MS SQL SERVER

Training Manual

APRIL 28, 2023

THE COLLAB TECHNOLOGY
Mid Baneswor, Kathmandu

Chapter I

Introduction to SQL

SQL stands for Structured Query Language and was originally developed by IBM in the 70's to interact with relational databases. It is the common language for databases, remains fairly readable and it is relatively simple to learn the basics

- SQL stands for Structured Query Language
- Pronounced as **"ess-que-el"** or **"sequel".**
- Most commercial Relational DBMS's support SQL to some degree. They usually provide "a superset of a subset." In the course, we will try to stick to MS SQL feature.
- White space and case insensitive
- Traditional way to put commands in upper case letter but not required.
- SQL provides several types of commands:
    - Queries
    - DDL - Data Definition Language
        - Create/destroy/alter relations and views
        - Define integrity constraints
        - CREATE and ALTER are DDL
    - DML - Data Manipulation Language
        - Inert/delete/update records
        - Interact closely with integrity constraints

        - INSERT, UPDATE, DELETE, and MODIFY are DML

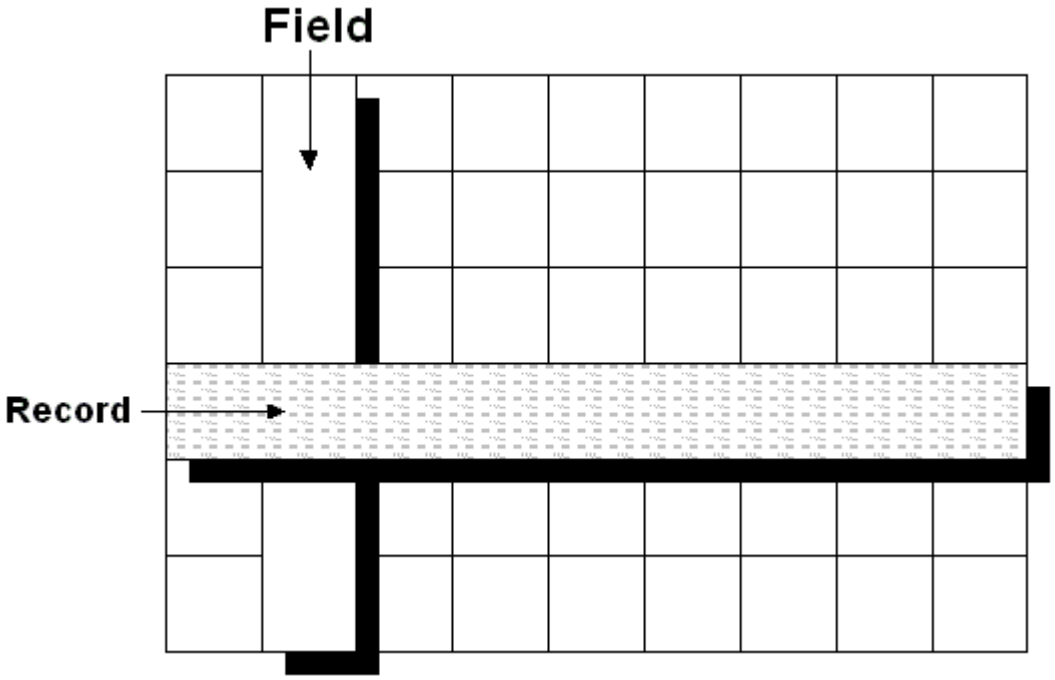
- DCL - Data Control Language (Control of access)
    - Controls your DML, and DDL Statements so that your data is protected and has consistency. COMMIT and ROLLBACK are DCL control statements.
    - Can grant/revoke the right to access and manipulate tables (relations/views)

❖ You will learn about these keywords SELECT, FROM, WHERE, DISTINCT, BETWEEN, IN, AND, OR, LIKE, ORDER BY.

Understanding Database Tables, Records, Fields

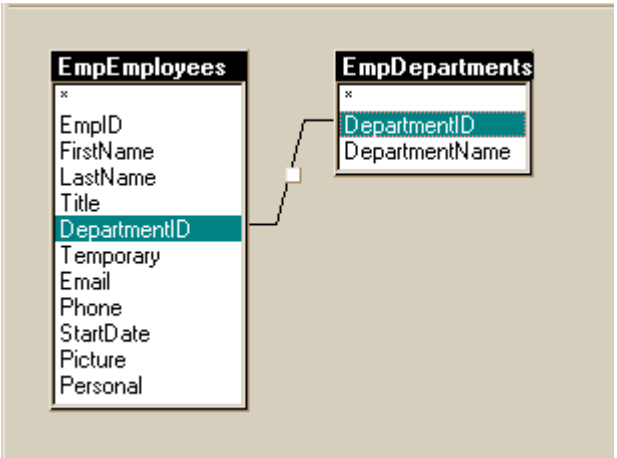
- Databases are organized in tables, which are collections of related items. For example, a table might contain the names, street addresses, and phone numbers of individuals.
- Think of a table as a grid of columns and rows. In this case, one column contains names, a second column contains street addresses, and the third column contains phone numbers.
- Each row constitutes one data record. In this case, each row is unique because it applies to one individual.
- Rows are also referred to as records.
- Columns are also referred to as fields.

Table



Note: You can organize data in multiple tables. This type of data structure is known as a relational database and is the type used for all but the simplest data sets

Relational Database



Because you have multiple departments for employees, but you would not store the information about the departments in every employee row for several reasons:

- The department information is the same for each employee in a given department; however, repeating the department information for each employee is redundant. Storing redundant data takes up more disk space.
- If the department information changes, you can update one occurrence. All references to that department are updated automatically.
- Storing multiple occurrences of the same data is rarely a good thing.
- Good relational database design separates application entities into their own tables.
- Key values from one table are often stored in a related table rather than repeating the information.
- The key value is used to join the data between the tables to return the complete set of data required.

From this basic description, a few database design rules emerge:

- Each record should contain a unique identifier, known as the primary key. This can be an employee ID, a part number, or a customer number. The primary key is typically the column used to maintain each record's unique identity among the tables in a relational database.
- After you define a column to contain a specific type of information, you must enter data in that column in a consistent way.
- To enter data consistently, you define a data type for the column, such as allowing only numeric values to be entered in the salary column.
- Assessing user needs and incorporating those needs in the database design is essential to a successful implementation. A well-designed database accommodates the changing data needs within an organization

Data Type

Type	From	To
bigint	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
int	-2,147,483,648	2,147,483,647
smallint	-32,768	32,767
tinyint	0	255
Bit	0	1
Decimal	-10^38 +1	10^38 –1
Numeric	-10^38 +1	10^38 –1
Money	-922,337,203,685,477.5808	+922,337,203,685,477.5807
smallmoney	-214,748.3648	+214,748.3647

Numeric and decimal are fixed precision and scale data types and are functionally equivalent.

Approximate numeric

Type	From	To
Float	-1.79E + 308	1.79E + 308
Real	-3.40E + 38	3.40E + 38

datetime and smalldatetime

Type	From	To
datetime (3.33 milliseconds accuracy)	Jan 1, 1753	Dec 31, 9999
smalldatetime (1 minute accuracy)	Jan 1, 1900	Jun 6, 2079

Character Strings

Type	Description
Char	Fixed-length non-Unicode character data with a maximum length of 8,000 characters.
varchar	Variable-length non-Unicode data with a maximum of 8,000 characters.
varchar(max)	Variable-length non-Unicode data with a maximum length of 2^{31} characters (SQL Server 2005 only).
Text	Variable-length non-Unicode data with a maximum length of 2,147,483,647 characters.

Unicode Character Strings

Type	Description
nchar	Fixed-length Unicode data with a maximum length of 4,000 characters.
nvarchar	Variable-length Unicode data with a maximum length of 4,000 characters.
nvarchar(max)	Variable-length Unicode data with a maximum length of 2^{30} characters (SQL Server 2005 only).
ntext	Variable-length Unicode data with a maximum length of 1,073,741,823 characters.

Binary Strings

Type	Description
Binary	Fixed-length binary data with a maximum length of 8,000 bytes.

varbinary	Variable-length binary data with a maximum length of 8,000 bytes.
varbinary(max)	Variable-length binary data with a maximum length of 2^{31} bytes (SQL Server 2005 only).
Image	Variable-length binary data with a maximum length of 2,147,483,647 bytes.

Other Data Types

- sql_variant: Stores values of various SQL Server-supported data types, except text, ntext, and timestamp.
- Timestamp: Stores a database-wide unique number that gets updated every time a row gets updated.
- uniqueidentifier: Stores a globally unique identifier (GUID).
- xml: Stores XML data. You can store xml instances in a column or a variable (SQL Server 2005 only).
- Cursor: A reference to a cursor.
- Table: Stores a result set for later processing.

SQL Select

Basic Syntax for SQL Select:

```
Select <attribute-list>
From <relation-list>
   [Where <condition>]
```

The **where** clause (optional) specifies which data values or rows will be returned or displayed, based on the criteria described after the keyword **where**.

Conditional selections used in the **where** clause:

= Equal

> Greater than

< Less than

>= Greater than or equal

<= Less than or equal

<> Not equal to

LIKE Pattern Matching

Note: Examples listed here will use Northwind database

```
SELECT
        EmployeeID,
        FirstName,
        LastName,
        City,
        Country
FROM employees
```

Result:

	EmployeeID	FirstName	LastName	City	Country
1	1	Nancy	Davolio	Seattle	USA
2	2	Andrew	Fuller	Tacoma	USA
3	3	Janet	Leverling	Kirkland	USA
4	4	Margaret	Peacock	Redmond	USA
5	5	Steven	Buchanan	London	UK
6	6	Michael	Suyama	London	UK
7	7	Robert	King	London	UK
8	8	Laura	Callahan	Seattle	USA
9	9	Anne	Dodsworth	London	UK

Multiple column names can be selected, as well as multiple table names. We can select all column names by using asterisk *.

```
SELECT * FROM Employees
```

Avoid using *: SELECT * FROM *table name* should not be used anywhere as a means to perform a SELECT operation on a table. Required columns should be mentioned explicitly in the select list. This technique results in reduced disk I/O and better performance.

Performance Tips: To make SQL Statements more readable, start each clause on a new line and indent when needed. Following is an example:

```
SELECT
        EmployeeID,
        FirstName,
        LastName,
        City,
        Country
FROM  Employees
WHERE Country = 'USA'
ORDER BY FirstName
```

Removing duplicate elements: By default, SQL will not remove duplicate elements in a query. For example, to select all distinct cities from Employees Table *Employees*,

```
SELECT DISTINCT city, country FROM Employees
```

Result:

	city	country
1	Kirkland	USA
2	London	UK
3	Redmond	USA
4	Seattle	USA
5	Tacoma	USA

Performance Tips: Use DISTINCT whenever absolutely necessary unnecessary use of DISTINCT reduces the server performance.

Selecting from range values: The "Where" part of a query may contain range condition with BETWEEN keyword

```
SELECT
        EmployeeID,
        FirstName,
        LastName,
        City,
        Country
FROM  Employees
WHERE EmployeeID BETWEEN 4 AND 8
```

Result:

	EmployeeID	FirstName	LastName	City	Country
1	4	Margaret	Peacock	Redmond	USA
2	5	Steven	Buchanan	London	UK
3	6	Michael	Suyama	London	UK
4	7	Robert	King	London	UK
5	8	Laura	Callahan	Seattle	USA

We can compare with group of elements using IN keyword: This query will select employees whose Id is 1,4,6,7 and 9.

```
SELECT
        EmployeeID,
        FirstName,
        LastName,
        City,
        Country
FROM  Employees
WHERE EmployeeID IN (1,4,6,7,9)
```

Result:

	EmployeeID	FirstName	LastName	City	Country
1	1	Nancy	Davolio	Seattle	USA
2	4	Margaret	Peacock	Redmond	USA
3	6	Michael	Suyama	London	UK
4	7	Robert	King	London	UK
5	9	Anne	lemonnew	London	UK

Combining more than one conditions: The "Where" part of a query may contain multiple conditions. For example select all employees who reside in USA and their name

contains Letter 'N'.

```
SELECT
        EmployeeID,
        FirstName,
        LastName,
        City,
        Country
FROM  Employees
WHERE Country = 'USA' AND FirstName LIKE '%N%'
```

Result:

	EmployeeID	FirstName	LastName	City	Country
1	1	Nancy	Davolio	Seattle	USA
2	2	Andrew	Fuller	Tacoma	USA
3	3	Janet	Leverling	Kirkland	USA

Pattern matching: The Like operator effects pattern matching. Here is a query which finds all employees whose first names begin with the letter "M".

```
SELECT
        EmployeeID,
        FirstName,
        LastName,
        City,
        Country
FROM  Employees
WHERE FirstName LIKE 'M%'
```

Result:

	EmployeeID	FirstName	LastName	City	Country
1	4	Margaret	Peacock	Redmond	USA
2	6	Michael	Suyama	London	UK

Performance Tips: Try to avoid wildcard characters at the beginning of a word while searching using the LIKE keyword, as that result in an index scan, which defeats the purpose of an index. The following statement results in an index scan, while the second statement results in an index seek:

```
SELECT LocationID FROM Locations WHERE Specialities LIKE '%pples'
SELECT LocationID FROM Locations WHERE Specialities LIKE 'A%s'
```

Ordering:

We can sort the result in ascending or descending order by using ASC and DESC keywords. The following query orders the employee names in ascending order i.e. A to Z.

```
SELECT
        EmployeeID,
        FirstName,
        LastName,
        City,
        Country
FROM  Employees
ORDER BY FirstName ASC
```

Result:

	EmployeeID	FirstName	LastName	City	Country
1	2	Andrew	Fuller	Tacoma	USA
2	9	Anne	Dodsworth	London	UK
3	3	Janet	Leverling	Kirkland	USA
4	8	Laura	Callahan	Seattle	USA
5	4	Margaret	Peacock	Redmond	USA
6	6	Michael	Suyama	London	UK
7	1	Nancy	Davolio	Seattle	USA
8	7	Robert	King	London	UK
9	5	Steven	Buchanan	London	UK

In addition to column name, we may also use column position (based on the SQL query) to indicate which column we want to apply the **ORDER BY** clause. The first column is 1, second column is 2, and so on. In the above example, we will achieve the same results by the following command:

```
SELECT
        EmployeeID,
        FirstName,
        LastName,
        City,
        Country
FROM  Employees
ORDER BY 2 ASC
```

Performance Tips: Avoid this method since when any column(s) are inserted/deleted then it does not give expected result.

Performance Tips: Use ORDER BY whenever absolutely necessary unnecessary use of ORDER BY reduces the server performance

Alias - Naming columns: Sometimes, it is essential to use aliases. This example retrieves the name of each employee, together with the First Name & Last Name

```
SELECT
        EmployeeID,
        FirstName + ' ' + LastName AS Emp_Name,
        City,
        Country
FROM  Employees
```

Result:

	EmployeeID	Emp_Name	City	Country
1	1	Nancy Davolio	Seattle	USA
2	2	Andrew Fuller	Tacoma	USA
3	3	Janet Leverling	Kirkland	USA
4	4	Margaret Peacock	Redmond	USA
5	5	Steven Buchanan	London	UK
6	6	Michael Suyama	London	UK
7	7	Robert King	London	UK
8	8	Laura Callahan	Seattle	USA
9	9	Anne lemonnew	London	UK
10	10	alex jones	NULL	sdfs

Exercise

1. Display the first name and city for everyone that's in the employee's table.
2. Display all columns for everyone except firstname in employee's table.
3. Display the first name, last name, and city for everyone that's not from UK.
4. Display all columns for everyone that is TitleOfCourtesy is 'Mr.'
5. Display the first and last names for everyone whose last name ends in an "an".
6. Display the Employees Firstname equal to Nancy or Anne or Robert.
7. Display all columns for everyone whose first name contains "a".
8. Write a query to display Employee table by remove a duplicate City names
9. Write a query to find all employee order by city in Descending order.
10. Write a query to find all products in the products table that have stock zero.
11. Write a query to list all products ID, Product Name, and Unit Price in the products table in descending order based on Unit price.
12. Write a query to list all products ID, Product Name, and Unit Price in the products table whose stock in range 10 to 20.
13. Write a query to list all Territories with Territorydescription starts with letter 'B'.
14. Write a query to list all countries from Suppliers table by removing the duplicate country names.
15. Write a query to list company name, contact address, & city from countries 'UK", "USA" and 'Japan' in suppliers table. (use IN keyword).

Chapter II

Select TOP

The TOP clause limits the number of rows returned in the result set.

TOP *n* [PERCENT]; *n* specifies how many rows are returned. If PERCENT is not specified, *n* is the number of rows to return. If PERCENT is specified, *n* is the percentage of the result set rows to return: To see what this means in practice, execute the following query against the northwind database:

```
SELECT TOP 5 ProductName,UnitPrice FROM products ORDER BY unitprice DESC
```

Result:

	ProductName	UnitPrice
1	Côte de Blaye	263.50
2	Thüringer Rostbratwurst	123.79
3	Mishi Kobe Niku	97.00
4	Sir Rodney's Marmalade	81.00
5	Carnarvon Tigers	62.50

```
SELECT TOP 5 PERCENT ProductName, UnitPrice
FROM products ORDER BY unitprice DESC
```

Result:

	ProductName	UnitPrice
1	Côte de Blaye	263.50
2	Thüringer Rostbratwurst	123.79
3	Mishi Kobe Niku	97.00
4	Sir Rodney's Marmalade	81.00

Select INTO

Creates a new table and inserts the resulting rows from the query into it. Example select from employee table into temp_emp table.

```
SELECT LastName,FirstName,City,Country INTO Temp_emp FROM Employees
```

It will create new table and inserts selected columns into it. i.e. select from Temp_emp it gives the following result

```
SELECT LastName,FirstName,City,Country  FROM Temp_emp
```

Result:

	LastName	FirstName	City	Country
1	Davolio	Nancy	Seattle	USA
2	Fuller	Andrew	Tacoma	USA
3	Leverling	Janet	Kirkland	USA
4	Peacock	Margaret	Redmond	USA
5	Buchanan	Steven	London	UK
6	Suyama	Michael	London	UK
7	King	Robert	London	UK
8	Callahan	Laura	Seattle	USA
9	lemonnew	Anne	London	UK
10	jones	alex	NULL	sdfs

SQL JOINS

A join combines records from two tables in a relational database and results in a new (temporary) table, also called joined table. In the Structured Query Language (SQL), there are two types of joins: inner and outer.

We will use these two tables to show the join examples

Table: **Department**

	DepartmentName	DepartmentID
1	Sales	1
2	Engineering	2
3	Clerical	3
4	Marketing	4

Table: Employee

	LastName	DepartmentID
1	Rafferty	1
2	Jones	2
3	Steinberg	2
4	Robinson	3
5	Smith	3
6	Jasper	7

INNER JOIN

An inner join essentially finds the intersection between the two tables. This is the most common type of join used, and is considered the default join type. Inner joins return all rows from multiple tables where the join condition is met.

```
SELECT *
FROM Employee  INNER JOIN Department
ON Employee.DepartmentID = Department.DepartmentID
```

Result:

	LastName	DepartmentID	DepartmentName	DepartmentID
1	Rafferty	1	Sales	1
2	Jones	2	Engineering	2
3	Steinberg	2	Engineering	2
4	Robinson	3	Clerical	3
5	Smith	3	Clerical	3

Notice that employee 'Jasper' and department Marketing do not appear. Neither of these records have accompanying rows in their associative tables, and are thus omitted from the inner join result.

Example of an implicit inner join:

```
SELECT *
FROM Employee, Department
WHERE Employee.DepartmentID = Department.DepartmentID
```

CROSS JOIN

Cross join is the foundation upon which inner joins are built. A cross join returns the cartesian product of the sets of rows from the joined tables. Thus, it is an inner join where the join condition always evaluates to True.

Example

```
SELECT *
FROM Employee CROSS JOIN department
```

Result:

	LastName	DepartmentID	DepartmentName	DepartmentID
1	Rafferty	1	Sales	1
2	Jones	2	Sales	1
3	Steinberg	2	Sales	1
4	Robinson	3	Sales	1
5	Smith	3	Sales	1
6	Jasper	7	Sales	1
7	Rafferty	1	Engineering	2
8	Jones	2	Engineering	2
9	Steinberg	2	Engineering	2
10	Robinson	3	Engineering	2
11	Smith	3	Engineering	2
12	Jasper	7	Engineering	2
13	Rafferty	1	Clerical	3
14	Jones	2	Clerical	3
15	Steinberg	2	Clerical	3
16	Robinson	3	Clerical	3
17	Smith	3	Clerical	3
18	Jasper	7	Clerical	3
19	Rafferty	1	Marketing	4
20	Jones	2	Marketing	4
21	Steinberg	2	Marketing	4
22	Robinson	3	Marketing	4
23	Smith	3	Marketing	4
24	Jasper	7	Marketing	4

As you can see the cross join does not apply any predicate when matching records for the joined table. These joins are almost never used, except to generate all possible combinations of records from tables that do not share a common element. Still, the results of a cross joins can be further filtered.

OUTER JOIN

Outer joins are subdivided further into left outer joins, right outer joins, and full outer joins.

LEFT OUTER JOIN

A left outer join is very different from an inner join. Instead of limiting results to those in both tables, it limits results to those in the "left" table (A). This means that if the ON clause matches 0 records in B, a row in the result will still be returned — but with NULL in each column from B.

A left outer join returns all the values from left table + matched values from right table (or NULL in case of no matching value).

For example, this allows us to find the employee's departments, but still show the employee even when their department is NULL or does not exist (contrary to the inner join example above, where employees in non-existent departments were ignored).

Examples of a left outer join:

```
SELECT  *
FROM Employee LEFT OUTER JOIN Department
ON Employee.DepartmentID = Department.DepartmentID
```

Result:

	LastName	DepartmentID	DepartmentName	DepartmentID
1	Rafferty	1	Sales	1
2	Jones	2	Engineering	2
3	Steinberg	2	Engineering	2
4	Robinson	3	Clerical	3
5	Smith	3	Clerical	3
6	Jasper	7	NULL	NULL

Notice that for 'Jasper' DepartmentName and DepartmentId are NULL.

RIGHT OUTER JOIN

A right outer join is much like a left outer join, except that the tables are reversed. Every record from the right side, B, will be returned, and NULL will be returned for columns from A for those rows that have no matching record in A.

A right outer join returns all the values from right table + matched values from left table (or NULL in case of no matching value).

Example right outer join:

```
SELECT *
FROM Employee RIGHT OUTER JOIN Department
ON Employee.DepartmentID = Department.DepartmentID
```

Result:

	LastName	DepartmentID	DepartmentName	DepartmentID
1	Rafferty	1	Sales	1
2	Jones	2	Engineering	2
3	Steinberg	2	Engineering	2
4	Robinson	3	Clerical	3
5	Smith	3	Clerical	3
6	NULL	NULL	Marketing	4

Notice that LastName and DepartmentID are NULL for 'Marketing' Department

FULL OUTER JOIN

A full outer join combines the results of both left and right outer joins. These joins will show records from both tables, and fill in NULLs for missing matches on either side.

Some database systems do not support this functionality, but it can be emulated through the use of left and right outer joins and unions (see below).

Example full outer join:

```
SELECT *
FROM Employee FULL OUTER JOIN Department
ON Employee.DepartmentID = Department.DepartmentID
```

Result:

	LastName	DepartmentID	DepartmentName	DepartmentID
1	Rafferty	1	Sales	1
2	Jones	2	Engineering	2
3	Steinberg	2	Engineering	2
4	Robinson	3	Clerical	3
5	Smith	3	Clerical	3
6	Jasper	7	NULL	NULL
7	NULL	NULL	Marketing	4

Notice the NULL Values for **'Jasper'** LastName and **'Marketing'** DepartmentName.

Exercise

Note: Use Northwind Database

1. List products with top 10 Units On Order.
2. List all columns with top 20 highest freight from orders table.
3. Write a query to get top 20 records from orders table and insert into Temp_Order Table.
4. Write a query to get 20 percent of total rows from orders table.
5. Write a query to insert Territory into Temp_Territory a temporary table with their Region Description (hint: use Territories and region tables).
6. List out all Territory Description with their Region Description using INNER JOIN (hint: use Territories and region tables)
7. Write a query to list all product name with Supplier contact name and city using INNER JOIN (Hint: use Supplier and Products tables)
8. Write a query to list all regions with their territory description even if there is no territory for the region than also should be listed.
9. Write a query to list all product name with Supplier contact name, Address, UnitPrice and city using LEFT OUTER JOIN (Hint: use Supplier and Products tables)
10. Write a query to list all product name with Supplier contact name, Address, UnitPrice and city using RIGHT OUTER JOIN (Hint: use Supplier and Products tables)
11. Write a query to list all product name with Supplier contact name, Address, UnitPrice and city using FULL OUTER JOIN (Hint: use Supplier and Products tables)

Chapter III

Table Manipulations

Create table

The SQL simple syntax for **CREATE TABLE** is

```
CREATE TABLE "table_name"
("column 1" "data_type_for_column_1",
 "column 2" "data_type_for_column_2",
 "column 2" "data_type_for_column_2",
 ... )
```

So, if we are to create the customer table specified as above, we would type in

```
CREATE TABLE Customer
( First_Name VARCHAR(50),
  Last_Name VARCHAR(50),
  Address VARCHAR(50),
  City VARCHAR(50),
  Country VARCHAR(25),
  Birth_Date DATETIME)
```

Sometimes, we want to provide a default value for each column. A default value is used when you do not specify a column's value when inserting data into the table. To specify a default value, add "Default [value]" after the data type declaration. In the above example, if we want to default column "Address" to "Unknown" and City to "Bangalore", we would type in

```
CREATE TABLE Customer
( First_Name VARCHAR(50),
  Last_Name VARCHAR(50),
  Address VARCHAR(50) DEFAULT 'Unknown',
  City VARCHAR(50) DEFAULT 'Bangalore',
  Country VARCHAR(25),
  Birth_Date DATETIME)
```

You can also limit the type of information a table / a column can hold. This is done through the **CONSTRAINT** keyword, which is discussed next.

Constraint(CONDITION)

You can place constraints to limit the type of data that can go into a table. Such constraints can be specified when the table when the table is first created via the **CREATE TABLE** statement, or after the table is already created via the **ALTER TABLE** statement.

Common types of constraints include the following:

- **NOT NULL**
- **UNIQUE**
- **CHECK**
- **Primary Key**
- **Foreign Key**

Each is described in detail next.

NOT NULL

By default, a column can hold NULL. If you not want to allow NULL value in a column, you will want to place a constraint on this column specifying that NULL is now not an allowable value.

For example, in the following statement,

```
CREATE TABLE Customer
( SID INT NOT NULL,
  Last_Name VARCHAR (30) NOT NULL,
  First_Name VARCHAR (30))
```

Columns "SID" and "Last_Name" cannot include NULL, while "First_Name" can include NULL.

UNIQUE

The UNIQUE constraint ensures that all values in a column are distinct.

For example, in the following statement,

```
CREATE TABLE Customer
( SID INT UNIQUE,
  Last_Name VARCHAR (30),
  First_Name VARCHAR (30))
```

Column "SID" cannot include duplicate values, while such constraint does not hold for columns "Last_Name" and "First_Name".

Please note that a column that is specified as a primary key must also be unique. At the same time, a column that's unique may or may not be a primary key.

Primary Key

A primary key is used to uniquely identify each row in a table. It can either be part of the actual record itself, or it can be an artificial field (one that has nothing to do with the actual record). A primary key can consist of one or more fields on a table. When multiple fields are used as a primary key, they are called a composite key.

Primary keys can be specified either when the table is created (using **CREATE TABLE**) or by changing the existing table structure (using **ALTER TABLE**).

Below are examples for specifying a primary key when creating a table:

```
CREATE TABLE Customer
 SID INT PRIMARY KEY,
 Last_Name VARCHAR (30),
 First_Name VARCHAR (30))
```

Below are examples for specifying a primary key by altering a table:

```
ALTER TABLE Customer ADD PRIMARY KEY (SID)
```

Note: Before using the ALTER TABLE command to add a primary key, you'll need to make sure that the field is defined as 'NOT NULL' -- in other words, NULL cannot be an accepted value for that field.

Foreign Key

A foreign key is a field (or fields) that points to the primary key of another table. The purpose of the foreign key is to ensure referential integrity of the data. In other words, only values that are supposed to appear in the database are permitted.

For example, say we have two tables, a CUSTOMER table that includes all customer data, and an ORDERS table that includes all customer orders. The constraint here is that all orders must be associated with a customer that is already in the CUSTOMER table. In this case, we will place a foreign key on the ORDERS table and have it relate to the primary key of the CUSTOMER table. This way, we can ensure that all orders in the ORDERS table are related to a customer in the CUSTOMER table. In other words, the ORDERS table cannot contain information on a customer that is not in the CUSTOMER table.

The structure of these two tables will be as follows:

Table ***CUSTOMER***

column name	characteristic
SID	Primary Key
Last_Name	
First_Name	

Table ***ORDERS***

column name	characteristic
Order_ID	Primary Key
Order_Date	
Customer_SID	Foreign Key
Amount	

In the above example, the Customer_SID column in the ORDERS table is a foreign key pointing to the SID column in the CUSTOMER table.

Below we show examples of how to specify the foreign key when creating the ORDERS table:

```
CREATE TABLE ORDERS
( Order_ID INT PRIMARY KEY,
  Order_Date DATETIME,
  Customer_SID INT REFERENCES CUSTOMER(SID),
  Amount FLOAT)
```

Below are examples for specifying a foreign key by altering a table. This assumes that the ORDERS table has been created, and the foreign key has not yet been put in:

```
ALTER TABLE Orders
ADD FOREIGN KEY (Customer_SID) REFERENCES Customer(SID)
```

Alter Table Statement

Once a table is created in the database, there are many occasions where one may wish to change the structure of the table. Using the ALTER table command, you can make the following types of change to an existing table:

- Change the data type or NULL property of a single column.
- Add one or more new columns, with or without defining constraints for those column’s
- Add one or more constraints.
- Drop one or more constraints.
- Drop one or more columns.
- Enable or disable one more constraints (only applies to CHECK and FOREIGN KEY constraints).
- Enable or disable one or more triggers.

Let's run through examples for each one of the above, using the "customer" table created in the CREATE TABLE section:

Table *customer*

Column Name	Data Type
First_Name	VARCHAR (50)
Last_Name	VARCHAR (50)
Address	VARCHAR (50)
City	VARCHAR (50)
Country	VARCHAR (25)
Birth_Date	DATETIME

First, we want to add a column called "Gender" to this table. To do this, we key in:

```
ALTER TABLE Customer ADD Gender BIT
```

Resulting table structure:

Column Name	Data Type
First_Name	VARCHAR (50)
Last_Name	VARCHAR (50)
Address	VARCHAR (50)
City	VARCHAR (50)
Country	VARCHAR (25)
Birth_Date	DATETIME
Gender	BIT

Then, we want to change the data type for "Addrress" to VARCHAR(30). To do this, we key in,

```
ALTER TABLE Customer ALTER COLUMN Address VARCHAR(30)
```

Resulting table structure:

Column Name	Data Type
First_Name	VARCHAR (50)
Last_Name	VARCHAR (50)
Address	VARCHAR (30)
City	VARCHAR (50)
Country	VARCHAR (25)
Birth_Date	DATETIME
Gender	BIT

Finally, we want to drop the column "Gender". To do this, we key in,

```
ALTER TABLE Customer DROP COLUMN Gender
```

Resulting table structure:

Column Name	Data Type
First_Name	VARCHAR(50)
Last_Name	VARCHAR (50)
Address	VARCHAR (50)
City	VARCHAR (50)
Country	VARCHAR (25)
Birth_Date	DATETIME

Drop Table Statement

Sometimes we may decide that we need to get rid of a table in the database for some reason. In fact, it would be problematic if we cannot do so because this could create a maintenance nightmare for the DBA's. Fortunately, SQL allows us to do it, as we can use the DROP TABLE command. The syntax for DROP TABLE is

DROP TABLE "table_name"

So, if we wanted to drop the table called customer that we created in the CREATE TABLE section, we simply type

DROP TABLE Customer

Caution: Use this command carefully; you may lose important data.

Insert Into Statement

In the previous sections, we have seen how to retrieve information from tables. But how do these rows of data get into these tables in the first place? This is what this section, covering the INSERT statement, and next section, covering the UPDATE statement, is about.

In SQL, there are essentially basically two ways to INSERT data into a table: One is to insert it one row at a time, the other is to insert multiple rows at a time. Let's first look at how we may INSERT data one row at a time:

The syntax for inserting data into a table one row at a time is as follows:

```
INSERT INTO "table_name" ("column1", "column2", ...)
VALUES ('value1', 'value2', ...)
```

Assuming that we have a table that has the following structure,

Table *Store_Information*

Column Name	Data Type
Store_name	VARCHAR(50)
Sales	FLOAT
Date	DATETIME

And now we wish to insert one additional row into the table representing the sales data for Los Angeles on January 10, 1999. On that day, this store had $900 in sales. We will hence use the following SQL script:

```
INSERT INTO Store_Information (store_name, Sales, Date)
VALUES ('Los Angeles', 900, 'Jan-10-1999')
```

The second type of INSERT INTO allows us to insert multiple rows into a table. Unlike the previous example, where we insert a single row by specifying its values for all columns, we now use a SELECT statement to specify the data that we want to insert into the table. If you are thinking whether this means that you are using information from another table, you are correct. The syntax is as follows:

```
INSERT INTO "table1" ("column1", "column2", ...)
SELECT "column3", "column4", ...
FROM "table2"
```

Note that this is the simplest form. The entire statement can easily contain WHERE, GROUP BY, and HAVING clauses, as well as table joins and aliases.

So for example, if we wish to have a table, *Store_InformationCopy*, that contains information from *Store_Information*, we'll type in:

```
INSERT INTO Store_InformationCopy (store_name, Sales, Date)
SELECT store_name, Sales, Date
FROM Sales_Information
```

Update Statement

Once there's data in the table, we might find that there is a need to modify the data. To do so, we can use the UPDATE command. The syntax for this is

```
UPDATE "table_name"
SET "column_1" = [new value]
WHERE {condition}
```

For example, say we currently have a table as below:

Table *Store_Information*

store_name	Sales	Date
Los Angeles	$1500	Jan-05-1999
San Diego	$250	Jan-07-1999
Los Angeles	$300	Jan-08-1999
Boston	$700	Jan-08-1999

And we notice that the sales for Los Angeles on 01/08/1999 are actually $500 instead of $300, and that particular entry needs to be updated. To do so, we use the following SQL:

```
UPDATE Store_Information
SET Sales = 500
WHERE store_name = 'Los Angeles'
AND Date = 'Jan-08-1999'
```

The resulting table would look like

Table *Store_Information*

store_name	Sales	Date
Los Angeles	$1500	Jan-05-1999
San Diego	$250	Jan-07-1999
Los Angeles	$500	Jan-08-1999
Boston	$700	Jan-08-1999

In this case, there is only one row that satisfies the condition in the WHERE clause. If there are multiple rows that satisfy the condition, all of them will be modified.

It is also possible to UPDATE multiple columns at the same time. The syntax in this case would look like the following:

```
UPDATE "table_name"
SET column_1 = [value1], column_2 = [value2]
WHERE {condition}
```

Delete From Statement

Sometimes we may wish to get rid of records from a table. To do so, we can use the DELETE FROM command. The syntax for this is

```
DELETE FROM "table_name"
WHERE {condition}
```

It is easiest to use an example. Say we currently have a table as below:

Table *Store_Information*

store_name	Sales	Date
Los Angeles	$1500	Jan-05-1999
San Diego	$250	Jan-07-1999
Los Angeles	$300	Jan-08-1999
Boston	$700	Jan-08-1999

And we decide not to keep any information on Los Angeles in this table. To accomplish this, we type the following SQL:

```
DELETE FROM Store_Information
WHERE store_name = 'Los Angeles'
```

Now the content of table would look like,

Table *Store_Information*

store_name	Sales	Date
San Diego	$250	Jan-07-1999
Boston	$700	Jan-08-1999

Truncate Table Statement

Sometimes we wish to get rid of all the data in a table. One way of doing this is with DROP TABLE, which we saw in the last section. But what if we wish to simply get rid of the data but not the table itself? For this, we can use the TRUNCATE TABLE command. The syntax for TRUNCATE TABLE is

```
TRUNCATE TABLE "table_name"
```

So, if we wanted to truncate the table called customer that we created in SQL CREATE TABLE, we simply type,

```
TRUNCATE TABLE Customer
```

Difference between Delete and Truncate

- Truncate command is faster than the delete command.
- We can rollback transaction in delete statement but we cant in case of Truncate
- It is not possible to remove some rows using truncate but is possible with delete.
- We can remove rows based on condition in delete in case of truncate it is not possible.
- After removing the rows using delete the identity seed starts from previous numbers (e.g. previously it has 25 rows then after delete if we insert new row it gives next ID as 26). But in case of truncate it starts from its initial number. (e.g. previously it has 25 rows then after truncate if we insert new row it gives next ID as 1).

Exercise

1. Write a Query to create table with following fields.
   EmployeeID, FirstName, LastName, Salary, and City
2. Write a Query to create above table with EmployeeID as primary key and 'Bangalore' as default city.
3. Write a Query to create above table with EmpID as Not Null, Firstname, Lastname as varchar.
4. Write a Query to create a table with EmpID as INT, Firstname, Lastname as varchar. With the Column" EmpID"must only include integers greater than 0.
5. Write a Query to create table with following data types:
   Int, smallint, numeric, varchar and date
6. Write a query to alter the table employee to add new field DOB.
7. Write a query to alter the table employee to modify the existing field "Firstname" datatype "Varchar" into "Char" datatype.
8. Write a query to drop the column DOB in "employee" table.
9. Write a query to insert Firstname of "Robert" and Lastname of "Calne" in "employee" table.
10. Create a new table "Emp" with Fields EmployeeID, Firstname, Lastname and insert all records from 'employee' table into "Emp" table
11. Write a query to update the Title into the "employee" from "Sales Representative" into "Sales Executive"
12. What happens if you execute a **DELETE** statement without a **WHERE** clause?
13. Create a books table containing the title ID, title, publisher ID, and price of books. Title ID as Primary key. Use this table for the remaining exercises.
14. Write a query to insert the title, publisher ID, and price into the table.
15. Write a query to update the price of all books decreased by 20 percent.

16. Write a query to decrease the price of all books whose price is more than 100 by 10 percent.
17. Delete all books, which are having price less than 10.

Chapter IV

View

Before we create our first view, we should know some rules about the select statement. It cannot:

- Include ORDER BY clause; unless there is also a TOP clause (remember that a view is nothing else but a virtual table, why would we want to order it?)
- Include the INTO keyword.
- Include COMPUTE or COMPUTE BY clauses.
- Reference a temporary table or a table variable.

I thought it would be a good idea to create the vwProductSales I used earlier as an example, and that is also the one you can see in the picture in the Introduction section. So, start Query Analyzer, and logon as a user with permissions to create a view. Then run this:

```
USE Northwind
GO
CREATE VIEW vwProductSales
AS
	SELECT Products.ProductName, SUM(OD.UnitPrice*OD.Quantity*(1-
OD.Discount)) AS Total
	FROM [Order Details] AS OD
	INNER JOIN Products ON OD.ProductID = Products.ProductID
	GROUP BY Products.ProductName
GO
```

Result:

```
Command(s) completed successfully.
```

As you can see, it's not difficult at all! It's basically a SELECT statement.

Now, say that we by some reason want this ordered by the Total column. Again, think about a view as a virtual table, so there is actually no reason to do it (do you usually sort your data in your tables when you insert data? Me neither!), but just for fun:

```
USE Northwind
GO
CREATE VIEW vwProductSalesSorted
AS
	SELECT Products.ProductName, SUM(OD.UnitPrice*OD.Quantity*(1-
OD.Discount)) AS Total
	FROM [Order Details] AS OD
	INNER JOIN Products ON OD.ProductID = Products.ProductID
	GROUP BY Products.ProductName
	ORDER BY Total DESC
GO
```

Result:

```
Msg 1033, Level 15, State 1, Procedure vwProductSalesSorted, Line 7
The ORDER BY clause is invalid in views, inline functions, derived tables, and
subqueries, unless TOP is also specified.
```

As you can see by the output, we were not allowed to do this. On the other hand, if we were only interested in the best selling products, we could write this:

```
USE Northwind
GO
CREATE VIEW vwProductSalesTop10
AS
	SELECT TOP 10 Products.ProductName, SUM(OD.UnitPrice*OD.Quantity*(1-
OD.Discount)) AS Total
	FROM [Order Details] AS OD
	INNER JOIN Products ON OD.ProductID = Products.ProductID
	GROUP BY Products.ProductName
	ORDER BY Total DESC
GO
```

Result:

Command(s) completed successfully.

As you can see, this worked. We did not violate the rules of a view, and this was also one of the advantages of using a view, to select only the data we were interested in.

you can see, there it is, our view! But of course, Microsoft does not force us to write all this to get information about a view (or a rule, a default, an unencrypted stored procedure, user-defined function, trigger). The system stored procedure sp_helptext does this for us.

Using views

You can use a view to retrieve data, or update data. A view can be used whenever a table could be used, for retrieving data. For updating data, this may not always be true, and it depends on the view.

Retrieve data

We have already seen several examples of retrieving data through a view. I will now compare two queries, one where we use a view, and one without.

```
SELECT TOP 1 *
FROM vwProductSales
ORDER BY Total DESC
```

Result:

	ProductName	Total
1	Côte de Blaye	141396.735229492

```
SELECT
        TOP 1 Products.ProductName,
        SUM(OD.UnitPrice*OD.Quantity*(1-OD.Discount)) AS Total
FROM [Order Details] AS OD
INNER JOIN Products ON OD.ProductID = Products.ProductID
GROUP BY Products.ProductName
ORDER BY Total DESC
```

Result:

	ProductName	Total
1	Côte de Blaye	141396.735229492

Update data

There are some restrictions for updating data through a view.

- A view cannot modify more than one table. So if a view is based on two or more tables, and you try to run a DELETE statement, it will fail. If you run an UPDATE or INSERT statement, all columns referenced in the statement must belong to the same table.
- It’s not possible to update, insert or delete data in a view with a DISTINCT clause.
- You cannot update, insert or delete data in a view that is using GROUP BY.
- It’s not possible to update, insert or delete data in a view that contains calculated columns.

Also be aware of columns that cannot contain NULL. If it has no default value, and is not referenced in the view, you will not be able to update the column.

Drop views

Deleting a view is not complicated. It’s as easy as:

```
DROP VIEW vwProductSalesTop10
```

Alter views

Alter views is as simple as deleting them. We can try to remove the encryption of the view vwProductSalesTop10

```
ALTER VIEW vwProductSalesTop10
AS
	SELECT Products.ProductName, SUM(OD.UnitPrice*OD.Quantity*(1-
OD.Discount)) AS Total
	FROM [Order Details] AS OD
	INNER JOIN Products ON OD.ProductID = Products.ProductID
	GROUP BY Products.ProductName
```

Built In functions:

While your primary job as a database developer consists of creating lists, probably your second most important job is to assist your users with the various assignments they must perform on your application. One way you can assist is to use functions that perform otherwise complex tasks. We introduced and described functions in the previous sections. To assist your development with the different tasks of a database, You just need to be aware of these functions, their syntax, and the results they produce.

To help you identify the functions you can use, they are categorized by their types and probably their usefulness.

Conversion Functions

Casting a Value

To assist with conversion, you can use either the CAST () or the CONVERT () function. The syntax of the CAST () function is:

Converting each string is converted to a Decimal type:

```
DECLARE @StrSalary Varchar(10),
        @StrHours  Varchar(6),
        @WeeklySalary Decimal(6,2)
SET @StrSalary = '22.18';
SET @StrHours  = '38.50';
SET @WeeklySalary = CAST(@StrSalary As Decimal(6,2)) *
                    CAST(@StrHours As Decimal(6,2));
SELECT @WeeklySalary;
```

Converting a Value

Like CAST (), the CONVERT () function is used to convert a value. Unlike CAST (), CONVERT can be used to convert a value its original type into a non-similar type. For example, you can use CONVERT to cast a number into a string and vice-versa. Here is an example:

```
-- Square Calculation

DECLARE @Side As Decimal(10,3),
        @Perimeter As Decimal(10,3),
        @Area As Decimal(10,3);

SET     @Side = 4;
SET     @Perimeter = @Side * 4;
SET     @Area = @Side * @Side;
PRINT 'Square Characteristics';
PRINT '-----------------------';
PRINT 'Side      = ' + CONVERT(varchar, @Side);
PRINT 'Perimeter = ' + CONVERT(varchar, @Perimeter);
PRINT 'Area      = ' + CONVERT(varchar, @Area);
GO
```

Result:

Square Characteristics

Side   = 48.126

Perimeter = 192.504

Area   = 2316.112

String Based Functions

The Length of a String

To get the length of a string, you can use the LEN () function. This function takes one argument as the string to be considered. It returns the number of characters in the string. Here is an example:

```
SELECT LEN('Welcome') Length
```

Result:

	Length
1	7

String Conversions: Lowercase

If you want to convert all of characters to lowercase, you can use the LOWER () function.

```
SELECT LOWER('WELCOME') AS LowerCase
```

Result:

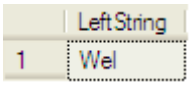
	LowerCase
1	welcome

Sub-Strings: The Starting Characters of a String

This function takes two arguments. The first argument specifies the original string. The second argument specifies the number of characters from the most-left that will constitute the sub-string.

```
SELECT LEFT('Welcome',3) AS LeftString
```

Result:



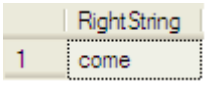
	LeftString
1	Wel

Sub-Strings: The Ending Characters of a String

Instead of the starting characters of a string, you may want to create a string using the most-right characters of an existing string. Transact-SQL provides the RIGHT() function. This function takes two arguments. The first argument specifies the original string. The second argument specifies the number of characters from the most-right that will constitute the sub-string.

```
SELECT RIGHT('Welcome',4) AS RightString
```

Result:



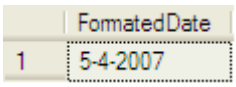
	RightString
1	come

Sub-Strings: Replacing Occurrences in a String

To replace one character or a sub-string from a string, you can use the REPLACE() function.

```
SELECT REPLACE('5/4/2007','/','-') AS FormatedDate
```

Result:

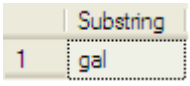


	FormatedDate
1	5-4-2007

Substring: Getting Middle string from a string: The Substring function in SQL is used to grab a portion of the stored data.

```
SELECT SUBSTRING('Bangalore',4,3) Substring
```

***Result*:**



	Substring
1	gal

```
SELECT SUBSTRING('Bangalore',4,8) Substring
```

Result:

	Substring
1	galore

String Conversions: Converting From Integer to ASCII

If you have a string, to get the ASCII code of its leftmost character, you can use the ASCII() function. This function takes as argument as string and returns the ASCII code of the first (the left) character of the string. Here is an example:

```
SELECT ASCII('A') AsciiCode
```

Result:

	AsciiCode
1	65

String Conversions: Converting From ASCII to Integer

If you have the ASCII code of a character and want to find its actual character, you can use the CHAR () function. This function takes as argument a numeric value as an integer. Upon conversion, the function returns the ASCII equivalent of that number.

```
SELECT CHAR('66') AS Character
```

Result:

	Character
1	B

Aggregate Functions

Since we have started dealing with numbers, the next natural question to ask is if it is possible to do math on those numbers, such as summing them up or taking their average. The answer is yes! SQL has several arithmetic functions and some of them are:

AVG()

COUNT()

MAX()

MIN()

SUM() etc...

Here is a query which gives sum of all sales from the product table.

```
SELECT SUM(UnitsOnOrder) FROM Products
```

Result:

	(No column name)
1	780

780 represent the sum of all Units on Order entries.

Grouping aggregated results: This query will find total stock for each category.

```
SELECT CategoryID,SUM(UnitsInStock) AS UnitsInStock
FROM PRODUCTS
GROUP BY CategoryID
```

Result:

	CategoryID	UnitsInStock
1	1	559
2	2	507
3	3	386
4	4	393
5	5	308
6	6	165
7	7	100
8	8	701

The Having operation- Grouping aggregation with conditions: This is similar to the previous example, with condition that UnitsInStock > 500.

```
SELECT CategoryID,SUM(UnitsInStock) AS UnitsInStock
FROM PRODUCTS
GROUP BY CategoryID
HAVING SUM(UnitsInStock) > 500
```

Result:

	CategoryID	UnitsInStock
1	1	559
2	2	507
3	8	701

Counting: The following query counts the number of store entries

```
SELECT COUNT(UnitsOnOrder)
FROM Products
WHERE UnitsOnOrder > 0
```

Result:

	(No column name)
1	17

COUNT and **DISTINCT** can be used together in a statement to fetch the number of distinct entries in a table. For example, if we want to find out the number of distinct stores, we'd type,

```
SELECT COUNT(DISTINCT CategoryID)
FROM Products
```

Result

	(No column name)
1	8

Further Comments on the *Group By* and *Having* Directives..When using these directives, every field in the *Select* clause must be either an attribute which is used in the *Group By* directive, or an aggregation. Thus, the following query is not legal, even though it makes perfect sense.

```
SELECT CategoryID,UnitsInStock AS UnitsInStock
FROM PRODUCTS GROUP BY CategoryID
HAVING UnitsInStock > 30
```

The following query, which has exactly the same semantics, is legal.

```
SELECT CategoryID,AVG(UnitsInStock) AS UnitsInStock
FROM PRODUCTS GROUP BY CategoryID
HAVING AVG(UnitsInStock) > 30
```

Mathematical Functions

The following scalar functions perform a calculation, usually based on input values that are provided as arguments, and return a numeric value. All mathematical functions, except for RAND, are deterministic functions. This means they return the same results each time they are called with a specific set of input values. RAND is deterministic only when a seed parameter is specified.

Mathematical Functions

Function	Results
ABS(numeric_expr)	Absolute value
ACOS, ASIN, ATAN (float_expr)	Angle in radians whose cosine, sine, or tangent is a floating-point value
ATN2(float_expr1 , float_expr2)	Returns the angle in radians whose tangent is between float1 and float2
COS, SIN, COT, TAN (float_expr)	Cosine, sine, or tangent of the angle (in radians)
CEILING(numeric_expr)	Smallest integer greater than or equal to specified value
DEGREES(numeric_expr)	Conversion from radians to degrees
EXP(float_expr)	Exponential value of specified value
FLOOR(numeric_expr)	Largest integer less than or equal to specified value
LOG(float_expr)	Natural log
LOG10(float_expr)	Base-10 log
PI()	Constant 3.141592653589793
POWER (numeric_expr , y)	Value of *numeric_expr* to the power of *y*
RADIANS(numeric_expr)	Conversion from degrees to radians

RAND([seed])	Random float number between 0 and 1
ROUND (numeric_expr,length)	*Numeric_exp* rounded to the specified length, length in an integer value
SIGN(numeric_expr)	Positive, negative, or zero
SQUARE(float_expr)	Squares the specified value
SQRT(float_expr)	Square root of specified value

Note: When using mathematical functions with monetary data types, you should always precede the data type with a dollar sign ($). Otherwise the value is treated as a numeric with a scale of 4.

Arithmetic functions, such as ABS, CEILING, DEGREES, FLOOR, POWER, RADIANS, and SIGN, return a value having the same data type as the input value. Trigonometric and other functions, including EXP, LOG, LOG10, SQUARE, and SQRT, cast their input values to float and return a float value

Mathematical Functions and Results

Statement	***Result***
SELECT SQRT(9)	3.0
SELECT ROUND(1234.56, 0)	1235
SELECT ROUND(1234.56, 1)	1234.60
SELECT ROUND($1234.56, 1)	1,234.60
SELECT ROUND(1234.56, 1)	256.0
SELECT FLOOR(1332.39)	1332
SELECT ABS(-365)	365

Date Functions

You can manipulate datetime values with date functions. Date functions can be used in the *column_list,* the WHERE clause, or wherever an expression can be used. Use the following syntax for date functions:

```
SELECT date_function (parmeters)
```

You must enclose datetime values passed as parameters between single quotation marks or double quotation marks. Some functions take a parameter known as a datepart. Table 11.6 lists the datepart values and their abbreviations.

datepart Values

datepart	***Abbreviation***	***Values***
day	dd	1–31
day of year	dy	1–366
hour	hh	0–23
millisecond	ms	0–999
minute	mi	0–59
month	mm	1–12
quarter	qq	1–4
second	ss	0–59
week	wk	0–53
weekday	dw	1–7 (Sun–Sat)
year	yy	1753–9999

Table lists the date functions, their parameters, and their results. Bellow table shows some date function examples.

Date Functions

Function	Results
DATEADD(DATEPART, Number, DATE)	Adds the number of *date parts* to the *date*
DATEDIFF(DATEPART, Date1, Date2)	Number of *date parts* between two dates
DATENAME(DATEPART, DATE)	Returns ASCII value for specified *datepart* for *date* listed
DATEPART(DATEPART, DATE)	Returns integer value for specified *datepart* for *date* listed
GETDATE()	Current date and time in internal format

Date Function Examples

Function	Results
SELECT DATEDIFF(mm, '1/1/97', '12/31/99')	35
SELECT GETDATE()	2007-04-27 11:55:38.530
SELECT DATEADD(mm, 6, '1/1/97')	1997-07-01 00:00:00.000
SELECT DATEADD(mm, -5, '10/6/97')	1997-05-06 00:00:00.000

Exercise

1. Write a query to convert '5000' string to decimal and add it to 3000
2. Write a query to convert 5000 to string and append this to 'Sample' string.
3. Write a query to get all employee names in Uppercase.
4. Write a query to get current date.
5. Write a query to get Employee complete name with combining TitleOfCourtesy, FirstName and lastname.
6. Write a query to get the Sqare and Sqareroot of 64.
7. Write a query to count how many items for each supplier.
8. Write a query that will find total stock for each supplier.
9. Create a query that returns the total unitsinstock of products group by category
10. List all suppliers ID who have 2 or more products.
11. Write a query to find the Square root value for 100.
12. Write a query to find the difference between today's date and 15th July 2006.
13. Write a query to count "contact title" without duplication in "suppliers" table
14. Write a query to find the total "UnitPrice" for each category
15. Write a query to get a month, Year from today's date.

Chapter V

User Defined Functions

There are three types of UDF in SQL Server 2000:

- Scalar functions
- Inline table-valued functions
- Multistatement table-valued functions

Scalar functions return a single data value (not a table) with RETURNS clause. Scalar functions can use all scalar data types, with exception of timestamp and user-defined data types.

Inline table-valued functions return the result set of a single SELECT statement.

Multistatement table-valued functions return a table, which was built with many TRANSACT-SQL statements.

User-defined functions can be invoked from a query like built-in functions such as OBJECT_ID, LEN, DATEDIFF, or can be executed through an EXECUTE statement like stored procedures.

UDF Scalar functions examples

Database creation date: This UDF will return the creation date for a given database (you should specify database name as parameter for this UDF):

```
CREATE FUNCTION dbo.DBCreationDate
  ( @dbname sysname )
RETURNS datetime
AS
BEGIN
  DECLARE @crdate datetime
  SELECT @crdate = crdate FROM master.dbo.sysdatabases
    WHERE name = @dbname
  RETURN ( @crdate )
END
GO
```

This is the example for use:

Statement	Result:
SELECT dbo.DBCreationDate('Northwind')	2000-08-06 01:41:00.310
SELECT dbo.DBCreationDate('pubs')	2000-08-06 01:40:58.560
SELECT dbo.DBCreationDate('sampleDb')	2007-03-23 14:37:59.043
SELECT dbo.ObjCreationDate('Orders')	2000-08-06 01:34:06.610
SELECT dbo.ObjCreationDate('Products')	2000-08-06 01:34:07.700

Date the object was created

This UDF will return the creation date for a given object in the current database:

```
CREATE FUNCTION dbo.ObjCreationDate ( @objname sysname)
RETURNS datetime
AS
BEGIN
  DECLARE @crdate datetime
  SELECT @crdate = crdate FROM sysobjects WHERE name = @objname
  RETURN ( @crdate )
END
```

This is the example for use:

Statement	**Result:**
SELECT dbo.ObjCreationDate('Employees')	2000-08-06 01:34:04.547
SELECT dbo.ObjCreationDate('Orders')	2000-08-06 01:34:06.610
SELECT dbo.ObjCreationDate('vwProductSales')	2007-04-21 11:06:21.530

UDF Inline table valued functions examples

```
USE Northwind
go
CREATE FUNCTION fx_Customers_ByCity
( @City nvarchar(15) )
RETURNS table
AS
RETURN (
        SELECT CompanyName
        FROM Customers
        WHERE City =@City
                )
go
```

This is the example for use:

```
SELECT * FROM fx_Customers_ByCity('London')
```

Result:

	CompanyName
1	Around the Hom
2	B's Beverages
3	Consolidated Holdings
4	Eastern Connection
5	North/South
6	Seven Seas Imports

UDF Multistatement table-valued functions examples

The multi-statement table-valued function is slightly more complicated than the other two types of functions because it uses multiple statements to build the table that is returned to the calling statement. Unlike the inline table-valued function, a table variable must be explicitly declared and defined. The following example shows how to implement a multi-statement table-valued function that populates and returns a table variable.

```
USE Northwind
go
CREATE FUNCTION fx_OrdersByDateRangeAndCount
( @OrderDateStart smalldatetime,
  @OrderDateEnd smalldatetime,
  @OrderCount smallint )
RETURNS @OrdersByDateRange TABLE
  (  CustomerID nchar(5),
     CompanyName nvarchar(40),
     OrderCount smallint,
     Ranking char(1) )
AS
BEGIN
 --Statement 1
 INSERT @OrdersByDateRange
 SELECT a.CustomerID,
        a.CompanyName,
        COUNT(a.CustomerID)AS OrderCount,
       'B'
 FROM Customers a
```

```
 JOIN Orders b ON a.CustomerID =b.CustomerID
 WHERE OrderDate BETWEEN @OrderDateStart AND @OrderDateEnd
 GROUP BY a.CustomerID,a.CompanyName
 HAVING COUNT(a.CustomerID)>@OrderCount

--Statement 2
UPDATE @OrdersByDateRange
SET Ranking ='A'
WHERE CustomerID IN (SELECT TOP 5 WITH TIES CustomerID
                FROM (SELECT a.CustomerID,
                        COUNT(a.CustomerID)AS OrderTotal
                    FROM Customers a
                    JOIN Orders b ON a.CustomerID =b.CustomerID
                    GROUP BY a.CustomerID) AS DerivedTable
                    ORDER BY OrderTotal DESC)
RETURN
END
```

The rows ranking values of 'A' indicate the top five order placers of all companies. The function allows you to perform two operations with one object. Retrieve the companies who have placed more than two orders between 1/1/96 and 1/1/97 and let me know if any of these companies are my top five order producers.

This is the example for use:

```
SELECT *
FROM fx_OrdersByDateRangeAndCount ('1/1/96','1/1/97',2)
ORDER By Ranking
```

Result:

	CustomerID	CompanyName	OrderCount	Ranking
1	ERNSH	Ernst Handel	6	A
2	FOLKO	Folk och fä HB	3	A
3	HUNGO	Hungry Owl All-Night Grocers	5	A
4	SAVEA	Save-a-lot Markets	3	A
5	QUICK	QUICK-Stop	6	A
6	RATTC	Rattlesnake Canyon Grocery	7	B
7	ROMEY	Romero y tomillo	3	B
8	SEVES	Seven Seas Imports	3	B
9	SPLIR	Split Rail Beer & Ale	5	B
10	TORTU	Tortuga Restaurante	4	B
11	VINET	Vins et alcools Chevalier	3	B
12	WANDK	Die Wandernde Kuh	4	B
13	WARTH	Wartian Herkku	4	B
14	ISLAT	Island Trading	3	B
15	LAMAI	La maison d'Asie	3	B
16	LEHMS	Lehmanns Marktstand	3	B
17	LILAS	LILA-Supermercado	5	B
18	MEREP	Mère Paillarde	3	B
19	QUEDE	Que Delícia	3	B
20	FRANK	Frankenversand	4	B
21	BERGS	Berglunds snabbköp	3	B
22	BLONP	Blondesddsl père et fils	3	B
23	BONAP	Bon app'	3	B

Exercise

1. Write a scalar function to get product of three numbers.
2. Write a scalar function to get Largest of two numbers.
3. Write a scalar function to get factorial of a number.
4. Write a function to get the all orders whose freight is more than 100.
5. Write a function to get the products for a supplier. (hint use products table, supplier ID as paramater)
6. Write a scalar to get Last Day of the month.
7. Write a function, which accepts a city and returns a table containing all employees' first name, last name, and address.
8. Write a function to get the month name using month Number.
9. Write a function to get number in words up to 99.
10. Write a function to receive number as parameter and return in this format 'Rs.' + sent number e.g. value entered 1000 then result is Rs. 1000.00. Value is 1234 the Rs. 1234.00.

Chapter VI

Variable fundamentals

Declaring Variables

A variable is an area of memory used to store values that can be used in a program.

Here is an example:

```
DECLARE @Count INT

DECLARE @264
```

Such a name made of digits can create confusion with a normal number. So avoid using such numbers as name of a variable

You can also declare more than one variable. To do that, separate them with a comma. The formula would be:

```
DECLARE @Count INT, @Average FLOAT,@Gender BIT
```

Initializing a Variable

We can initialize using following two methods.

```
SELECT @Count = Count(*) FROM employees
```

or

```
SET @Gender = 1
```

Once a variable has been initialized, you can make its value available or display it. This time, you can type the name of the variable to the right side of **PRINT** or **SELECT**.

Logical comparisons

A comparison is a Boolean operation that produces a true or a false result, depending on the values on which the comparison is performed. A comparison is performed between two values of the same type; for example, you can compare two numbers, two characters, or the names of two cities. To support comparisons, Transact-SQL provides all necessary operators.

Equality Operator	=	V1 = V2	If *V1* and *V2* are the same, the comparison produces a TRUE result. Otherwise FALSE
Not Equal	<>	V1 <> V2	If both hold different values, the comparison produces a TRUE. Otherwise FALSE.
Less Than	<	V1 < V2	If the V1 is lower than that of V2, the comparison produces a true. Otherwise false
Less Than or Equal to	<=	V1 <= V2	If the V1 is lower than or equal to that of V2, the comparison produces a true. Otherwise false
Greater Than	>	V1 > V2	If the V1 is greater than that of V2, the comparison produces a true. Otherwise false
Greater Than or Equal to	>=	V1 >= V2	If the V1 is greater than or Equal to that of V2, the comparison produces a true. Otherwise false

Conditional statements

BEGIN…END

With the above formula, we will always let you know what keyword you can use, why, and when. After the expression, you can write the statement in one line. This is the statement that would be executed if/when the *Expression* of our formula is satisfied. In most cases, you will need more than one line of code to specify the *Statement*.

As it happens, the interpreter considers whatever comes after the *Statement* as a unit but only the line immediately after the *Expression*. To indicate that your *Statement* covers more than one line, start it with the **BEGIN** keyword. Then you must use the **END** keyword to indicate where the *Statement* ends. In this case, the formula of a conditional statement would appear as follows:

```
Keyword Expression
BEGIN
 Statement Line 1
 Statement Line 2

 Statement Line n
END
```

You can still use the **BEGIN...END** combination even if your *Statement* covers only one line:

```
Keyword Expression
BEGIN
 Statement
```

```
END
```

Using the **BEGIN...END** combination makes your code easier to read because it clearly indicates the start and end of the *Statement*.

EXISTS

In the previous section, we used **IN** to link the inner query and the outer query in a sub query statement. **IN** is not the only way to do so -- one can use many operators such as >, <, or =. **EXISTS** is a special operator that we will discuss in this section.

EXISTS simply tests whether the inner query returns any row. If it does, then the outer query proceeds. If not, the outer query does not execute, and the entire SQL statement returns nothing.

The syntax for **EXISTS** is:

```
SELECT "column_name1"
FROM "table_name1"

WHERE EXISTS (SELECT * FROM "table_name2" WHERE [Condition])
```

IF Condition

Probably the primary comparison you can perform on a statement is to find out whether it is true. This operation is performed using an **IF** statement in Transact-SQL. Here is an example.

```
IF @Name = ''
BEGIN
        PRINT 'Name is empty'
END
```

IF...ELSE

```
IF @Gender = 1
BEGIN
        PRINT 'Male'
END
ELSE
BEGIN
        PRINT 'Female'
END
```

CASE...WHEN...THEN

```
SET @Gender =
        CASE @CharGender
            WHEN 'm' THEN 'Male'
            WHEN 'M' THEN 'Male'
            WHEN 'f' THEN 'Female'
            WHEN 'F' THEN 'Female'
        END
```

CASE...WHEN...THEN...ELSE

```
SET @Gender =
        CASE @CharGender
            WHEN 'm' THEN 'Male'
            WHEN 'M' THEN 'Male'
            WHEN 'f' THEN 'Female'
            WHEN 'F' THEN 'Female'
            ELSE 'Unknown'
        END
```

This means that it is a valuable safeguard to always include an **ELSE** sub-statement in a **CASE** statement.

WHILE

To examine a condition and evaluate it before taking action, you can use the **WHILE** operator.

```
DECLARE @Number INT
SET @Number = 1
WHILE @Number < 5
BEGIN
        PRINT @Number
        SET @Number = @Number + 1
END
```

BREAK and CONTINUE Statements

You may want to break out of the loop if some particular condition arises.

```
DECLARE @counter INT
SET @counter = 0
WHILE @counter < 5
BEGIN
  IF @counter = 2
            BREAK
  SET @counter = @counter + 1
  PRINT @Counter
END
```

It will stops the loop when counter reaches 2 even though while condition is true.

```
DECLARE @counter INT
SET @counter = 0
WHILE @counter < 5
BEGIN
        SET @counter = @counter + 1
        IF @counter < 3
          continue
        PRINT @Counter
END
```

It will print 3, 4 and 5 since before 3 it will skips executing the Print statement

GOTO Statement

```
DECLARE @counter INT
SET @counter = 0
WHILE @counter < 5
BEGIN
        SET @counter = @counter + 1
        IF @counter < 3
          GOTO BLOCK
        PRINT @Counter

        Block:
        PRINT @Counter + 10
END
```

It will print

11

12

3

13

4

14

5

15

Boolean constants

The TRUE and FALSE Constants

The comparison for a True or False value is mostly performed on Boolean fields. If a record has a value of 1, the table considers that such a field is **True**. If the field has a 0 value, then it holds a **FALSE** value.

The NULL Constant

To support the null value, Transact-SQL provides a constant named **NULL**. The **NULL** constant is mostly used for comparison purposes. For example,

```
IF @ID = NULL
BEGIN
        PRINT 'With NULL ID record can not be searched'
END
```

The IS Operator

```
IF @ID IS NULL
BEGIN
        PRINT 'A NULL value is not welcome'
END
```

The NOT Operator

To deny the presence, the availability, or the existence of a value, you can use the **NOT** operator. This operator is primarily used to reverse a Boolean value. For example,

```
IF NOT @ID < 0
BEGIN
        PRINT 'ID is Positive'
END
```

Stored Procedures

Benefits of Stored Procedures

- Execution plan retention and reuse
- Query auto parameterization
- Encapsulation of business rules and policies
- Application modularization
- Sharing of application logic between applications
- Access to database objects that is both secure and uniform
- Consistent, safe data modification
- Network bandwidth conservation
- Support for automatic execution at system start-up

Precompiled execution: SQL Server compiles each stored procedure once and then reutilizes the execution plan. This results in tremendous performance boosts when stored procedures are called repeatedly.

Reduced client/server traffic: If network bandwidth is a concern in your environment, you'll be happy to learn that stored procedures can reduce long SQL queries to a single line that is transmitted over the wire.

Efficient reuse of code and programming abstraction: Stored procedures can be used by multiple users and client programs. If you utilize them in a planned manner, you'll find the development cycle takes less time.

Enhanced security controls: You can grant users permission to execute a stored procedure independently of underlying table permissions

A simple stored procedure.

```
-- Lists all CUSTOMERS with Company,city and country

CREATE PROCEDURE dbo.ListCustomers

AS
BEGIN
	SELECT
		ContactName,
		CompanyName,
		City,
		Country
	FROM Customers
END
```

Executing Stored Procedures

Executing a stored procedure can be as easy as listing it on a line by itself.

```
EXEC dbo.ListCustomers
OR
EXECUTE dbo.ListCustomers
```

Stored Procedure with parameters

```
--Inerts New Employee
CREATE PROCEDURE Sp_AddEmployee
@Lastname nvarchar(20)
@Firstname nvarchar(20)
@Title nvarchar(30)
@TitleOfCourtersy  nvarchar(25)
@Address nvarchar(60)
@City nvarchar(15)
@Region nvarchar(15)
@Country nvarchar(15)
@PostalCode nvarchar(10)
@HomePhone nvarchar(25)
AS
BEGIN
        INSERT INTO Employees(
                                        Lastname,
                                        Firstname,
                                        Title,
                                        TitleOfCourtersy,
                                        Address,
                                        City,
                                        Region,
                                        Country,
                                        PostalCode,
                                        HomePhone
                                )
```

```
        VALUES(
                        @Lastname,
                        @Firstname,
                        @Title,
                        @TitleOfCourtersy ,
                        @Address,
                        @City,
                        @Region,
                        @Country,
                        @PostalCode,
                        @HomePhone
                )
END
```

Stored Procedure with optional parameters

```
-- Lists all Cities with number of customers from each city Country name -- is optional
Parameter if not supplied lists from all countries.



CREATE PROCEDURE dbo.ListCustomersByCity
@Country VARCHAR(30)='%'
AS

BEGIN
	SET NOCOUNT OFF;
	SELECT City, COUNT(*) AS NumberOfCustomers
	FROM Customers
	WHERE Country LIKE @Country
	GROUP BY City
	SET NOCOUNT ON;
END
GO
```

Executing Stored Procedures with parameter

```
EXEC dbo.ListCustomersByCity 'USA'
```

More than one parameter then they are separted by comma

```
EXEC dbo.Sample 'Fuller','Andrew','Vice President','Dr.'
```

Modifying a Procedure

As a regular SQL Server database object, you can modify a stored procedure without recreating it.

```
-- Lists all CUSTOMERS with Company,city and country
ALTER PROCEDURE dbo.ListCustomers

AS
	BEGIN
		SELECT
			ContactName,
			CompanyName,
			City,
			Country
		FROM Customers
		ORDER BY ContactName
	END
```

Deleting a Procedure

To delete a procedure in SQL, the syntax to use is:

```
DROP PROCEDURE dbo.ListCustomers
```

Of course, you should make sure you are in the right database and also that the Procedure Name exists.

Exercise

Note: use Northwind database

1. Write a stored procedure to get all employees First Name, last Name, city, country.
2. Write a stored procedure to list out all cities without repeating the city name.
3. Write a stored procedure to list all suppliers.
4. Write a stored procedure to list all customers from a city (Hint: User Customers table, use City name as parameter)
5. Write a stored procedure to list all product name with Supplier contact name and city (Hint: use Supplier and Products tables)
6. Write a stored procedure to list all regions with their territory description even if there is no territory for the region than also it should list region name.
7. Write a stored procedure to get Company Name, Contact Name, city, Country and Region if region is null than it should list as "Unknown'. (Hint use suppliers table and Case when then statement)
8. Write a stored procedure to get all orders details for each Customer. (Hint Use CustomerID as parameter).
9. Write a stored procedure to receive Unitprice and list all products, which are less then or equal to the Unitprice. (Hint use products table and if statement).
10. Write a stored procedure to get order details for a particular order. (Hint: use OrderID as parameter).
11. Write a Stored procedure to return the power of value.
12. Write a stored procedure to check employee lastname endswith "ng" then update that records Title into "sales Executive"

Chapter VII

Cursors

Cursor Functions

Recall that cursors are one way to loop through records within a table (or several tables joined together) and perform a certain action on each affected record. SQL Server supports three functions that can help you while working with cursors: @@FETCH_STATUS, @@CURSOR_ROWS, and CURSOR_STATUS. Cursor functions are non-deterministic.

In order to understand how cursor functions work, you must first be familiar with the cursor's life cycle. Although there is not room to go into a detailed discussion of cursors, the typical cursor life cycle is as follows:

- **Declare the cursor:** The cursor is declared using the DECLARE CURSOR statement—this simply creates a cursor within SQL Server memory.
- **Open the Cursor:** The cursor is OPENED—at this point, you can start populating the cursor with rows. (However, the cursor doesn't have any data yet.)
- **Fetch the Cursor:** The cursor is populated by using the FETCH keyword.
- **Repeat through each record:** A WHILE loop is executed within the cursor to do some work with the rows in the cursor, with the condition that FETCH command is successful.
- **Close the Cursor:** The cursor is CLOSED. At this point, you can't populate the cursor with additional rows, nor can you work with rows within the cursor. However, you can reopen the cursor with the OPEN keyword and perform additional work with the cursor.

- **Deallocate the Cursor:** Finally, the cursor is DEALLOCATED. At this point, the cursor representation is destroyed and the cursor cannot be resurrected.

Example Cursor to print each employee name

```
DECLARE @EmpName VARCHAR(50)
-- First Declare the Cursor
DECLARE c1 CURSOR FOR
SELECT FirstName
FROM employees

-- Open the Cursor
OPEN c1

--Fetch first record from the cursor
FETCH NEXT FROM c1
INTO @EmpName
-- loop through each row
WHILE @@FETCH_STATUS = 0
BEGIN

        PRINT @EmpName
        -- fetching the next record
        FETCH NEXT FROM c1
        INTO @EmpName
END

-- close and deallocate the cursor
CLOSE c1
DEALLOCATE c1
```

Triggers

Trigger is special kind of stored procedure that executes automatically when a user attempts the specified data-modification statement on the specified table. SQL Server allows the creation of multiple triggers for any given INSERT, UPDATE, or DELETE statement.

There are two types of trigger available

- **After triggers** –
    - The type of trigger that gets executed automatically after the statement that triggered it completes is called an AFTER trigger.
    - An AFTER trigger is a trigger that gets executed automatically before the transaction is committed or rolled back.
    - These triggers fire after the data is updated - hence if the data violates a constraint then the error will occur before the trigger fires.
    - It is possible to define multiple after triggers on a table. These will all fire. The first and last trigger to firs can be set by sp_settriggerorder.

- **Instead of triggers.**

    - These triggers fire instead of the update taking place and can be applied to updateable views.
    - This means that the trigger can alter the data to fit in with database integrity but has the added complication that the trigger must apply the updates itself.

Of course only one instead of trigger is allowed on a table.

INSTEAD OF triggers gets executed automatically before the Primary Key and the Foreign Key constraints are checked, whereas the traditional AFTER triggers gets executed automatically after these constraints are checked

Example trigger which displays messege when any one inserts or updates the employee table

```
USE Northwind
GO
        CREATE TRIGGER reminder
        ON employees
        FOR INSERT, UPDATE
        AS RAISERROR ('Employe Table updated', 16, 10)
GO
```

INSTEAD OF Triggers

- INSTEAD OF triggers are inherently different from AFTER triggers because INSTEAD OF triggers fire in place of the triggering action.
- So using the same example, if an INSTEAD OF UPDATE trigger exists on the Employees table and an UPDATE statement is executed against the Employees table, then the UPDATE statement will not change a row in the Employees table.
- Instead, the UPDATE statement is used to kick off the INSTEAD OF UPDATE trigger, which may or may not modify data in the Employees table.

So how do you determine the right time and place to use an INSTEAD OF trigger?

- Several key factors are worth considering when making this decision.

- AFTER triggers are more commonly used in situations where actions must be taken following data modifications on tables. For example, an AFTER trigger could be used to log any data updates to a separate auditing table.
- INSTEAD OF triggers could do the same job, but they are less efficient in this particular scenario since the update will be allowed exactly as it occurred after writing to the audit table.

- In general, in any situation where the data modification will not be affected, an AFTER trigger is more efficient.

- An AFTER trigger is also a great choice when the data modification is evaluated and is either allowed to commit as a whole or denied entirely. For example, a rule could exist that any change to a product's price of more than 30 percent in the Products table must be undone.

- An AFTER trigger could do the job here nicely, using the inserted and deleted tables to compare the price of the product and then roll back the transaction if need be.

- These are ideal situations for AFTER triggers, but sometimes INSTEAD OF triggers are better.

INSTEAD OF triggers are great features that allow you to perform complex action queries in place of a single action query on a table or a view.

- Unlike AFTER triggers, which can only be created against tables, INSTEAD OF triggers can be created against both tables and views.

How to resolve the situation in which there is a view that represents a join of multiple tables and you want to allow an update to the view.

If the view exposes the primary key and required fields of a base table, it is often simple to update the view's base table. However, when there are multiple tables represented in a join, the update logic is more complicated than a single UPDATE statement.

So how do you resolve this situation using alternative tools?

- One solution is to place an INSTEAD OF UPDATE trigger on the view.
- INSTEAD OF triggers can be defined on views with one or more tables.
- The INSTEAD OF trigger can then extend the type of modifications that will take place on the multiple base tables.

For example, if a view joins the Customers, Products, Orders, and Order Details tables together to show all of the data on a screen through an application, updates could be allowed to take place through this view. Assuming a view exists that joins these four tables in the Northwind database and is named vwCustomersOrdersOrderDetailsProducts, it could look like

```
CREATE VIEW vwCustomersOrdersOrderDetailsProducts
AS
   SELECT    c.CustomerID,
      c.CompanyName,
      o.OrderID,
      o.OrderDate,
      od.UnitPrice,
      od.Quantity,
      od.Discount,
      p.ProductID,
      p.ProductName
   FROM Customers c
      INNER JOIN Orders o ON c.CustomerID = o.CustomerID
      INNER JOIN [Order Details] od ON o.OrderID = od.OrderID
      INNER JOIN Products p ON od.ProductID = p.ProductID
GO
```

```
-- INSTEAD OF Trigger Updating a View
CREATE TRIGGER tr_vwCustomersOrdersOrderDetailsProducts_IO_U
   ON vwCustomersOrdersOrderDetailsProducts
   INSTEAD OF UPDATE
AS
   — Update the Customers
   UPDATE     Customers
   SET    CompanyName = i.CompanyName
   FROM    inserted i
      INNER JOIN Customers c ON i.CustomerID = c.CustomerID

   — Update the Orders
   UPDATE     Orders
   SET    OrderDate = i.OrderDate
   FROM    inserted i
      INNER JOIN Orders o ON i.OrderID = o.OrderID

   — Update the Order Details
   UPDATE     [Order Details]
   SET    UnitPrice = i.UnitPrice,
      Quantity = i.Quantity
   FROM    inserted i
      INNER JOIN [Order Details] od ON i.OrderID = od.OrderID AND
         i.ProductID = od.ProductID

   — Update the Products
   UPDATE     Products
```

```
    SET    ProductName = i.ProductName
    FROM    inserted i
        INNER JOIN Products p ON i.ProductID = p.ProductID
GO
```

- The vwCustomersOrdersOrderDetailsProducts view joins the four tables and exposes a sampling of fields from each of the tables.
- One key to remember when designing views that will have INSTEAD OF UPDATE triggers is that it is helpful to include the primary key fields from each table in the SELECT statement.
- Even if these fields are not used in the application, they can be used in the INSTEAD OF trigger to locate the row(s) that were intended to be modified and then make the appropriate changes in the base tables.
- Let's assume that you want to allow updates to this view to funnel down to the base tables on the non-key fields.
- Code would then have to be written in the INSTEAD OF UPDATE trigger to update the CompanyName in the Customers table, the OrderDate in the Orders table, the UnitPrice and Quantity in the Order Details table, and the ProductName in the Products table.
- At this point using an AFTER trigger won't cut it, but an INSTEAD OF trigger is a good option.

```
CREATE TRIGGER tr_vwCustomersOrdersOrderDetailsProducts_IO_U
   ON vwCustomersOrdersOrderDetailsProducts
   INSTEAD OF UPDATE
AS
   — Update the Customers
   UPDATE     Customers
   SET    CompanyName = i.CompanyName
   FROM    inserted i
      INNER JOIN Customers c ON i.CustomerID = c.CustomerID

   — Update the Orders
   UPDATE     Orders
   SET    OrderDate = i.OrderDate
   FROM    inserted i
      INNER JOIN Orders o ON i.OrderID = o.OrderID

   — Update the Order Details
   UPDATE     [Order Details]
   SET    UnitPrice = i.UnitPrice,
      Quantity = i.Quantity
   FROM    inserted i
      INNER JOIN [Order Details] od ON i.OrderID = od.OrderID AND
         i.ProductID = od.ProductID

   — Update the Products
   UPDATE     Products
   SET    ProductName = i.ProductName
   FROM    inserted i
      INNER JOIN Products p ON i.ProductID = p.ProductID
```

GO

- Notice that the code in the INSTEAD OF UPDATE trigger in (tr_vwCustomersOrdersOrderDetailsProducts_IO_U) contains four UPDATE statements.
- Each targets one of the base tables with the purpose of modifying the exposed non-key fields. The key fields for each base table are joined in the UPDATE statements to the corresponding fields that the view exposes.
- This allows the UPDATE statements to locate the appropriate rows in the underlying tables and update only those rows.
- The following UPDATE statement puts this INSTEAD OF trigger to the test:

```
UPDATE   vwCustomersOrdersOrderDetailsProducts
SET    Quantity = 100,
       UnitPrice = 20,
       CompanyName = 'Fake Name',
       OrderDate = '11/23/2001',
       ProductName = 'Widget'
WHERE    OrderID = 10265
AND      ProductID = 17
```

- If you examine the values in the underlying tables (either through the view or in the tables themselves) it is evident that the values have been updated.
- Of course, several modifications could be made to this INSTEAD OF UDPATE trigger to yield different results. For example, it is not a requirement to write the INSTEAD OF trigger to modify all four base tables.
- Therefore, one or more of the UDPATE statements contained within the trigger could be removed. Assuming that the trigger was intended only to update the Order Details values, it could be modified to update only those fields in the Order Details table—ignoring any changes that were attempted on the other base tables.
- In this situation, no error is raised nor are the changes made to Customers, Products, or Orders tables.
- Of course, an error could be raised if one of these fields was updated.
- The UPDATE and COLUMNS_UPDATED functions are ideal for checking what fields have been modified, as I'll demonstrate later in this column.

Checking for Changes

- The UPDATE and COLUMNS_UPDATED functions are available within both types of triggers to allow the trigger to determine which columns were modified by the triggering action statement.

For example, the following trigger prevents any modifications to the lastname column in the Employees table.

- Here, the UPDATE function is used to determine if a modification was made to the column. If so, then an error is raised using the RAISERROR function and the transaction is rolled back, which undoes any changes that were made.
- The UPDATE function works in both the AFTER and INSTEAD OF triggers, but not outside of the triggers:

```
CREATE TRIGGER tr_Employees_U on Employees AFTER UPDATE AS
   IF UPDATE(lastname)
   BEGIN
      RAISERROR ('cannot change lastname', 16, 1)
      ROLLBACK TRAN
      RETURN
   END
GO
```

- The UPDATE function is designed to determine if a single column was modified by an INSERT OR UPDATE statement.

- UPADATE (column) is the standard method used to check for updates. However, it becomes less efficient when the need arises to check if multiple columns were affected by an INSERT or UPDATE.
- This is where the COLUMNS_UPDATED function steps into the spotlight. The COLUMNS_UPDATED function returns a bit mask to evaluate if specific columns were modified.
- The bit mask contains a bit for every column in the table that was modified, in the order that the columns are defined in the table's schema.
- f the column was modified, the bit's value is 1; otherwise it is 0.
- Unlike the conventional way to read bytes going right to left, the bit mask reads from left to right.
- For example, the following code shows a trigger on the Order Details table that checks to see if both the Quantity and UnitPrice fields are modified:

```
CREATE TRIGGER tr_OrderDetails ON [Order Details] AFTER UPDATE
AS
   IF (COLUMNS_UPDATED() = 12)
   BEGIN
      RAISERROR ('Cannot change both UnitPrice and Quantity at the
                  same time', 16, 1)
      ROLLBACK TRAN
   END
GO
```

- If both fields are modified, an error is raised and the transaction is rolled back.
- For the Order Details table, the COLUMNS_UPDATED function returns a single byte with the first five bits representing the columns in the Order Details table.
- Since the situation called for determining if only the third and fourth columns were modified, it looked to see if only those bits were set to 1.
- When the third and fourth bits are turned on, it looks like this: 00110.
- Since this bit mask represents the powers of 2, the first bit represents 1, the second represents 2, the third represents 4, the fourth represents 8, and the fifth represents 16 (yes, this is the reverse order of bits in a normal binary number).
- Therefore, the bit mask value that means only the UnitPrice and Quantity were changed is 00110, which equates to the integer value of 12 (4 + 8).
- Note that this trigger only rolls back the transaction if the UnitPrice and Quantity fields were modified.
- If any other fields were modified, the bit mask would be different and thus not equal to the integer value of 12.
- If the trigger were modified to prohibit changes to these two fields even if other fields were modified, it could be rewritten like so:

```
ALTER TRIGGER tr_OrderDetails ON [Order Details] AFTER UPDATE
AS
   IF (COLUMNS_UPDATED() & 12 >= 12)
   BEGIN
      RAISERROR ('Cannot change both UnitPrice and Quantity
            at the same time', 16, 1)
      ROLLBACK TRAN
   END
GO
```

Exercise

1. What is the cursor?
2. What is use of cursors?
3. What command is used to retrieve the results after a cursor has been opened?
4. Are triggers executed before or after an INSERT, DELETE, or UPDATE?
5. What are the steps to be followed when using cursors?
6. Process a table using simple cursor?
7. Explain Triggers?
8. Create any sample table and write simple triggers on this table.
9. What is the use of instead of triggers?

Chapter VIII

Error Handling

When you write your own client program, you can choose your own way to display error messages. You may be somewhat constrained by what your client library supplies to you. The full information is available with low-level interfaces such as DB-Library, ODBC or the OLE DB provider for SQL Server. On the other hand, in ADO you only have access to the error number and the text of the message.

There are two ways an error message can appear:

1) An SQL statement can result in an error (or a warning)

2) you emit it yourself with RAISERROR (or PRINT).

Let's take a brief look at RAISERROR here. Here is sample statement:

```
RAISERROR('This is a test', 16, 1)
```

Here you supply the message text, the severity level and the state. The output is:

```
Server: Msg 50000, Level 16, State 1, Line 1
This is a test.
```

Thus, SQL Server supplies the message number 50000, which is the error number you get when you supply a text string to RAISERROR. (There is no procedure name here, since it ran the statement directly from Query Analyzer.) Rather than a string, you could have a supplied a number of 50001 or greater, and SQL Server would have looked up that number in sysmessages to find the message text. You would have

stored that message with the system procedure **sp_addmessage**. (If you just supply a random number, you will get an error message, saying that the message is missing.) Whichever method you use, the message can include placeholders, and you can provide values for these placeholders as parameters to RAISERROR, something I do not cover here. Please refer to Books Online for details.

As I mentioned *State* is rarely of interest. With RAISERROR, you can use it as you wish.

If you raise the same message in several places, you can provide different values to *State* so that you can conclude which RAISERROR statement that fired. The command-line tools OSQL and ISQL have a special handling of state: if you use a state of 127, the two tools abort and set the DOS variable ERRORLEVEL to the message number. This can be handy in installation scripts if you want to abort the script if you detect some serious condition. (For instance, that database is not on the level that the installation script is written for.) This behavior is entirely client-dependent; for instance, Query Analyzer does not react on state 127.

How to Detect an Error in T-SQL – @@error

SQL Server sets the global variable @@error to 0, unless an error occurs, in which case @@error is set to the number of that error. (Note: these days, the SQL Server documentation refers to @@error as a "function". Being an old-timer, I prefer "global variables" for the entities whose names that start with @@.)

More precisely, if SQL Server emits a message with a severity of 11 or higher, @@error will hold the number of that message. And if SQL Server emits a message with a severity level of 10 or lower, SQL Server does not set @@error, and thus you cannot tell from T-SQL that the message was produced.

There is *no way* to prevent SQL Server from raising error messages. There is a small set of conditions for which you can use SET commands to control whether these conditions are errors or not. We will look closer at these possibilities later, but I repeat that this is a small set, and there is *no* general way in T-SQL to suppress error messages. You will need to take care of that in your client code

As I mentioned, @@error is set after ***each*** statement. Therefore, you should always save the save the value of @@error into a local variable, before you do anything with it. Here is an example of what happens if you don't:

```
CREATE TABLE notnull(a int NOT NULL)
DECLARE @value int
INSERT notnull VALUES (@value)
IF @@error <> 0
   PRINT '@@ERROR IS ' + LTRIM(STR(@@ERROR)) + '.'
```

The output is:

```
Server: Msg 515, Level 16, State 2, Line 3
Cannot insert the value NULL into column 'a', table
'tempdb.dbo.notnull'; column does not allow nulls. INSERT fails.
The statement has been terminated.
@@error is 0.
```

Here is the correct way.

```
CREATE TABLE notnull(a int NOT NULL)
DECLARE @err int,
        @value int
INSERT notnull VALUES (@value)
SELECT @err = @@error
```

```
IF @err <> 0
   PRINT '@err is ' + ltrim(str(@err)) + '.'
```

The output is:

```
Server: Msg 515, Level 16, State 2, Line 3
Cannot insert the value NULL into column 'a', table
'tempdb.dbo.notnull'; column does not allow nulls. INSERT fails.
The statement has been terminated.
@err is 515.
```

Return Values from Stored Procedures

All stored procedures have a return value, determined by the RETURN statement. The RETURN statement takes one optional argument, which should be a numeric value. If you say RETURN without providing a value, the return value is 0 if there is no error during execution. If an error occurs during execution of the procedure, the return value may be 0, or it may be a negative number. The same is true if there is no RETURN statement at all in the procedure: the return value may be a negative number or it may be 0.

Whether these negative numbers have any meaning, is a bit difficult to tell. It used to be the case, which the return values -1 to -99 were reserved for system-generated returns values, and Books Online for earlier versions of SQL Server specified meanings for values -1 to -14. However, Books Online for SQL 2000 is silent on any such reservations, and does not explain what -1 to -14 would mean.

With some occasional exception, the system stored procedures that Microsoft ships with SQL Server return 0 to indicate success and any non-zero value indicates failure.

IN SQL Server 2005 you can use Try catch to handle the errors efficiently

LEFT

TRY...CATCH

Let's look at what TRY...CATCH brings to the table.

As you've probably guessed, TRY..CATCH is a TSQL implementation of the exception handling semantics we've grown to love in .NET / C++.

- A TRY block determines the scope of the TSQL statements that will be monitored for RAISERRORs (caused by either SQL Server or custom RAISERRORs).
- If an error is raised the TRY block is immediately exited and control is passed to the corresponding CATCH block.
- TRY..CATCH blocks can be nested in the same manner as C#. However, there is no concept of a FINALLY block.
- The following TSQL is an example of a simple TRY...CATCH block. Products is the table name with ProductID field is having Auto Seed on

```
BEGIN TRY
   -- Fails with a primary key violation
   INSERT INTO Products (ProductID,Description,Price)
   VALUES (20,'LCD Projector','75000')

   PRINT 'Will not be reached...'
END TRY

BEGIN CATCH
   PRINT 'In catch block...'
END CATCH
```

Running this produces the following output:

In catch block...

The good thing about this behavior is that we no longer need to include the standard @@ERROR test after any DML. In fact, if the code is within a TRY block we'll never get a chance to test @@ERROR anyway. 'Will not be reached...' is never printed. Also note that the stored procedure did not report any error.

SQL Transactions

Def: A *transaction* is a unit of work that is performed against a database. Transactions are units or sequences of work accomplished in a logical order, whether in a manual fashion by a user or automatically by some sort of a database program.

A transaction can either be one *DML* statement or a group of statements. When managing transactions, each designated transaction (group of *DML* statements) must be successful as one entity or none of them will be successful.

The following list describes the nature of transactions:

- All transactions have a beginning and an end.
- A transaction can be saved or undone.
- If a transaction fails in the middle, no part of the transaction can be saved to the database.

Transactions group a set of tasks into a single execution unit. Each transaction begins with a specific task and ends when all the tasks in the group successfully complete. If any of the tasks fails, the transaction fails. Therefore, a transaction has only two results: success or failure. Incomplete steps result in the failure of the transaction.

Users can group two or more Transact-SQL statements into a single transaction using the following statements:

- Begin Transaction
- Rollback Transaction
- Commit Transaction

If anything goes wrong with any of the grouped statements, all changes need to be aborted. The process of reversing changes is called rollback in SQL Server terminology. If everything is in order with all statements within a single transaction,

all changes are recorded together in the database. In SQL Server terminology, we say that these changes are committed to the database.

Here is an example of a transaction:

```
USE pubs

DECLARE @intErrorCode INT

BEGIN TRAN
   UPDATE Authors
   SET Phone = '415 354-9866'
   WHERE au_id = '724-80-9391'

   SELECT @intErrorCode = @@ERROR
   IF (@intErrorCode <> 0) GOTO PROBLEM

         UPDATE Publishers
         SET city = 'Calcutta', country = 'India'
         WHERE pub_id = '9999'

          SELECT @intErrorCode = @@ERROR
   IF (@intErrorCode <> 0) GOTO PROBLEM
COMMIT TRAN

PROBLEM:
IF (@intErrorCode <> 0) BEGIN
```

```
    PRINT 'Unexpected error occurred!'
    ROLLBACK TRAN
END
```

Before the real processing starts, the BEGIN TRAN statement notifies SQL Server to treat all of the following actions as a single transaction. It is followed by two UPDATE statements. If no errors occur during the updates, all changes are committed to the database when SQL Server processes the COMMIT TRAN statement, and finally the stored procedure finishes. If an error occurs during the updates, it is detected by if statements and execution is continued from the PROBLEM label. After displaying a message to the user, SQL Server rolls back any changes that occurred during processing. Note: Be sure to match BEGIN TRAN with either COMMIT or ROLLBACK.

Exercise

1. What is the error handling?
2. Explain about "@@Error"?
3. How to find an error in stored procedure?
4. Explain SQL Transaction.
5. Briefly describe the purpose of each one of the following commands: COMMIT, ROLLBACK, and SAVEPOINT.
6. Is it necessary to issue a commit after every INSERT statement?

Chapter IX

Advanced SQL

For advanced SQL we use these two tables.

Table **T1** Table **T2**

	C1	C2
1	a	b
2	a	c
3	a	d

	C1	C2
1	a	b
2	a	c
3	a	e

UNION

The purpose of the SQL **UNION** command is to combine the results of two queries together. In this respect, **UNION** is somewhat similar to **JOIN** in that they are both used to related information from multiple tables. One restriction of **UNION** is that all corresponding columns need to be of the same data type. Also, when using **UNION**, only distinct values are selected (similar to **SELECT DISTINCT**).

The purpose of the SQL **UNION ALL** command is also to combine the results of two queries together. The difference between **UNION ALL** and **UNION** is that, while **UNION** only selects distinct values, **UNION ALL** selects all values.

The syntax is as follows:

[SQL Statement 1]
UNION [ALL]
[SQL Statement 2]

The following are basic rules for combining the result sets of two queries by using UNION:

- The number and the order of the columns must be the same in all queries.
- The data types must be compatible.

```
SELECT C1,C2 FROM T1
UNION
SELECT C1,C2 FROM T2
```

Result:

	C1	C2
1	a	b
2	a	c
3	a	d
4	a	e

```
SELECT C1,C2 FROM T1
UNION ALL
SELECT C1,C2 FROM T2
```

Result:

	C1	C2
1	a	b
2	a	c
3	a	d
4	a	b
5	a	c
6	a	e

INTERSECT

Similar to the **UNION** command, **INTERSECT** also operates on two SQL statements. The difference is that, while **UNION** essentially acts as an **OR** operator (value is selected if it appears in either the first or the second statement), the **INTERSECT** command acts as an **AND** operator (value is selected only if it appears in both statements). Returns distinct values by comparing the results of two queries.

INTERSECT returns any distinct values that are returned by both the query on the left and right sides of the INTERSECT operand.

The basic rules for combining the result sets of two queries that use INTERSECT are the following:

- The number and the order of the columns must be the same in all queries.
- The data types must be compatible.

The syntax is as follows:

```
[SQL Statement 1]
INTERSECT
[SQL Statement 2]
SELECT C1,C2 FROM T1
INTERSECT
SELECT C1,C2 FROM T2
```

Result:

	C1	C2
1	a	b
2	a	c

Note: INTERSECT **will work only in SQL 2005.**

EXCEPT

The **EXCEPT** (MINUS) operates on two SQL statements. It takes all the results from the first SQL statement, and then subtract out the ones that are present in the second SQL statement to get the final answer. If the second SQL statement includes results not present in the first SQL statement, such results are ignored.

EXCEPT returns any distinct values from the left query that are not also found on the right query.

The basic rules for combining the result sets of two queries that use EXCEPT are the following:

- The number and the order of the columns must be the same in all queries.
- The data types must be compatible.

The syntax is as follows:

```
[SQL Statement 1]
EXCEPT
[SQL Statement 2]
SELECT C1,C2 FROM T1
EXCEPT
SELECT C1,C2 FROM T2
```

Result:

	C1	C2
1	a	d

Note: EXCEPT **will work only in SQL 2005**

Sub queries

Types of Sub Queries

There are three types of sub queries distinguished from one another by the result how many rows and columns they return.

- If a sub query can return exactly one column and one row, it is known as a *scalar sub query*. A scalar sub query is legal everywhere that a regular scalar value (e.g. a column value or literal) is legal in an SQL statement. It is usually found in a WHERE clause, immediately after a comparison operator.

- If a sub query can return multiple columns and exactly one row, it is known as a *row sub query*. A row sub query is a derivation of a scalar sub query and can thus be used anywhere that a scalar sub query can be used.

- Finally, if a sub query can return multiple columns and multiple rows, it is known as a *table sub query*. A table sub query is legal everywhere that a table reference is legal in an SQL statement, including the FROM clause of a SELECT. It, too, is usually found in a WHERE clause, immediately after an IN or EXISTS predicate or a quantified comparison operator. (A quantified comparison operator is a comparison operator used with either the SOME, ALL, or ANY quantifiers.)

The difference between scalar and table sub queries can be subtle. Here's a problem that arises when a sub query is written as a scalar sub query, but the sub query result contains multiple rows. Assume our two tables have only these rows:

Table: Clients

Clno	FirstName	LastName	Desgination	Salary
10	Sam	Smith	Auditor	5525.75

Table: firms

Clno	Company	Description
10	Abc co	Lead
30	GHI inc	Nisku

Since the firms table has two rows, this query:

```
SELECT * FROM clients WHERE clno <  (SELECT clno FROM firms);
```

Fails with:

```
Msg 512, Level 16, State 1, Line 1
Subquery returned more than 1 value. This is not permitted when the subquery
follows =, !=, <, <= , >, >= or when the subquery is used as an expression.
```

ANY & ALL Operator

There are two solutions to this. The first is to change the query to include a table sub query quantified by ANY, to compare the outer query results with any sub query value:

```
SELECT * FROM clients WHERE clno < ANY (SELECT clno FROM firms);
```

In this case, the comparison for the first client is false (10 < 10), but is true for the second client (10 < 30), and so the sub query result is "true" for clno 10. The rules for ANY are as follows:

- ANY returns "true" if the comparison operator is "true" for at least one row returned by the sub query.
- ANY returns "false" if the sub query returns zero rows or if the comparison operator is "false" for every row returned by the sub query.

SOME is a synonym for ANY; using IN is equivalent to using = ANY.

The second solution to the problem is to change the query to include a table sub query quantified by ALL, to compare the outer query results with every sub query value:

```
SELECT * FROM clients WHERE clno < ALL (SELECT clno FROM firms);
```

In this case, the comparison is once again false for the first client and true for the second client Ч but this time, the sub query result is "false" and so the query returns zero rows. The rules for ALL are:

- ALL returns "true" if the sub query returns zero rows or if the comparison operator is "true" for every row returned by the sub query.

- ALL returns "false" if the comparison operator is "false" for at least one row returned by the sub query.

Example Using ALL and ANY keywords

```
CREATE TABLE Employee(
    ID          INT,
    Name        VARCHAR(10),
    Salary      INT )
GO


CREATE TABLE job(
    ID                      INT,
    title                   VARCHAR(10),
    averageSalary     INT)
GO

insert into employee (ID, name, salary) values (1,  'Jason', 1234)
insert into employee (ID, name, salary) values (2,  'Robert', 4321)
insert into employee (ID, name, salary) values (3,  'Celia', 5432)
insert into employee (ID, name, salary) values (4,  'Linda', 3456)
insert into employee (ID, name, salary) values (5,  'David', 7654)
insert into employee (ID, name, salary) values (6,  'James', 4567)
insert into employee (ID, name, salary) values (7,  'Alison', 8744)
insert into employee (ID, name, salary) values (8,  'Chris', 9875)
insert into employee (ID, name, salary) values (9,  'Mary', 2345)

insert into job(ID, title, averageSalary) values(1,'Developer',3000)
insert into job(ID, title, averageSalary) values(2,'Tester', 4000)
insert into job(ID, title, averageSalary) values(3,'Designer', 5000)
```

```
insert into job(ID, title, averageSalary) values(4,'Programmer', 6000)
```

After executing above statements you will get the Result:

```
SELECT * FROM employee;
```

	ID	name	salary
1	1	Jason	1234
2	2	Robert	4321
3	3	Celia	5432
4	4	Linda	3456
5	5	David	7654
6	6	James	4567
7	7	Alison	8744
8	8	Chris	9875
9	9	Mary	2345

```
SELECT * FROM job
```

	ID	title	averageSalary
1	1	Developer	3000
2	2	Tester	4000
3	3	Designer	5000
4	4	Programmer	6000

If your subquery returns a scalar value, you can use a comparison operator with ANY,

```
SELECT e.ID,e.name
 FROM Employee e
 WHERE e.salary > ANY (SELECT averageSalary FROM job j)—ANY MEANS 'OR'
```

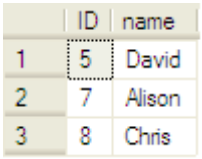
	ID	name
1	2	Robert
2	3	Celia
3	4	Linda
4	5	David
5	6	James
6	7	Alison
7	8	Chris

This is equivalent to the following:

```
SELECT e.ID,e.name
 FROM Employee e
 WHERE (e.salary > 3000 OR
      e.salary > 4000 OR
      e.salary > 5000      OR
      e.salary > 6000)
```

Example with ALL Operator

```
SELECT e.ID,e.name
 FROM Employee e
 WHERE e.salary > ALL (SELECT averageSalary FROM job j) -ALL means
AND
```



	ID	name
1	5	David
2	7	Alison
3	8	Chris

This is equivalent to the following:

```
SELECT
                e.ID,
e.Name
 FROM Employee e
 WHERE (e.salary > 3000 AND
     e.salary > 4000 AND
     e.salary > 5000 AND
     e.salary > 6000)
```

Some more examples using Sub Query

Correlated subquery using Distinct

```
SELECT
        ID,
        Name
FROM Employee AS e
WHERE 1 =  (SELECT DISTINCT ID FROM Job As j WHERE j.ID = e.ID)
```

	ID	Name
1	1	Jason

The WHERE clause in the subquery's SELECT statement links the inner query to the outer query.

This makes the inner query a correlated subquery

```
SELECT
        e.ID,
        e.Name,
        (SELECT j.title FROM job j WHERE j.ID= e.ID) 'Title'
FROM Employee e
```

	ID	name	title
1	1	Jason	Developer
2	2	Robert	Tester
3	3	Celia	Designer
4	4	Linda	Programmer
5	5	David	NULL
6	6	James	NULL
7	7	Alison	NULL
8	8	Chris	NULL
9	9	Mary	NULL

Correlated subquery using Distinct:

Subqueries can be nested.

```
SELECT
        ID,
        Name
FROM Employee
WHERE ID IN
    ( SELECT ID
      FROM Job
      WHERE title IN  ( SELECT title FROM job)
    )
```

	ID	Name
1	1	Jason
2	2	Robert
3	3	Celia
4	4	Linda

Using the EXISTS() Function

It doesn't really matter what column or columns are returned in the subquery because you don't actually use these values.

```
SELECT
        ID,
        Name
FROM Employee
WHERE EXISTS
        (SELECT * FROM Job WHERE ID = Employee.ID)
```

	ID	Name
1	1	Jason
2	2	Robert
3	3	Celia
4	4	Linda

```
-- NOT EXISTS() Add the NOT operator before the EXISTS statement:
SELECT
        ID,
        Name
FROM Employee
WHERE NOT EXISTS
    (SELECT * FROM Job WHERE ID = Employee.ID)
```

	ID	Name
1	5	David
2	6	James
3	7	Alison
4	8	Chris
5	9	Mary

The NTILE Ranking Function

NTILE, RANK, DENSE_RANK and ROW_NUMBER form part of a new set of functions, called 'Ranking Functions' (or 'Ranking Window Functions') that use the new OVER operator.

The ranking functions all return integers whose values and ordering are determined by the context supplied to the OVER operator. The OVER clause determines the ordering the function needs to apply to the result set when it 'ranks' (i.e. evaluates) each row. Typically the OVER clause contains an ORDER BY statement but can include a PARTITION as well

Use of NTILE and OVER is best illustrated with an example. Suppose I wanted to return a result set and have the result set split into equally-sized logical groups. That is, I want a number associated with each row that tells me which group a specific row belongs in (*1...n* where *n* is the number of group). I don't want to specify a page size for these groups. I just want the rows divided equally.

```
DECLARE @numberOfGroups INT;
SET @numberOfGroups = 5;

SELECT NTILE(@numberOfGroups) OVER(ORDER BY Name) AS [GroupNumber],
   Name, salary
FROM employee
```

	GroupNumber	Name	salary
1	1	Alison	8744
2	1	Celia	5432
3	2	Chris	9875
4	2	David	7654
5	3	James	4567
6	3	Jason	1234
7	4	Linda	3456
8	4	Mary	2345
9	5	Robert	4321

When you run this query you'll see that SQL Server has returned a number in the first column that represents the group number of that row. SQL Server has taken the total number of rows returned from the query and divided it into the specified number of groups (which could have been hard coded) and then numbered the groups from *1 to n*, where *n* is the @numberOfGroups. If the number of rows cannot be divided equally amongst the number of groups the lower numbered groups will have more rows in them (at most 1 more).

The OVER clause determines how SQL Server orders the rows when it divides them into groups. You can perform the grouping in one order and return the result set in another one. For example:

```
DECLARE @numberOfGroups INT;
SET @numberOfGroups = 5;

SELECT NTILE(@numberOfGroups) OVER(ORDER BY Name) AS [GroupNumber],
   Name, salary
FROM employee
ORDER BY Salary
```

	GroupNumber	Name	salary
1	3	Jason	1234
2	4	Mary	2345
3	4	Linda	3456
4	5	Robert	4321
5	3	James	4567
6	1	Celia	5432
7	2	David	7654
8	1	Alison	8744
9	2	Chris	9875

In this case the GroupNumber column will appear to be jumbled as SQL Server has assigned the rows into groups according to Name but then sorted the rows on salary before returning the result set. If you just want to return a specific group number you need to use a derived table.

```
DECLARE @numberOfGroups INT;
SET @numberOfGroups = 5;

DECLARE @groupToReturn  INT;
SET @groupToReturn  = 3;

SELECT Name, Salary
FROM (
   SELECT NTILE(@numberOfGroups) OVER(ORDER BY Name) AS [GroupNumber],
      Name, Salary
   FROM employee
        WHERE salary > 1000) AS Grouped
WHERE [GroupNumber] = @groupToReturn
Go
```

	Name	Salary
1	James	4567
2	Jason	1234

This query just returns the third group - very simple paging / chunking.

Note: This works only in SQL server 2005

The OUTPUT Clause

If any new feature of TSQL will be a big hit it will be the OUTPUT clause. This is one new addition to TSQL that you'll use everywhere! How many times have you written this code?

```
-- A typical table...
CREATE TABLE someTable (
   Id INT IDENTITY(1,1) PRIMARY KEY CLUSTERED,
   SomeData VARCHAR(100) NOT NULL);
GO

DECLARE @ID INT;
-- Insert the data.
INSERT someTable (someData)
VALUES ('DataOne');

-- Figure out what identity the row was given.
SET @ID = SCOPE_IDENTITY();
-- Return it...
SELECT @ID AS ID;
GO
```

Isn't it tedious that you have to read the data that you just inserted as a second operation following the insert? The whole reason scope_identity exists is to protect you from any context that might be introduced between these two operations that you may not be aware of, like triggers. After all, why can't you just write the following...

```
DECLARE @ID INT;


-- Insert the data, better...
INSERT someTable (someData)
OUTPUT inserted.Id
VALUES ('DataTwo');
GO
```

This is exactly how the OUTPUT clause works.

- It allows data that is inserted / updated / deleted to be returned.
- Anyone that's done any work with triggers would have immediately recognized the 'virtual' inserted table. In fact, drawing parallels between triggers and the OUTPUT clause is the best way to understand what's going on here.
- The OUTPUT clause allows you to access the virtual trigger tables inline with your INSERT / UPDATE / DELETE statements.
- You essentially get 'on-the-fly' trigger access. For INSERT you have the inserted table, for update you have both the inserted and deleted tables, and for DELETE just the deleted table. Simple, isn't it?
- In the above example we use the OUTPUT clause to get to the identity column we inserted.
- It could have been any column in the table.
- Think how useful this would be for retrieving timestamp values or computed columns.

Have a look at these examples to see what some of the practical applications are:

(1) What did I just delete?

```
DELETE someTable
OUTPUT deleted.*
WHERE ID = 1;
GO
```

This query returns all of the data that was just deleted. Great for audit records! It also prevents having to take out locks on the rows you're going to delete so that the values read are the ones deleted. No encompassing transaction required here.

(2) Before and after values

```
UPDATE someTable
SET someData = 'DataThree'
OUTPUT deleted.someData AS Before,
   inserted.someData AS After,
   DIFFERENCE(deleted.someData, inserted.someData) AS Delta
WHERE ID = 2;
```

	Before	After	Delta
1	DataTwo	DataThree	3

This query will return the values for someData as it was before the update and it is after the update in a single row. In addition, note that the OUTPUT clause is defined as a meaning that any expressions (i.e. functions / calculations, etc.) you could do in a SELECT you can do in an OUTPUT.

(3) Deleting multiple rows

```
-- Declare a table variable.
DECLARE @someTableTemp TABLE (
    ID INT NOT NULL,
    someData VARCHAR(100) NOT NULL);

-- Capture the output into a table variable.
DELETE someTable
OUTPUT deleted.id, deleted.someData
INTO @someTableTemp
    (ID, someData);

-- What was deleted...
SELECT * FROM @someTableTemp;
```

	Id	someData
1	2	DataThree

The output can also be captured into a table variable. It doesn't have to be a table variable. It could be any permanent table. In this case think of the OUTPUT ... INTO as an INSERT ... SELECT. Just for the hell of it you can write crazy TSQL like this:

```
-- The delete that never was!
DELETE someTable
OUTPUT deleted.someData
INTO someTable (someData);
GO
```

Lastly, a couple of points to remember:

- The values in the output virtual tables will be populated after any triggers for the table have fired, so remember that you are at the back of the queue.
- Inserting / updating / deleting data and then re-reading the rows affected is always dangerous because it makes you susceptible to deadlocks. Using the OUTPUT clause will avoid this as the action and read occur as a single unit. In SQL Server 2000 there was a trick with the UPDATE statement that allowed you to avoid re-reading updates if you're interested in the value of a specific column that was updated (this only works if the UPDATE affects a single row). Consider the following table:

```
CREATE TABLE ReserveRange (
   IdName VARCHAR(100) NOT NULL PRIMARY KEY CLUSTERED,
   ID INT NOT NULL);
GO

INSERT ReserveRange (IdName, Id)
VALUES ('Id#1', 0);
```

You want to use this table to reserve ranges of Id's. You need to:

- read the current Id,
- increment it by a certain amount,
- return the range that has been reserved in a thread-safe manner.

Normally developers wrap the TSQL in a transaction and use a UPDLOCK locking hint to serialize access to the table like this (a couple of variations of this are possible but all require the transaction and a long-running update lock):

```
BEGIN TRAN;

DECLARE @range INT;
SET @range = 100;
DECLARE @ID INT;

-- Get the current value.
SELECT @ID = ID
FROM ReserveRange WITH(updlock)
WHERE IdName = 'Id#1';

-- Reserve a range.
UPDATE ReserveRange
SET ID = @ID + @range
WHERE IdName = 'Id#1';

-- Return the range
SELECT @ID + 1 AS LowRange,
   @ID + @range AS HighRange;

COMMIT TRAN;
GO
```

The trick is to notice that the UPDATE statement supported triple assignments. Have a look at the following:

```
DECLARE @range INT;
SET @range = 100;
DECLARE @ID INT;

-- Update the current range and assign it to
-- a local variable
UPDATE ReserveRange
SET @ID = ID + @range
WHERE IdName = 'Id#1';

-- Return the range
SELECT @ID - @range + 1 AS LowRange,
   @ID AS HighRange;
```

Note: This works only in SQL server 2005

Exercise

1. Write a query using these.
    a. **Union**
    b. **Union ALL**
    c. **Intersect**
    d. **Except**
2. What are sub queries?
3. Explain types of sub queries?
4. What is the use of OUTPUT?
5. Write a query using NTILE.

References

http://www.w3schools.com/sql/default.asp

http://www.1keydata.com/sql/sql.html

http://www.sqlzoo.net

http://support.microsoft.com/support/sql/

http://www.fotia.co.uk/fotia/Default.aspx

http://www.databasejournal.com/features/article.php/3593466

http://sqlcourse.com/intro.html