityID not in (1)
```

Selecting and Filtering data

Pattern matching (LIKE):



Determines whether a specific character string matches a specified pattern
Pattern chars:

- **%** - any string of zero or more characters;
- **_** - any single character;
- **escape** – set the escape-character.

Selecting and Filtering data

Like - % , _ , escape

```
SELECT FirstName, LastName, Phone FROM DimEmployee
WHERE phone NOT LIKE '612%’

SELECT FirstName, LastName, Phone FROM DimEmployee
WHERE phone NOT LIKE ‘6_2%’

DECLARE @variable1 as varchar(10) = 'abc%asd';
SELECT @variable1
WHERE @variable1 LIKE '%$%%'ESCAPE '$'
```

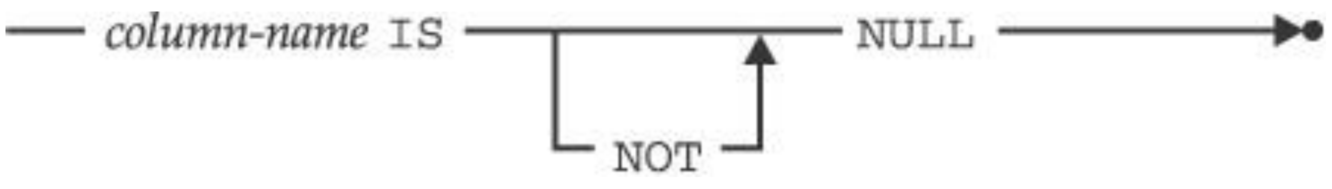
Selecting and Filtering data

NULL values:

- ✓ **NULL** are **UNKNOWN** values (used in operations as an error indicator)
- ✓ **NULL** is not equal to **0** or **empty** string
- ✓ Comparing **NULL** to any other value (as well as **NULL**) will return **UNKNOWN**
- ✓ You can get records containing **NULL** using the **IS [NOT] NULL** operator

Selecting and Filtering data

Checking for NULL



Find all departments that do not have a creation date:

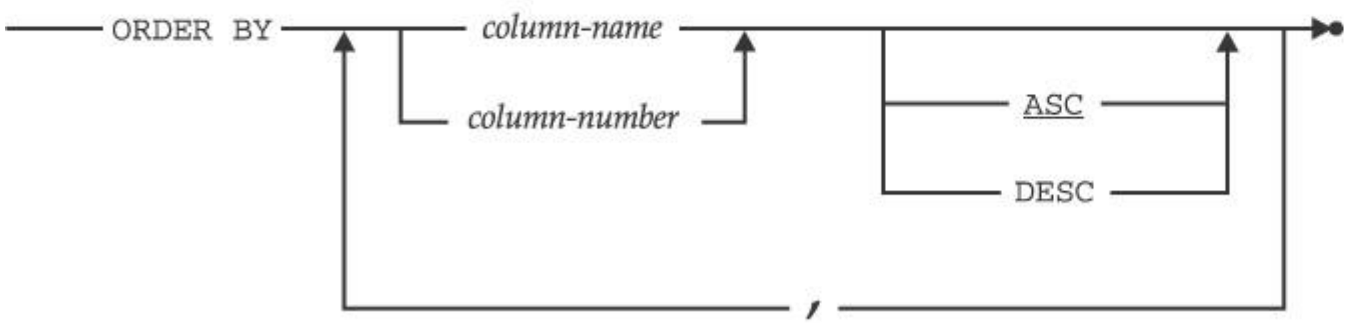
```
Select * from Departments
where DateOfCreation IS NULL
```

DepartmentID	CityID	DepartmentName	DateOfCreation
21	5	Office	NULL
22	7	Store	NULL

Selecting and Filtering data

Sorting ORDER BY:

The result set can be sorted before further use. This is done using the ORDER BY operator:



Selecting and Filtering data

Using ORDER BY:

By default, data is sorted in ascending order:

```
Select DepartmentName,DateOfCreation from Departments
order by DepartmentName
```

DepartmentName	DateOfCreation
% Store	2011-05-01
A Store	2011-02-01
B Store	2011-01-11
HR	2011-01-01

To sort in reverse order, use the following syntax:

```
Select DepartmentName,DateOfCreation from Departments
order by DepartmentName desc
```

Grouping and Aggregating data

Functions

Microsoft SQL Server has a number of built-in features.
Functions from the “Aggregate” category allow you to perform operations on datasets, but return only one aggregated value:

AVG()
MIN()
MAX()
COUNT()
GROUPING()
CHECKSUM_AGG()
STDEVP()
VAR()
STDEV()
SUM()
VARP()
COUNT_BIG()

Grouping and Aggregating data

Aggregate functions

In the first column, the average price for the Price column is calculated, in the second, the quantity of all goods:

```
SELECT
AVG(pr.Price), SUM(pr.Count) FROM Products AS pr
```

81.6666	**7**

Grouping and Aggregating data

Result sets can be grouped by some criterion

Count the number of offices of all types:

```
SELECT dp.DepartmentName, COUNT(dp.DepartmentName)
FROM Departments AS dp
GROUP BY dp.DepartmentName
```

DepartmentName	(No column name)
% Store	1
A Store	1
B Store	4
HR	1

Grouping and Aggregating data

Grouping by multiple parameters

Count the number of offices of all types in selected cities:

```
SELECT dp.DepartmentName, ct.CityName, COUNT(dp.DepartmentName)
FROM Departments as dp
JOIN Cities AS ct ON ct.CityID = dp.CityID
GROUP BY dp.DepartmentName, ct.CityName
```

DepartmentName	CityName	(No column name)
% Store	Kyiv	1
A Store	Lviv	1
B Store	Lviv	4
HR	Dnipro	1

Grouping and Aggregating data

Grouping with filtering

Count the number of offices of all types in selected cities with more than 2 offices:

```
SELECT …
GROUP BY dp.DepartmentName, ct.CityName
HAVING COUNT(dp.DepartmentName) > 1
```

DepartmentName	CityName	(No column name)
% Store	Kyiv	1
% Store	Lviv	1

Grouping and Aggregating data

Having vs where

```
SELECT ProductID
FROM SalesOrderDetail
WHERE UnitPrice < 25.00
GROUP BY ProductID
HAVING AVG(OrderQty) > 5
ORDER BY ProductID;

SELECT ProductID, AVG(OrderQty) AS AverageQuantity, SUM(LineTotal) AS
Total
FROM SalesOrderDetail
GROUP BY ProductID
HAVING SUM(LineTotal) > $1000000.00 AND AVG(OrderQty) < 3;
```

Grouping and Aggregating data

Having vs where

- To find the highest temperature among city min ones

```
SELECT max(temp_lo) FROM weather;
```

- In what cities it happened

```
SELECT city FROM weather WHERE temp_lo = (SELECT max(temp_lo) FROM weather);
SELECT city FROM weather WHERE temp_lo = (SELECT max(temp_lo) FROM weather);
```

- To get max temperature for every city

```
SELECT city, max(temp_lo) FROM weather GROUP BY city;
```

- To get cities started with “S” where max temperature not more than 40

```
SELECT city, max(temp_lo) FROM weather WHERE city LIKE 'S%' GROUP BY city HAVING max(temp_lo) < 40;
```

Grouping and Aggregating data

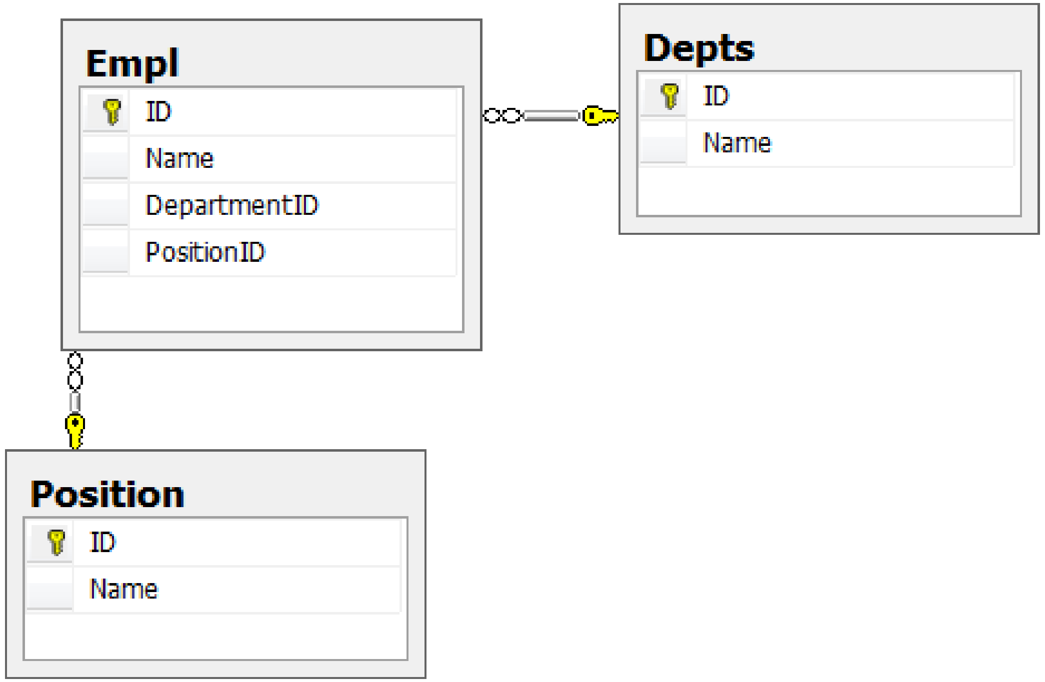
```
-- table Goods {Id, Name}
-- table Purchases {Id, OrderId, GoodId, GoodCount, Price}
SELECT goodname = g.Name, sum(p.GoodCount) number
FROM Goods g
JOIN Purchases p ON g.Id = p.GoodId
GROUP BY g.Name
HAVING SUM(p.GoodCount) > 5 – not number!
ORDER BY number
WHERE … – is it possible?
```

Yes, But…

```
SELECT * FROM
(SELECT goodname = g.Name, sum(p.GoodCount) number
FROM Goods g
JOIN Purchases p ON g.Id = p.GoodId
GROUP BY g.Name
HAVING SUM(p.GoodCount) > 5) s
WHERE …
ORDER BY s.number
```

Joining

In most cases, the required data is in different relations.
The mechanism that allows us to retrieve data from multiple tables is called joining.



Joining

Cartesian product

Let's try to get the names of employees with their roles:

```
SELECT Empl.Name, Depts.Name from Empl, Depts
```

Such a request will return as many as 24 records, although our employees only 6:

	Name	Name		Name	Name		Name	Name
1	Mike	Production	9	Eric	TC	17	Jessie	Maintenance
2	John	Production	10	Paul	TC	18	Jonny	Maintenance
3	Eric	Production	11	Jessie	TC	19	Mike	NULL
4	Paul	Production	12	Jonny	TC	20	John	NULL
5	Jessie	Production	13	Mike	Maintenance	21	Eric	NULL
6	Jonny	Production	14	John	Maintenance	22	Paul	NULL
7	Mike	TC	15	Eric	Maintenance	23	Jessie	NULL
8	John	TC	16	Paul	Maintenance	24	Jonny	NULL

We got the Cartesian product - a combination of all records from the left and right tables. This set must be correctly connected (filtered).

Joining

Joining by comparison

The easiest way to do a query like this is to use the WHERE clause:

```
SELECT Empl.Name,  Depts.Name  from  Empl,Depts
WHERE Empl.DepartmentID = Depts.ID
```

Name	Name
Mike	Production
John	TC
Paul	Maintenance
Jessie	Production

There are 4 records in total, since 2 employees have a DepartmentID value of NULL.
Filtering the set manually is an outdated approach.

Joining

Joins

Instead of manual filtering, you should use one of the built-in join types:

OUTER JOIN
FULL, LEFT, RIGHT

INNER JOIN

CROSS JOIN

SELF JOIN

Joining

FULL LEFT JOIN (LEFT JOIN)

Returns possible combinations from the right table for each record on the left (including the NULL value) that satisfy the selection condition:

```
SELECT Empl.Name, Depts.Name from Empl
LEFT JOIN Depts ON Empl.DepartmentID = Depts.ID
```

In the resulting set, we can see even those employees who do not have a specified department:

Name	Name
Mike	Production
John	TC
Eric	NULL
Paul	Maintenance
Jessie	Production
Jonny	NULL

Joining

FULL RIGHT JOIN (RIGHT JOIN)

Returns all possible combinations from the left table for each record on the right (including the NULL value) that satisfy the selection condition

```
SELECT Empl.Name, Depts.Name from Empl
RIGHT JOIN Depts ON Empl.DepartmentID = Depts.ID
```

We have a “Secret” department, to which no employee is assigned, and now we see it:

Name	Name
Mike	Production
Jessie	Production
John	TC
Paul	Maintenance
NULL	Secret

Joining

FULL OUTER JOIN (FULL JOIN)

It essentially combines the results of a LEFT and RIGHT join:

```
SELECT Empl.Name, Depts.Name from Empl
FULL JOIN Depts ON Empl.DepartmentID = Depts.ID
```

We see employees without departments, as well as departments without employees:

Name	Name
Mike	Production
John	TC
Eric	NULL
Paul	Maintenance
Jessie	Production
Jonny	NULL
NULL	Secret

Joining

INNER JOIN (JOIN)

Selects rows that strictly satisfy a condition:

```
SELECT Empl.Name, Depts.Name from Empl
JOIN Depts ON Empl.DepartmentID = Depts.ID
```

Name	Name
Mike	Production
John	TC
Paul	Maintenance
Jessie	Production

Joining

CROSS JOIN

Returns the Cartesian product:

```
SELECT Empl.Name, Depts.Name from Empl
CROSS JOIN Depts
```

Similar to the results previously:

	Name	Name		Name	Name		Name	Name
1	Mike	Production	9	Eric	TC	17	Jessie	Maintenance
2	John	Production	10	Paul	TC	18	Jonny	Maintenance
3	Eric	Production	11	Jessie	TC	19	Mike	NULL
4	Paul	Production	12	Jonny	TC	20	John	NULL
5	Jessie	Production	13	Mike	Maintenance	21	Eric	NULL
6	Jonny	Production	14	John	Maintenance	22	Paul	NULL
7	Mike	TC	15	Eric	Maintenance	23	Jessie	NULL
8	John	TC	16	Paul	Maintenance	24	Jonny	NULL

Joining

SELF JOIN

Joining a table to itself:

```
SELECT e1.PositionID, e1.Name,e2.PositionID, e2.Name from Empl e1
JOIN Empl e2
on e1.PositionID < e2.PositionID
```

We get something like a hierarchy of employees:

PositionID	Name	PositionID	Name
1	Mike	2	John
1	Mike	5	Eric
2	John	5	Eric
2	Paul	5	Eric
2	Jessie	5	Eric
1	Mike	2	Paul
1	Mike	2	Jessie

Subqueries

Queries can contain other queries inside. Such nested queries named subqueries.

This query will return all items with price above the average one:

```
select * from Products where
Price > (select avg(Price)
from Products)
```

ID	Name	Price	Count
1	Spoon	5.4	5
2	Knife	6.8	6
3	Fork	3.4	4

Subqueries

You can also use different conditions for filtering in form of subqueries.

This query will return all items with position ID which exist in the list of positions with ID above 1:

```
select * from Empl
where Empl.PositionID in (select Position.ID from Position where
PositionID > 1)
```

ID	Name	DepartmentID	PositionID
2	John	5.4	2
3	Paul	6.8	5
4	Jessie	3.4	1

Subqueries

You can insert subqueries instead of the attribute name in queries.

This query will return following row set:

```
Select dp.DepartmentName,
(Select CityName from Cities where CityID = dp.CityID)
from Departments as dp
```

	DepartmentName	No column name
1	Store	Kyiv
2	HR	Borispol
3	Support	Kharkov

Subqueries

You can insert subqueries into expressions.

This query finds the prices of all mountain bike products, their average price, and the difference between the price of each mountain bike and the average price.:

```
SELECT Name, ListPrice,
(SELECT AVG(ListPrice) FROM Production.Product) AS Average,
ListPrice - (SELECT AVG(ListPrice) FROM Production.Product) AS Difference
FROM Production.Product
WHERE ProductSubcategoryID = 1;
```

More examples https://docs.microsoft.com/en-us/sql/relational-databases/performance/subqueries?view=sql-server-ver15

Subqueries

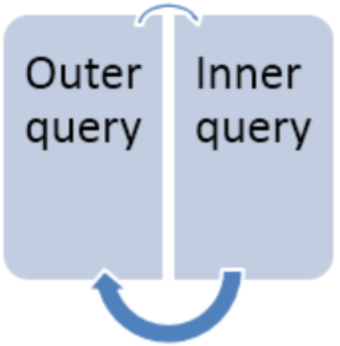
You can insert subqueries for existence check.

The following query finds the names of all products that are in the Wheels subcategory:

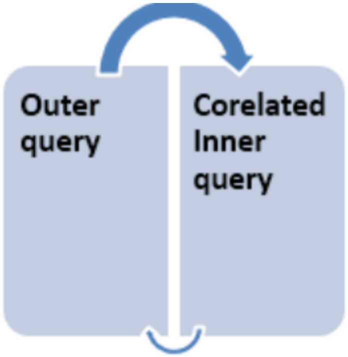
```
SELECT Name FROM Production.Product
WHERE EXISTS
    (SELECT *
     FROM Production.ProductSubcategory
     WHERE ProductSubcategoryID =
            Production.Product.ProductSubcategoryID AND Name = 'Wheels');
```

Correlated Subqueries

Many queries can be evaluated by executing the subquery once and substituting the resulting value or values into the WHERE clause of the outer query. In queries that include a correlated subquery (also known as a repeating subquery), the subquery depends on the outer query for its values. This means that the subquery is executed repeatedly, once for each row that might be selected by the outer query.



Ordinary Sub Query



Correlated Sub Query

Correlated Subqueries

The subquery in this statement cannot be evaluated independently of the outer query. It needs a value for Employee.BusinessEntityID, but this value changes as SQL Server examines different rows in Employee.

This query retrieves one instance of each employee's first and last name for which the bonus in the SalesPerson table is 5000 and for which the employee identification numbers match in the Employee and SalesPerson tables:

```
SELECT DISTINCT c.LastName, c.FirstName, e.BusinessEntityID
FROM Person AS c JOIN Employee AS e
ON e.BusinessEntityID = c.BusinessEntityID
WHERE 5000.00 IN
    (SELECT Bonus FROM SalesPerson sp WHERE e.BusinessEntityID =
sp.BusinessEntityID
```

Correlated Subqueries

For every row of the outer query:

```
SELECT DISTINCT c.LastName,
c.FirstName, e.BusinessEntityID
FROM Person …
WHERE 5000.00 IN
    (SELECT Bonus…
```

285

50.0, …

```
SELECT Bonus
FROM SalesPerson
WHERE BusinessEntityID =
e.BusinessEntityID
```

CRUD

CRUD

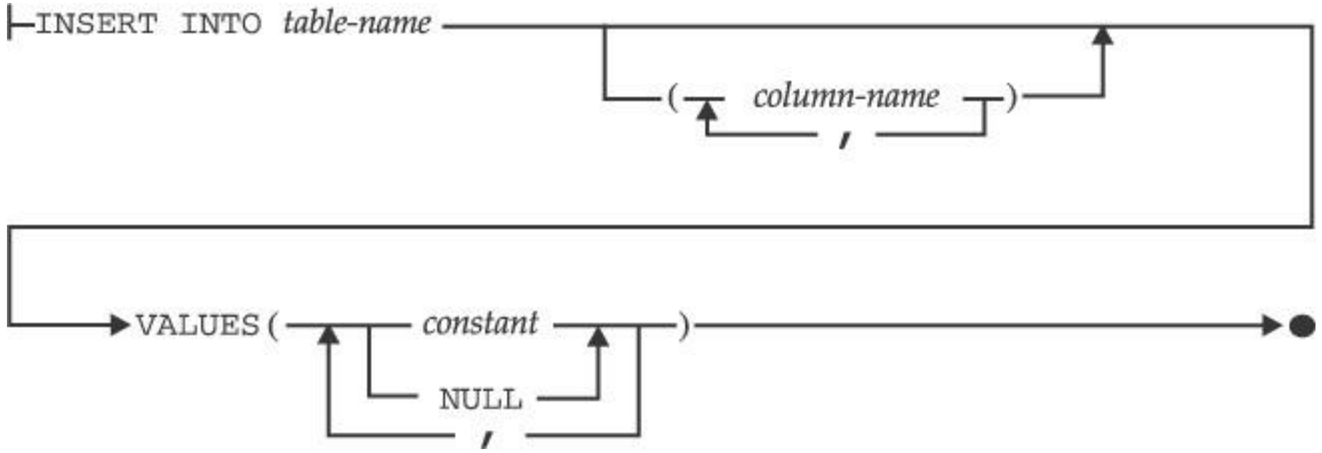
In addition to reading data, very often you have to insert new data, update existing records, as well as delete data.

The following operators are used to perform these operations:

- ✓ INSERT
- ✓ UPDATE
- ✓ DELETE

CRUD

Statement **INSERT** adds new rows into a table.
You can add one row as well as several ones



CRUD

Statement **INSERT**

```
INSERT INTO Products (Name, Price, Count)
VALUES ('ProductN','12,00’,5)
-- or
INSERT INTO Products
VALUES ('ProductN','12,00',5)
```

If attribute value is generated automatically, this attribute isn’t inserted:

```
TABLE Products (
       ID int IDENTITY(1,1) NOT NULL,
       Name nvarchar(50) NOT NULL,
```

CRUD

How to insert from one table into another.

All products are copied:

```
INSERT INTO OldProducts
SELECT Name, Price,[Count] FROM Products
```

ID	Name	Price	Count
1	Spoon	5.4	5
2	Knife	6.8	6
3	Fork	3.4	4

CRUD

Statement **DELETE**

To delete one row:

```
DELETE FROM OldProducts WHERE ID='1'
```

To delete several rows:

```
DELETE FROM OldProducts WHERE Price > '1,00'
```

To clear table:

```
DELETE FROM OldProducts
```

CRUD

Statement **DELETE** with subquery

The rows from the SalesPersonQuotaHistory table are deleted based on the year-to-date sales stored in the SalesPerson table:

```
DELETE FROM Sales.SalesPersonQuotaHistory
WHERE BusinessEntityID IN (SELECT BusinessEntityID FROM Sales.SalesPerson
WHERE SalesYTD > 2500000.00);
```

CRUD

Statement **UPDATE**

Update the name of the department "HR", replacing it with "Human Resources" :

```
UPDATE Departments
SET
DepartmentName = 'Human Resources'
WHERE DepartmentName = 'HR'
```

CRUD

Statement **UPDATE** with subquery

Update the vacation hours of the 10 employees with the earliest hire dates:

```
UPDATE HumanResources.Employee
SET VacationHours = VacationHours + 8

FROM (SELECT TOP 10 BusinessEntityID FROM HumanResources.Employee ORDER
BY HireDate ASC) AS th WHERE HumanResources.Employee.BusinessEntityID =
th.BusinessEntityID;
```

Indexes

Indexes are special lookup tables that the database search engine can use to speed up data retrieval.
n index in a database is very similar to an index in the back of a book.

For example, if you want to reference all pages in a book that discusses a certain topic, you first refer to the index, which lists all the topics alphabetically and are then referred to one or more specific page numbers.
When there are thousands of records in a table, retrieving information will take a long time. Therefore indexes are created on columns which are accessed frequently, so that the information can be retrieved quickly.

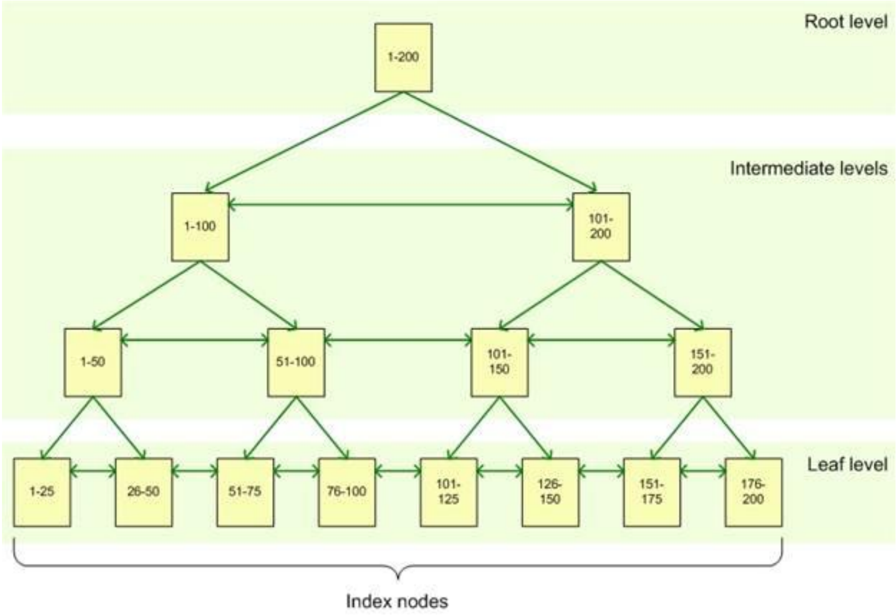
An index helps to speed up **SELECT** queries and **WHERE** clauses, but it slows down data input, with the **UPDATE** and the **INSERT** statements. Indexes can be created or dropped with no effect on the data. Creating an index involves the **CREATE INDEX** statement, which allows you to name the index, to specify the table and which column or columns to index, and to indicate whether the index is in an ascending or descending order.

Indexes

A **clustered** index alters the way that the rows are physically stored. When you create a clustered index on a column (or a number of columns), the SQL server sorts the table's rows by that column(s).
As a result, there can be only one clustered index on a table or view.

Note that primary key automatically creates a clustered index on the "id" column

A **non-clustered** index, on the other hand, does not alter the way the rows are stored in the table. Instead, it creates a completely different object within the table, that contains the column(s) selected for indexing and a pointer back to the table's rows containing the data.



Indexes

Creating index

Creating non-clustered index on the primary key:

```
CREATE TABLE Names(
 idName int IDENTITY(1,1),
 vcName varchar(50),
 CONSTRAINT PK_guid PRIMARY KEY NONCLUSTERED (idName)
)
```

Creating clustered index on the unique attribute:

```
CREATE TABLE Names(
 idName int, vcName varchar(50),
 vcLastName varchar(50), vcSurName varchar(50),
 dBirthDay datetime,
 CONSTRAINT cn_unique UNIQUE CLUSTERED(vcName, vcLastName, vcSurName,
dBirthDay) )
```

Indexes

Creating index

Creating clustered index on the not key attribute:

```
CREATE CLUSTERED INDEX I_CL_vcName
ON TestTable(vcName)
```

Creating non-clustered ordered index:

```
CREATE NONCLUSTERED INDEX I_CL_vcName
ON TestTable(vcName DESC)
```

User-defined functions

Like functions in programming languages, SQL Server user-defined functions are routines that accept parameters, perform an action, such as a complex calculation, and return the result of that action as a value. The return value can either be a single scalar value or a result set.

It can be used for the following purposes:

- ✓ In Transact-SQL statements such as SELECT.
- ✓ In applications as well as the function call.
- ✓ In the definition of another custom function.
- ✓ To parameterize the view.
- ✓ To define a table column.
- ✓ To define a CHECK constraint on a column.
- ✓ To replace a stored procedure

User-defined functions

User-defined function call

User-defined function call if the function returns scalar :

```
SELECT ufn_SaleByGood(IdGood) FROM Goods;
SELECT ufn_SaleByGood(67);
```

User-defined function call if the function returns vector or row set :

```
SELECT * FROM ufn_SalesByStore (602);
```

User-defined functions

User-defined function creating

Sets the local variable, previously created by using the DECLARE @local_variable statement, to the specified value:

```
CREATE FUNCTION FACTORIAL (@N AS INT)
RETURNS FLOAT(8)
AS
BEGIN
   DECLARE @I INT = 1;
   DECLARE @NFACTORIAL FLOAT(8);
   WHILE @I <= @N
   BEGIN
      SET @NFACTORIAL = @NFACTORIAL * @I;
      SET @I = @I + 1;
   END;
   RETURN @NFACTORIAL;
END;
SELECT FACTORIAL(5);
```

User-defined functions

```
CREATE FUNCTION DayOfWeek (@Date datetime) – scalar result
RETURNS int
AS
BEGIN
     DECLARE @day int;
     ...
          SET @day = ...;
     RETURN(@day);
END;
GO
SELECT DayOfWeek('12/26/2004') AS 'WeekDay';
--or
DECLARE @ret int;
EXEC @ret = DayOfWeek @Date = '12/26/2004';
```

User-defined functions

```
CREATE FUNCTION ufn_SalesByStore (@storeid int) -- table result
RETURNS TABLE
AS
RETURN
( --all returned columns should have names(for select)
    SELECT P.ProductID, P.Name, SUM(SD.LineTotal) AS 'Total'
    FROM Production.Product AS P
    JOIN Sales.SalesOrderDetail AS SD ON SD.ProductID = P.ProductID
    JOIN Sales.SalesOrderHeader AS SH ON SH.SalesOrderID =
SD.SalesOrderID
    JOIN Sales.Customer AS C ON SH.CustomerID = C.CustomerID
    WHERE C.StoreID = @storeid
    GROUP BY P.ProductID, P.Name
); SELECT * FROM ufn_SalesByStore (54);
```

User-defined functions

```
CREATE FUNCTION ufn_FindReports (@InEmpID INTEGER) -- table result
RETURNS @retFindReports TABLE
(
    EmployeeID int primary key NOT NULL,
    FirstName nvarchar(255) NOT NULL,
    LastName nvarchar(255) NOT NULL,
    JobTitle nvarchar(50) NOT NULL,
    RecursionLevel int NOT NULL
)
AS

-- Look at continuation
```

User-defined functions

```
BEGIN
-- CTE name and columns
WITH EMP_cte(EmployeeID, OrganizationNode, FirstName, LastName, JobTitle,
RecursionLevel)
    AS (
        SELECT ...
        )
-- copy the required columns to the result of the function
INSERT @retFindReports
   SELECT EmployeeID, FirstName, LastName, JobTitle, RecursionLevel
   FROM EMP_cte
   RETURN
END;
GO
SELECT EmployeeID, FirstName, LastName, JobTitle, RecursionLevel FROM
ufn_FindReports(1);
```

Stored procedures

Stored Procedure in SQL Server can be defined as the set of logically group of SQL statement which are grouped to perform a specific task. The main benefit of using a stored procedure is that it increases the performance of the database

- ✓ Reduces the amount of information sent to the database server
- ✓ Compilation step is required only once when the stored procedure is created
- ✓ SQL code reusability
- ✓ Enhances the security since we can grant permission to the user for executing the Stored procedure instead of giving permission on the tables used in the Stored procedure.

Stored procedures

```
CREATE PROCEDURE FACTORIAL( --scalar result
@N INT,
@NFACTORIAL INT OUTPUT)
AS
BEGIN
   DECLARE @I INT = 1;
   SET @NFACTORIAL = 1;
   WHILE @I <= @N
   BEGIN
      SET @NFACTORIAL = @NFACTORIAL * @I;
      SET @I = @I + 1;
   END;
END;
GO
DECLARE @F FLOAT;
EXEC FACTORIAL 5, @F OUTPUT;
```

Stored procedures

```
PROCEDURE GetstudentnameInOutputVariable(--scalar result
@studentid INT,
@studentname VARCHAR (200) OUTPUT,
@StudentEmail VARCHAR (200)OUTPUT )
AS
BEGIN
SELECT @studentname= Firstname+' '+Lastname,
    @StudentEmail=email FROM tbl_Students
WHERE studentid = @studentid
END
GO
Declare @Studentname as nvarchar(200);
Declare @Studentemail as nvarchar(50);
Execute GetstudentnameInOutputVariable 1 , @Studentname output,
@Studentemail output
Select @Studentname, @Studentemail
```

Stored procedures

```
PROCEDURE GetstudentnameInOutputVariable -- table result
AS
BEGIN
SELECT studentname = Firstname+' '+Lastname,
    StudentEmail = email
FROM tbl_Students
END
GO
Execute GetstudentnameInOutputVariable
--or
GetstudentnameInOutputVariable
-- select * from ProcGetGoods – error (select to select)
-- insert into TableGoods exec ProcGetGoods - possible
```

Stored procedures

```
Create Procedure InsertStudentrecord -- changing data
(
 @StudentFirstName Varchar(200),
 @StudentLastName  Varchar(200),
 @StudentEmail     Varchar(50)
)
As
 Begin
   Insert into tbl_Students (Firstname, lastname, Email)
   Values(@StudentFirstName, @StudentLastName, @StudentEmail)
 End
GO
Exec InsertStudentrecord
```

User-defined functions vs Stored procedures

Stored Procedure	Function
SP have to use EXEC or EXECUTE	Can be used with Select statement
It returns output parameters	It returns table variables
We can’t JOIN SP	We can easily JOIN UDF
SP can have both input and output params	UDF has only input parameters
We can apply both DML operations as well a Select in SP	We can only use Select in it
SP can have transaction	UDF can’t have transactions
UDF can be called from a SP	SP can’t be called from UDF
TRY-CATCH can be used in SP for exception handling	We can’t use TRY-CATCH in UDF
Insert, Update, Delete operations can be performed in SP	Not valid for UDF

Triggers

DML triggers is a special type of stored procedure that automatically takes effect when a data manipulation language (DML) event takes place that affects the table or view defined in the trigger.

DML events include INSERT, UPDATE, or DELETE statements. DML triggers can be used to enforce business rules and data integrity, query other tables, and include complex Transact-SQL statements. The trigger and the statement that fires it are treated as a single transaction, which can be rolled back from within the trigger. If a severe error is detected (for example, insufficient disk space), the entire transaction automatically rolls back.

DML triggers are similar to constraints in that they can enforce entity integrity or domain integrity. In general, entity integrity should always be enforced at the lowest level by indexes that are part of PRIMARY KEY and UNIQUE constraints or are created independently of constraints. Domain integrity should be enforced through CHECK constraints, and referential integrity (RI) should be enforced through FOREIGN KEY constraints. DML triggers are most useful when the features supported by constraints cannot meet the functional needs of the application.

Triggers

There are three action query types that you use in SQL which are INSERT, UPDATE and DELETE. So, there are three types of triggers and hybrids that come from mixing and matching the events and timings that fire them.

Basically, triggers are classified into two main types:

- ✓ After Triggers
- ✓ Instead Of Triggers

After Triggers

These triggers are executed after an action such as Insert, Update or Delete is performed.

Instead of Triggers

These triggers are executed instead of any of the Insert, Update or Delete operations. For example, let’s say you write an Instead of Trigger for Delete operation, then whenever a Delete is performed the Trigger will be executed first and if the Trigger deletes record then only the record will be deleted.

Triggers

Trigger to the Goods table for insertion. Rolls back an insert transaction if the item ID is greater than 100:

```
CREATE TRIGGER Trigger_insert_1
       ON Goods
       FOR INSERT
       AS
       IF @@ROWCOUNT=1
       BEGIN
              SET NOCOUNT ON
              DECLARE @id INT, @name VARCHAR(20)
              SELECT @id=ID, @name=Name FROM inserted
              if @id > 100
              BEGIN
           ROLLBACK TRAN
           PRINT 'good ''' + @name + ''' can''t be insert'
         END
       END
```

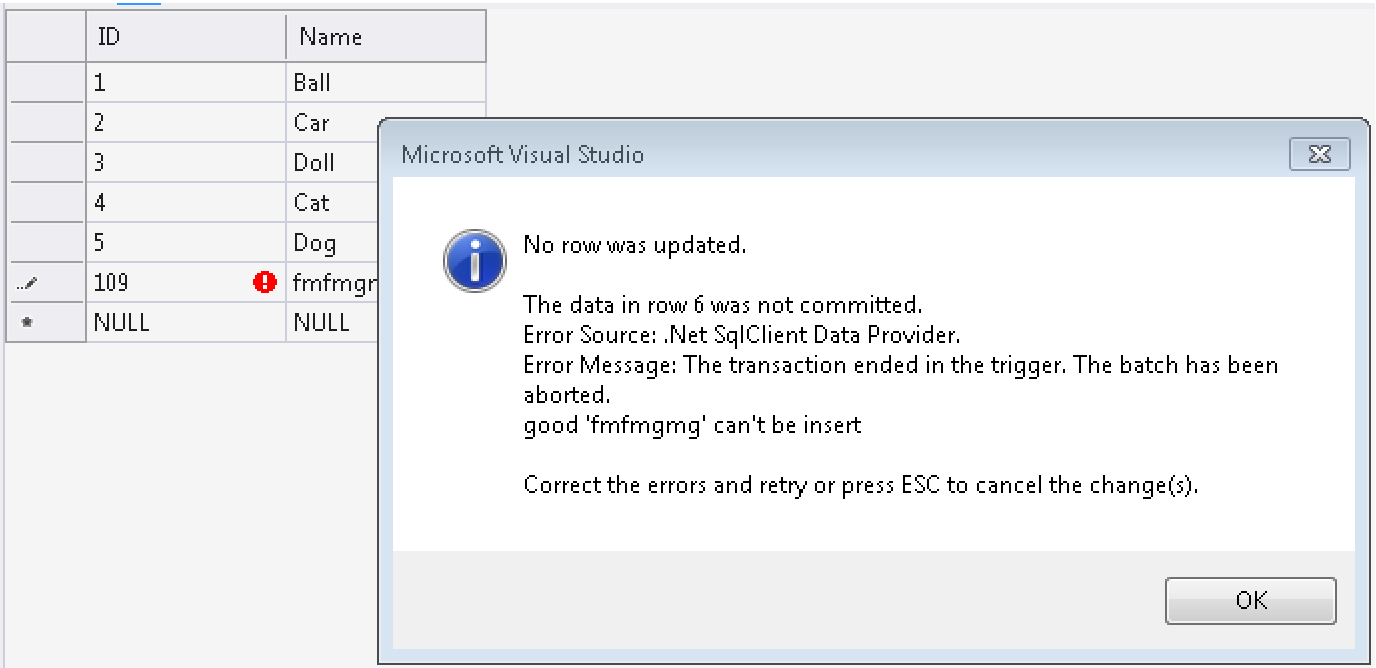
Triggers

Trigger to the Goods table for insertion. Rolls back an insert transaction if the item ID is greater than 100:

```sql
use sqlbd;
GO
insert into Goods (Id, Name) values (109, 'tarakan');
```

T-SQL | Results | Message

```
good 'tarakan' can't be insert
Msg 3609, Level 16, State 1, Line 3
The transaction ended in the trigger. The batch has been aborted.
```



Triggers

The trigger logs all changes in the Goods table:

```
CREATE TRIGGER Trigger_Log
ON Goods
AFTER INSERT, DELETE, UPDATE
AS
BEGIN
       SET NOCOUNT ON
       DECLARE @Operation NVARCHAR(10) = 'unknown',
    --@User NVARCHAR(25),
    @OldIdGood INT,
    @OldName NVARCHAR(20),
    @NewIdGood INT,
    @NewName NVARCHAR(20);

-- Look at continuation
```

Triggers

The trigger logs all changes in the Goods table:

```
-- Continuation
       --SET @User = CURRENT_USER;
       if NOT EXISTS (SELECT * FROM deleted)
              SET @Operation = 'insert'

       if NOT EXISTS (SELECT * FROM inserted)
              SET @Operation = 'delete'

       if @Operation = 'unknown'
              SET @Operation = 'update'

       INSERT INTO GoodLog (Operation, NewIdGood, NewName) SELECT
Operation = @Operation, Id, Name from inserted;
       INSERT INTO GoodLog (Operation, OldIdGood, OldName) SELECT
Operation = @Operation, Id, Name from deleted;
END
```

Content

- https://www.w3schools.com/sql/
- https://docs.microsoft.com/en-us/sql/?view=sql-server-ver15

Q&A

DRIVEN CANDID CREATIVE ORIGINAL INTELLIGENT EXPERT

UA .NET Online LAB