

## **INDEX**

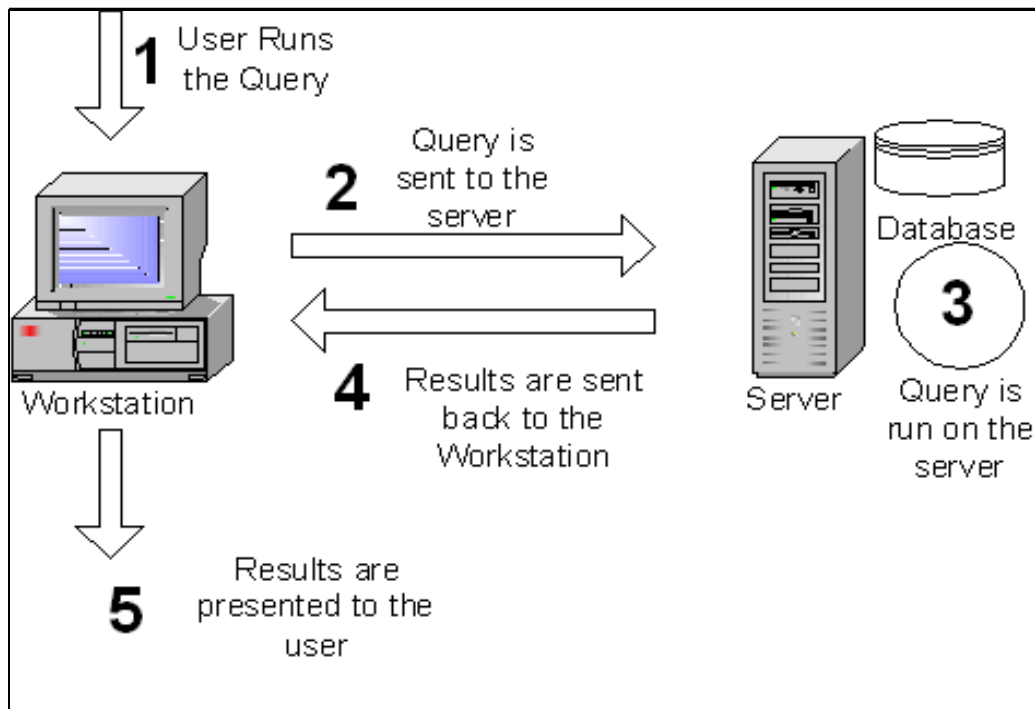
<b>Introduction</b>	<b>3</b>
<b>SQL Server 2008(Overview)</b>	<b>12</b>
<b>Structured Query Language (SQL)</b>	<b>20</b>
<b>Databases</b>	<b>22</b>
<b>Tables</b>	<b>32</b>
<b>Data Types</b>	<b>42</b>
<b>Constraints</b>	<b>47</b>
<b>Rules and Defaults</b>	<b>57</b>
<b>Joins</b>	<b>61</b>
<b>SubQueries</b>	<b>68</b>
<b>Built-in Functions</b>	<b>74</b>
<b>Operators</b>	<b>80</b>
<b>Views</b>	<b>85</b>
<b>Indexes</b>	<b>90</b>

<b>Synonyms</b>	<b>93</b>
<b>Normalization</b>	<b>95</b>
<b>T-SQL Programming</b>	<b>103</b>
<b>Stored procedures</b>	<b>108</b>
<b>Triggers</b>	<b>113</b>
<b>User Defined Functions (UDF)</b>	<b>120</b>
<b>Transactions</b>	<b>125</b>
<b>Locks</b>	<b>128</b>
<b>Cursors</b>	<b>133</b>
<b>Security</b>	<b>136</b>
<b>Back-up and recovery of Databases</b>	<b>140</b>
<b>Case Study</b>	<b>144</b>
<b>More Practical Examples</b>	<b>154</b>
<b>Frequently Asked Questions</b>	<b>166</b>

## INTRODUCTION

SQL Server is a Client/Server Relational Database Management System (RDBMS) that uses Transact-SQL to send request between a client and SQL Server.

### Client/Server Architecture:



SQL Server is designed to be a client/server system. Client/server systems are constructed so that the database can reside on a central computer, known as a *Server*, and be shared among several users. When users want to access the data in SQL Server, they run an application on their local computer, known as a *Client* that connects over a network to the server running SQL Server.

SQL Server can work with thousands of client application simultaneously. The server has features to prevent the logical problems that occur if a user tries to read or modify data currently used by others.

While SQL Server is designed to work as a server in a *Client/Server* network, it is also capable of working as a stand-alone database directly on the client. The scalability and easy-to-use features of SQL Server allow it to work efficiently on a client without consuming too many resources.

To begin with, a brief overview of the relational database model is provided as the SQL Server database is based on this model.

### **Database Management System:**

A **Database Management System (DBMS)** is a set of computer programs that controls the creation, maintenance, and the use of a database. It allows organizations to place control of database development in the hands of database administrators (DBAs) and other specialists.

A DBMS is a system software package that helps the use of integrated collection of data records and files known as databases. It allows different user application programs to easily access the same database.

DBMSs may use any of a variety of database models, such as the Hierarchical model, network model or relational model.

In large systems, a DBMS allows users and other software to store and retrieve data in a structured way. Instead of having to write computer programs to extract information, user can ask simple questions in a query language. Thus, many DBMS packages provide Fourth-generation programming language (4GLs) and other application development features. It helps to specify the logical organization for a database and access and use the information within a database. It provides facilities for controlling data access, enforcing data integrity, managing concurrency, and restoring the database from backups. A DBMS also provides the ability to logically present database information to users.

Database Management systems offers the following services.

#### **Data Definition:**

It is a method of data storage

#### **Data Maintenance:**

It checks whether each record has fields containing all information about one particular item. For example, in an employee table, all information about the employee like name, address, designation, salary, dept-name.

#### **Data Manipulation:**

This method helps in viewing and manipulating data.

#### **Data Integrity:**

This ensures accuracy of data.

## Entity-Relational Model (E-R Model):

Database design representation will be done using E-R model.

### Entity:

An entity is a any object, place, person, concept or activity about which an enterprise records data.

Ex:

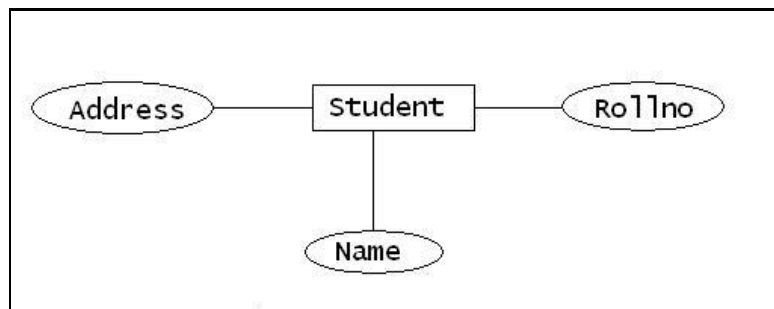
Name	Price	Description
Keyboard	800	Computer Accessories
Mouse	200	Computer Accessories

### Attributes:

An attribute is the characteristic property of an existing entity. Attribute type is the property of entity type, and attribute instance is the property of entity instance.

An ellipse always represents the attribute.

Ex:



### Relationship among data:

A relationship is defined as "an association among entities". A relationship type is an association of entity types. While a relationship instance is an association of entity instances.

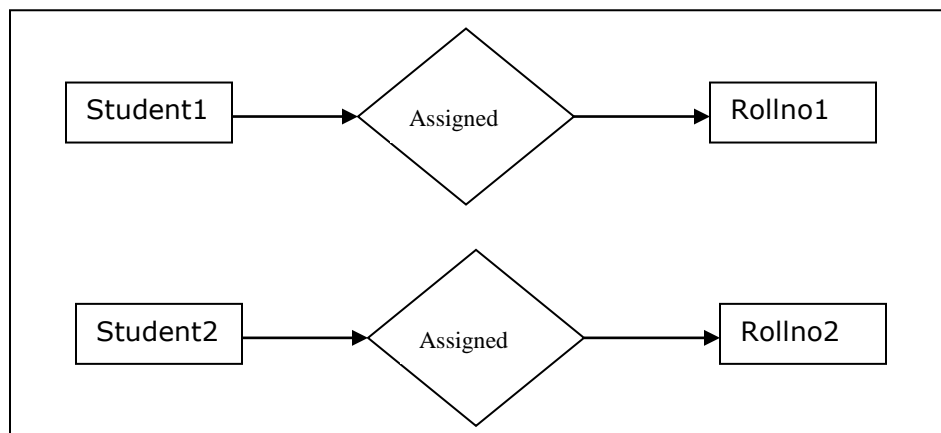
A relationship may associate an entity with itself. Several relationships may exist between the same entities.

The three different types of relationships recognized among various data stored in the database are:

- One-to-one
- One-to-Many( or Many-to-One)
- Many-to-Many

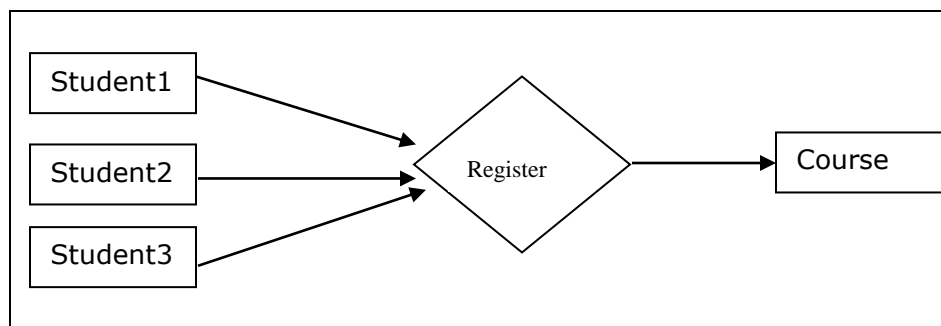
### **One-to-One:**

Consider for example a set of students in a class. Each student can have only one Roll number. Similarly each role number can be associated with on student. This is the case of a one-to-one relationship.



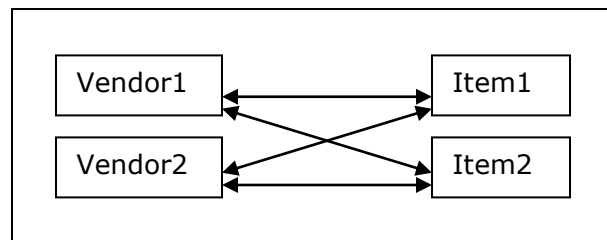
### **Many-to-One:**

One student can register for only one particular course at a time, whereas a number of students could register for the same course.



**Many-to-Many:**

A vendor can sell a number of items and many vendors can sell a particular item.

**Database models:**

A database model is a theory or specification describing how a database is structured and used. Several such models have been suggested.

Common models include:

- File Management System model
- Hierarchical model
- Network model
- Relational model
- Object-relational model

**File Management System Model:**

The File Management model was the first method used to store data in a computerized database. The data item is stored sequentially in one large file. A particular relationship cannot be drawn between the items other than the sequence in which it is stored. If a particular data item has to be located, the search starts at beginning and items are checked sequentially till the required item is found.

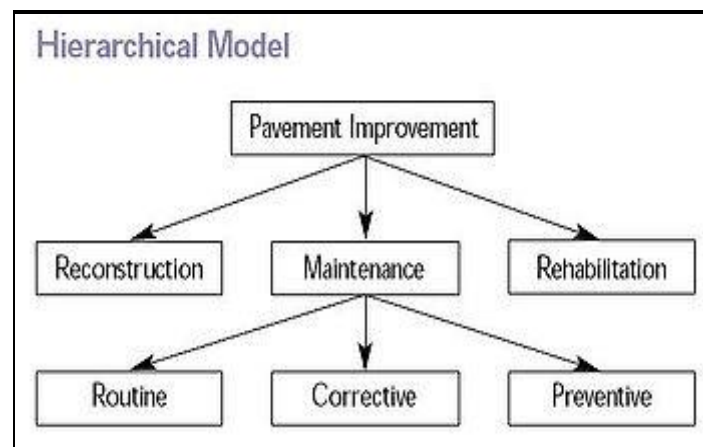
Disadvantages:

Locating and retrieving a record is a very tedious process because of its sequential access method. Altering the file structure such as updating a record is also cumbersome. To do this task the entire file has to be read and rewritten. It involves writing the new data item into a temporary file, adding this item as a last filed of each record, deleting the original file and renaming the temporary file.

**Hierarchical model:**

In this model data storage is in the form of a parent-child relationship. The origin of a data tree is the root. Data located at different levels along a particular branch from the root is called the node. The last node in the leaf is called the leaf. This model supports One-to-Many relationship.

Hierarchical structures were widely used in the early mainframe database management systems, such as the Information Management System (IMS) by IBM, and now describe the structure of XML documents.



Suppose information is required, say 2.1. It is not necessary for the DBMS to search the entire file to locate the data. Instead, it first follows the level2 branch and fetches the data.

**Disadvantages:**

It is not possible to enter a new level into the system. As and when such a need arises the entire structure has to be revamped. Another disadvantage is that this model does not support Many-to-Many relationship.

**Network model:**

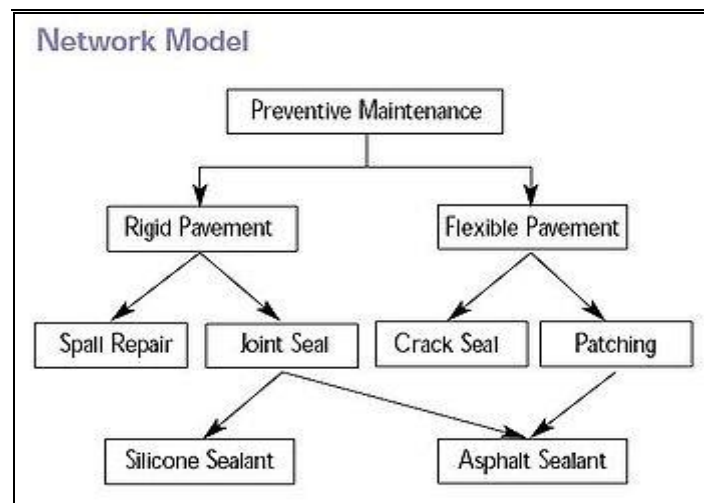
The network model (defined by the CODASYL specification) organizes data using two fundamental constructs, called *records* and *sets*. Records contain fields (which may be organized hierarchically, as in the programming language COBOL). Sets (not to be confused with mathematical sets) define one-to-many relationships between records: one owner, many members. A record may be an owner in any number of sets, and a member in any number of sets.



The network model is a variation on the hierarchical model, to the extent that it is built on the concept of multiple branches (lower-level structures) emanating from one or more nodes (higher-level structures), while the model differs from the hierarchical model in that branches can be connected to multiple nodes. The network model is able to represent redundancy in data more efficiently than in the hierarchical model.

The operations of the network model are navigational in style: a program maintains a current position, and navigates from one record to another by following the relationships in which the record participates. Records can also be located by supplying key values.

Although it is not an essential feature of the model, network databases generally implement the set relationships by means of pointers that directly address the location of a record on disk. This gives excellent retrieval performance, at the expense of operations such as database loading and reorganization.



**Disadvantages:**

The use of pointers leads to complexity in the structure. As a result of the increased complexity mapping of related data becomes very difficult.

## Relational model:

The relational model was introduced by E.F.Codd in 1970 as a way to make database management systems more independent of any particular application. It is a mathematical model defined in terms of predicate logic and set theory.

The products that are generally referred to as relational databases in fact implement a model that is only an approximation to the mathematical model defined by Codd.

Three key terms are used extensively in relational database models: **relations**, **attributes**, and **domains**.

A relation is a table with columns and rows. The named columns of the relation are called attributes, and the domain is the set of values the attributes are allowed to take.

The basic data structure of the relational model is the table, where information about a particular entity (say, an employee) is represented in rows (also called tuples) and columns. Thus, the "relation" in "relational database" refers to the various tables in the database; a relation is a set of tuples. The columns enumerate the various attributes of the entity (the employee's name, address or phone number, for example), and a row is an actual instance of the entity (a specific employee) that is represented by the relation. As a result, each tuple of the employee table represents various attributes of a single employee.

***All relations (and, thus, tables) in a relational database have to adhere to some basic rules to qualify as relations. First, the ordering of columns is immaterial in a table. Second, there can't be identical tuples or rows in a table. And third, each tuple will contain a single value for each of its attributes.***

A relational database contains multiple tables, each similar to the one in the "flat" database model. One of the strengths of the relational model is that, in principle, any value occurring in two different records (belonging to the same table or to different tables), implies a relationship among those two records. Yet, in order to enforce explicit integrity constraints, relationships between records in tables can also be defined explicitly, by identifying or non-identifying parent-child relationships characterized by assigning cardinality (1:1, (0)1:M, M:M). Tables can also have a designated single attribute or a set of attributes that can act as a "key", which can be used to uniquely identify each tuple in the table.

A key that can be used to uniquely identify a row in a table is called a primary key. Keys are commonly used to join or combine data from two or more tables. For example, an *Employee* table may contain a column named *Location* which contains a value that matches the key of a *Location* table. Keys are also critical in the creation of indexes, which facilitate fast retrieval of data from large tables. Any column can be a key, or multiple columns can be

grouped together into a compound key. It is not necessary to define all the keys in advance; a column can be used as a key even if it was not originally intended to be one.

VendorDetails		OrderDetails		
Vno	Company	ProdNo	ItemDesc	Vno
V1	Comp1	P1	Item1	V1
V2	Comp2	P2	Item2	V3
V3	Comp3	P3	Item3	V2
V4	Comp4	P4	Item4	V4

### Object-Relational Model:

An **object-relational database** (ORD), or **object-relational database management system** (ORDBMS), is a database management system (DBMS) similar to a relational database, but with an object-oriented database model: objects, classes and inheritance are directly supported in database schemas and in the query language. In addition, it supports extension of the data model with custom data-types and methods.

## SQL SERVER 2008

**SQL Server 2008** aims to make data management self-tuning, self organizing, and self maintaining with the development of *SQL Server Always On* technologies, to provide near-zero downtime. SQL Server 2008 also includes support for structured and semi-structured data, including digital media formats for pictures, audio, video and other multimedia data. In current versions, such multimedia data can be stored as BLOBs (binary large objects), but they are generic bitstreams. Intrinsic awareness of multimedia data will allow specialized functions to be performed on them.

SQL Server 2008 can be a data storage backend for *different varieties of data*: *XML, email, time/calendar, file, document, spatial, etc* as well as perform *search, query, analysis, sharing, and synchronization* across all data types.

Other new data types include specialized date and time types and a *Spatial* data type for location-dependent data.

Better support for unstructured and semi-structured data is provided using the new *FILESTREAM* data type, which can be used to reference any file stored on the file system.

Structured data and metadata about the file is stored in SQL Server database, whereas the unstructured component is stored in the file system. Such files can be accessed both via Win32 file handling APIs as well as via SQL Server using T-SQL;

The Full-Text Search functionality has been integrated with the database engine, which simplifies management and improves performance.

SQL Server includes better compression features, which also helps in improving scalability.

SQL Server 2008 supports the ADO.NET Entity Framework and the reporting tools, replication, and data definition will be built around the Entity Data Model. SQL Server Reporting Services will gain charting capabilities from the integration of the data visualization.

The version of SQL Server Management Studio included with SQL Server 2008 supports **IntelliSense** for SQL queries against a SQL Server 2008 Database Engine.

## **SQL Server 2008 R2:**

SQL Server 2008 R2 (formerly codenamed *SQL Server "Kilimanjaro"*) was announced at TechEd 2009, and was released to manufacturing on April 21, 2010. SQL Server 2008 R2 adds certain features to SQL Server 2008 including **master data management** system branded as *Master Data Services*, a centralized console to manage multiple SQL Server instances, and support for more than 64 logical processors.

SQL Server 2008 R2 boasts a number of new services, including PowerPivot for Excel and SharePoint, Master Data Services, StreamInsight, ReportBuilder 3.0, Reporting Services Add-in for SharePoint, a Data-tier function in Visual Studio that enables packaging of tiered databases as part of an application, and a SQL Server Utility named UC (Utility Control POint), part of AMSM (Application and Multi-Server Management) that is used to manage multiple SQL Servers.

## **Editions of SQL Server 2008:**

Microsoft makes SQL Server available in multiple editions, with different feature sets and targeting different users. These editions are:

### **SQL Server Compact Edition (SQL CE):**

The compact edition is an embedded database engine. Unlike the other editions of SQL Server, the SQL CE engine is based on SQL Mobile (initially designed for use with hand-held devices) and does not share the same binaries. The 3.5 version includes considerable work that supports ADO.NET Synchronization Services.

### **SQL Server Evaluation Edition:**

SQL Server Evaluation Edition, also known as the *Trial Edition*, has all the features of the Enterprise Edition, but is limited to 180 days, after which the tools will continue to run, but the server services will stop.<sup>[24]</sup>

### **SQL Server Express Edition:**

SQL Server Express Edition is a scaled down, free edition of SQL Server, which includes the core database engine. While there are no limitations on the number of databases or users supported, it is limited to using one processor, 1 GB memory and 4 GB database files (10 GB database files from SQL Server Express 2008 R2).

**SQL Server Fast Track:**

SQL Server Fast Track is specifically for enterprise-scale data warehousing storage and business intelligence processing, and runs on reference-architecture hardware that is optimized for Fast Track.

**SQL Server Standard Edition:**

SQL Server Standard edition includes the core database engine, along with the stand-alone services. It differs from Enterprise edition in that it supports fewer active instances (number of nodes in a cluster) and does not include some high-availability functions such as hot-add memory (allowing memory to be added while the server is still running), and parallel indexes.

**SQL Server Developer Edition:**

SQL Server Developer Edition includes the same features as SQL Server Enterprise Edition, but is limited by the license to be only used as a development and test system, and not as production server. This edition is available to download by students free of charge as a part of Microsoft's Dreams park program.

**SQL Server Enterprise Edition:**

SQL Server Enterprise Edition is the full-featured edition of SQL Server, including both the core database engine and add-on services, while including a range of tools for creating and managing a SQL Server cluster.

**SQL Server Web Edition:**

SQL Server Web Edition is a low-TCO option for Web hosting.

**SQL Server Workgroup Edition:**

SQL Server Workgroup Edition includes the core database functionality but does not include the additional services.

## Tools:

### SQLCMD:

SQLCMD is a command line application that comes with Microsoft SQL Server, and exposes the management features of SQL Server. It allows SQL queries to be written and executed from the command prompt. It can also act as a scripting language to create and run a set of SQL statements as a script. Such scripts are stored as a .sql file, and are used either for management of databases or to create the database schema during the deployment of a database.

SQLCMD was introduced with SQL Server 2005 and this continues with SQL Server 2008. Its predecessor for earlier versions was OSQL and ISQL, which is functionally equivalent as it pertains to TSQL execution, and many of the command line parameters are identical, although SQLCMD adds extra versatility.

### Visual Studio:

**Microsoft Visual Studio** includes native support for data programming with Microsoft SQL Server. It can be used to write and debug code to be executed by SQL CLR. It also includes a *data designer* that can be used to graphically create, view or edit database schemas. Queries can be created either visually or using code. SSMS 2008 onwards, provides **intellisense** for SQL queries as well.

### SQL Server Management Studio:

**SQL Server Management Studio** is a GUI tool included with SQL Server 2005 and later for configuring, managing, and administering all components within Microsoft SQL Server. The tool includes both script editors and graphical tools that work with objects and features of the server.

SQL Server Management Studio replaces Enterprise Manager as the primary management interface for Microsoft SQL Server since SQL Server 2005. A version of SQL Server Management Studio is also available for SQL Server Express Edition, for which it is known as *SQL Server Management Studio Express* (SSMSE).

A central feature of SQL Server Management Studio is the Object Explorer, which allows the user to browse, select, and act upon any of the objects within the server.<sup>[56]</sup> It can be used to visually observe and analyze query plans and optimize the database performance, among others.

SQL Server Management Studio can also be used to create a new database, alter any existing database schema by adding or modifying tables and

indexes, or analyze performance. It includes the query windows which provide a GUI based interface to write and execute queries.

### **Business Intelligence Development Studio:**

**Business Intelligence Development Studio (BIDS)** is the IDE from Microsoft used for developing data analysis and Business Intelligence solutions utilizing the Microsoft SQL Server Analysis Services, Reporting Services and Integration Services. It is based on the Microsoft Visual Studio development environment but customizes with the SQL Server services-specific extensions and project types, including tools, controls and projects for reports (using Reporting Services), Cubes and data mining structures (using Analysis Services).

### **SQL Server Services:**

SQL Server also includes an assortment of add-on services. While these are not essential for the operation of the database system, they provide value added services on top of the core database management system. These services either run as a part of some SQL Server component or out-of-process as Windows Service and presents their own API to control and interact with them.

#### **Service Broker:**

Used inside an instance, it is used to provide an asynchronous programming environment. For cross instance applications, Service Broker communicates The Service Broker, which runs as a part of the database engine, provides a reliable messaging and message queuing platform for SQL Server applications.

#### **Replication Services:**

SQL Server Replication Services are used by SQL Server to replicate and synchronize database objects, either in entirety or a subset of the objects present, across replication agents, which might be other database servers across the network, or database caches on the client side. Replication follows a publisher/subscriber model, i.e., the changes are sent out by one database server ("publisher") and are received by others ("subscribers"). SQL Server supports three different types of replication:

##### **Transaction replication**

Each transaction made to the publisher database (master database) is synced out to subscribers, who update their databases with



the transaction. Transactional replication synchronizes databases in near real time.

### **Merge replication**

Changes made at both the publisher and subscriber databases are tracked, and periodically the changes are synchronized bi-directionally between the publisher and the subscribers. If the same data has been modified differently in both the publisher and the subscriber databases, synchronization will result in a conflict which has to be resolved - either manually or by using pre-defined policies.

### **Snapshot replication**

Snapshot replication publishes a copy of the entire database (the then-snapshot of the data) and replicates out to the subscribers. Further changes to the snapshot are not tracked.

## **Analysis Services:**

SQL Server Analysis Services adds OLAP and data mining capabilities for SQL Server databases. The OLAP engine supports MOLAP, ROLAP and HOLAP storage modes for data. Analysis Services supports the XML for Analysis standard as the underlying communication protocol. The cube data can be accessed using MDX queries. Data mining specific functionality is exposed via the DMX query language.

## **Reporting Services:**

SQL Server Reporting Services is a report generation environment for data gathered from SQL Server databases. It is administered via a web interface. Reporting services features a web services interface to support the development of custom reporting applications. Reports are created as RDL files.

Reports can be designed using recent versions of Microsoft Visual Studio (Visual Studio.NET 2003, 2005, and 2008) with Business Intelligence Development Studio, installed or with the included Report Builder. Once created, RDL files can be rendered in a variety of formats including Excel, PDF, CSV, XML, TIFF (and other image formats), and HTML Web Archive.

## **Notification Services:**

Originally introduced as a post-release add-on for SQL Server 2000, Notification Services was bundled as part of the Microsoft SQL Server platform for the first and only time with SQL Server 2005. With SQL Server 2005, SQL Server Notification Services is a mechanism for generating data-driven

notifications, which are sent to Notification Services subscribers. A subscriber registers for a specific event or transaction (which is registered on the database server as a trigger); when the event occurs, Notification Services can use one of three methods to send a message to the subscriber informing about the occurrence of the event. These methods include SMTP, SOAP, or by writing to a file in the file system. Notification Services was discontinued by Microsoft with the release of SQL Server 2008 in August 2008, and is no longer an officially supported component of the SQL Server database platform.

### **Integration Services:**

SQL Server Integration Services is used to integrate data from different data sources. It is used for the ETL capabilities for SQL Server for data warehousing needs. Integration Services includes GUI tools to build data extraction workflows integration various functionality such as extracting data from various sources, querying data, transforming data including aggregating, duplication and merging data, and then loading the transformed data onto other sources, or sending e-mails detailing the status of the operation as defined by the user.

### **Full Text Search Service:**

SQL Server Full Text Search service is a specialized indexing and querying service for unstructured text stored in SQL Server databases. The full text search index can be created on any column with character based text data. It allows for words to be searched for in the text columns. While it can be performed with the SQL LIKE operator, using SQL Server Full Text Search service can be more efficient. Full Text Search (FTS) allows for inexact matching of the source string, indicated by a *Rank* value which can range from 0 to 1000 - a higher rank means a more accurate match. T-SQL exposes special operators that can be used to access the FTS capabilities.

The Full Text Search engine is divided into two processes - the *Filter Daemon* process (msftefd.exe) and the *Search* process (msftesql.exe). These processes interact with the SQL Server. The Search process includes the indexer (that creates the full text indexes) and the full text query processor. The indexer scans through text columns in the database.

When a full text query is received by the SQL Server query processor, it is handed over to the FTS query processor in the Search process. The FTS query processor breaks up the query into the constituent words, filters out the noise words, and uses an inbuilt thesaurus to find out the linguistic variants for each word. The words are then queried against the inverted index and a rank of their accurateness is computed. The results are returned to the client via the SQL Server process.

## **Programmability:**

### **T-SQL:**

T-SQL (Transact-SQL) is the primary means of programming and managing SQL Server. It exposes keywords for the operations that can be performed on SQL Server, including creating and altering database schemas, entering and editing data in the database as well as monitoring and managing the server itself. Client applications, both which consume data or manage the server, leverage SQL Server functionality by sending T-SQL queries and statements which are then processed by the server and results (or errors) returned to the client application. SQL Server allows it to be managed using T-SQL. For this it exposes read only tables from which server statistics can be read. Management functionality is exposed via system-defined stored procedures which can be invoked from T-SQL queries to perform the management operation. It is also possible to create linked Server using T-SQL. Linked server allows operation to multiple server as one query.

### **SQL Native Client:**

SQL Native Client is the native client side data access library for Microsoft SQL Server, version 2005 onwards. It natively implements support for the SQL Server features including the Tabular Data Stream implementation, support for mirrored SQL Server databases, full support for all data types supported by SQL Server, asynchronous operations, query notifications, encryption support, as well as receiving multiple result sets in a single database session. SQL Native Client is used under the hood by SQL Server plug-ins for other data access technologies, including ADO or OLE DB. The SQL Native Client can also be directly used, bypassing the generic data access layers.

## **STRUCTURED QUERY LANGUAGE**

SQL is a standard that every database should follow. That means all the SQL commands should work in all databases. Based on the type of operations we are performing on the database, SQL is divided into the following types:

- Data Definition Language (DDL)
- Data Manipulation Language (DML)
- Transaction Control Language (TCL)
- Data Control Language (DCL)

### **Data Definition Language:**

It is used to create an object (e.g. table), alter the structure of an object and also to delete the object from the server. All these commands will effect only the structure of the database object. The commands contained by this language are:

- CREATE - used to create an object
- ALTER - used to modify an existing object
- DROP - used to delete the object
- TRUNCATE - used to delete only the data of an object

Note: DDL commands are AUTO COMMIT. i.e., with the execution of these commands, the results will be stored directly into the database.

### **Data Manipulation Language:**

The DML commands are most frequently used SQL commands. They are used to query and manipulate existing objects like tables. They should affect only the data of an existing structure. The commands existing in this language are:

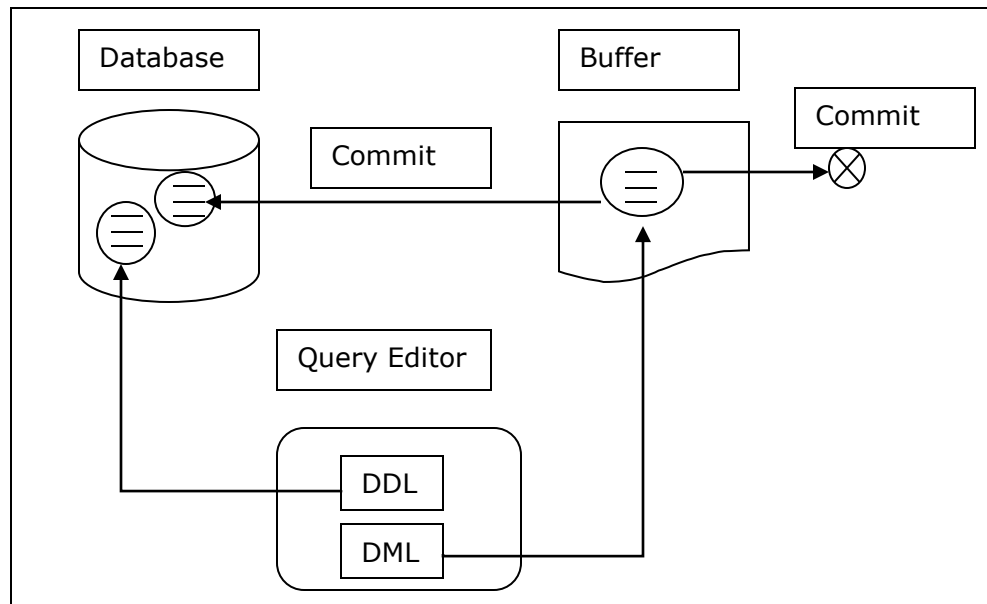
- SELECT - used to query the object data
- INSERT - used to enter the data into the object
- UPDATE - used to modify existing objects data
- DELETE - used to delete the data of an object.

## Transaction Control Language:

A transaction is a logical unit of work. All changes made to the database can be referred to as a transaction.

Whenever, we perform a DML operation, results will be stored in the buffer not in the database. The user can control Transaction changes with the use of this language commands. They are:

- COMMIT - used to make the changes permanent
- ROLLBACK - used to undo the changes



## Data Control Language:

In general to access an object which was created by another user, at first, we must get the permission from the owner of the object. To provide the permission or remove the permission, we use this language commands. They are:

- GRANT - Used to provide Permission
- REVOKE - Used to remove the Permission

# DATABASES

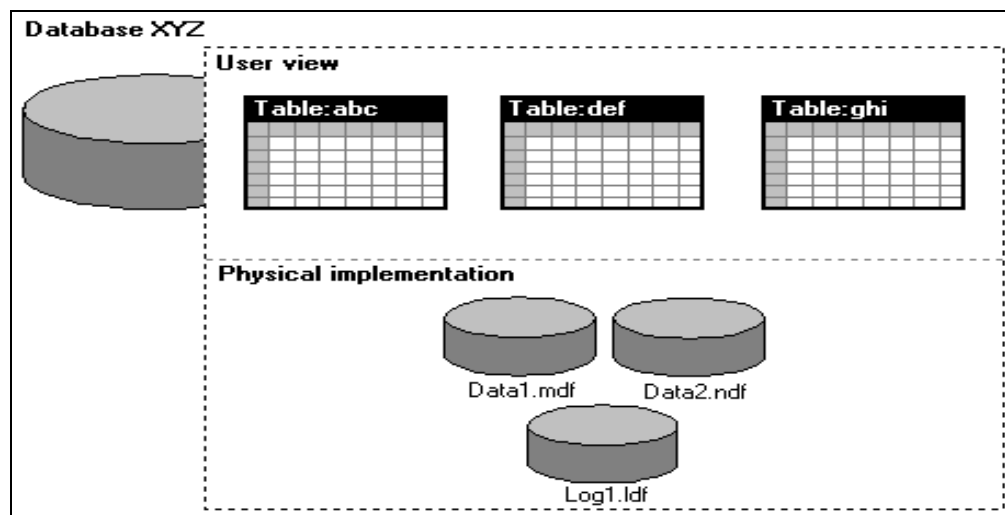
A database in Microsoft SQL Server consists of a collection of tables that contain data and other objects, such as views, indexes, stored procedures, and triggers, defined to support activities performed with the data. The data stored in a database is usually related to a particular subject or process, such as inventory information for a manufacturing warehouse.

SQL Server can support many databases. Each database can store either interrelated or unrelated data from other databases. For example, a server can have one database that stores personnel data and another that stores product-related data. Alternatively, one database can store current customer order data, and another related database can store historical customer orders used for yearly reporting.

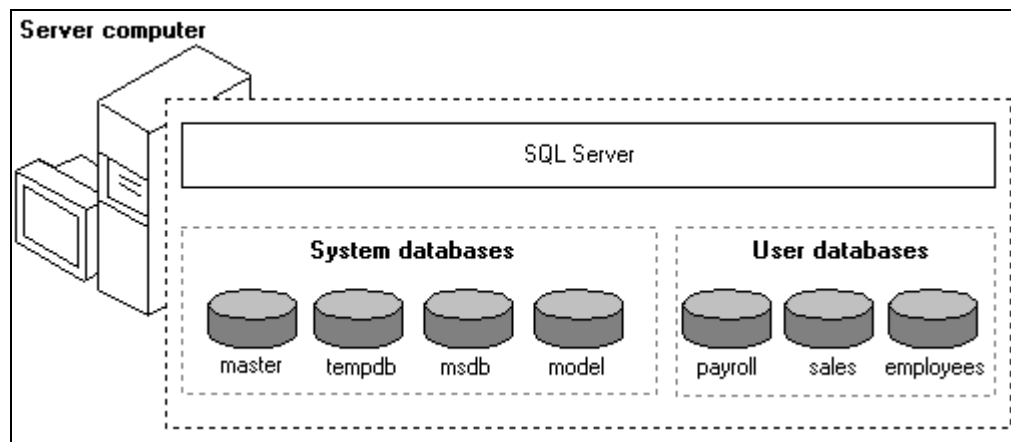
## Database Architecture:

Microsoft SQL Server 2000 data is stored in databases. The data in a database is organized into the logical components visible to users. A database is also physically implemented as two or more files on disk.

When using a database, you work primarily with the logical components such as tables, views, procedures, and users. The physical implementation of files is largely transparent. Typically, only the database administrator needs to work with the physical implementation.



Each instance of SQL Server has four system databases (**master**, **model**, **tempdb**, and **msdb**) and one or more user databases. Some organizations have only one user database, containing all the data for their organization. Some organizations have different databases for each group in their organization, and sometimes a database used by a single application. For example, an organization could have one database for sales, one for payroll, one for a document management application, and so on. Sometimes an application uses only one database; other applications may access several databases.



It is not necessary to run multiple copies of the SQL Server database engine to allow multiple users to access the databases on a server. An instance of the SQL Server Standard or Enterprise Edition is capable of handling thousands of users working in multiple databases at the same time. Each instance of SQL Server makes all databases in the instance available to all users that connect to the instance, subject to the defined security permissions.

### Types of Databases:

SQL Server provides two types of Databases.

- System Databases
- User Defined Databases

## System Databases:

Along with the installation of SQL Server **FOUR** databases will be created automatically called as system databases. They can be used by any user anywhere from SQL Server.

The System Databases are:

- **master**

The **master** database records all of the system level information for a SQL Server system. It records all login accounts and all system configuration settings. **master** is the database that records the existence of all other databases, including the location of the database files. **master** records the initialization information for SQL Server; always have a recent backup of **master** available.

- **model**

The **model** database is used as the template for all databases created on a system. When a CREATE DATABASE statement is issued, the first part of the database is created by copying in the contents of the **model** database, then the remainder of the new database is filled with empty pages. Because **tempdb** is created every time SQL Server is started, the **model** database must always exist on a SQL Server system.

- **msdb**

The **msdb** database is used by SQL Server Agent for scheduling alerts and jobs, and recording operators.

- **tempdb**

**tempdb** holds all temporary tables and temporary stored procedures. It also fills any other temporary storage needs such as work tables generated by SQL Server. **tempdb** is a global resource; the temporary tables and stored procedures for all users connected to the system are stored there. **tempdb** is re-created every time SQL Server is started so the system starts with a clean copy of the database

## User Defined Databases:

The databases that are created by the users are called as user defined databases.

While creating a database, the user has to specify a name to the database and the details of data files, which are used to specify the location, where the database data is to be stored.



## Files and Filegroups:

Microsoft® SQL Server™ 2000 maps a database using a set of operating-system files. All data and objects in the database, such as tables, stored procedures, triggers, and views, are stored within these operating-system files:

- Primary Data File (.mdf)  
This file contains the startup information for the database and is used to store data. Every database has one primary data file.
- Secondary Data File (.ndf)  
These files hold all of the data that does not fit in the primary data file. If the primary file can hold all of the data in the database, databases do not need to have secondary data files. Some databases may be large enough to need multiple secondary data files or to use secondary files on separate disk drives to spread data across multiple disks.
- Transaction Log File (.ldf)  
These files hold the log information used to recover the database. There must be at least one log file for each database.

Filegroups allow files to be grouped together for administrative and data allocation/placement purposes. For example, three files (Data1.ndf, Data2.ndf, and Data3.ndf) can be created on three disk drives, respectively, and assigned to the filegroup **fgroup1**. A table can then be created specifically on the filegroup **fgroup1**. Queries for data from the table will be spread across the three disks, thereby improving performance. Files and filegroups, however, allow you to easily add new files on new disks. Additionally, if your database exceeds the maximum size for a single Microsoft Windows NT® file, you can use secondary data files to allow your database to continue to grow.

## Creating a Database:

To create a database, determine the name of the database, its owner (the user who creates the database), its size, and the files and filegroups used to store it.

Before creating a database, consider that:

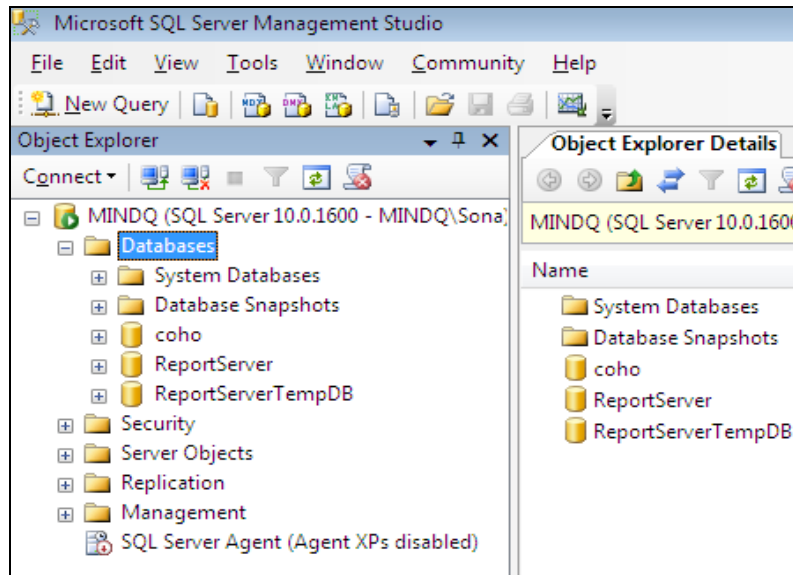
- Permission to create a database defaults to members of the **sysadmin** and **dbcreator** fixed server roles.
- The user who creates the database becomes the owner of the database.
- A maximum of 32,767 databases can be created on a server.
- The name of the database must follow the rules for identifiers.

## Creating Database using Navigation:

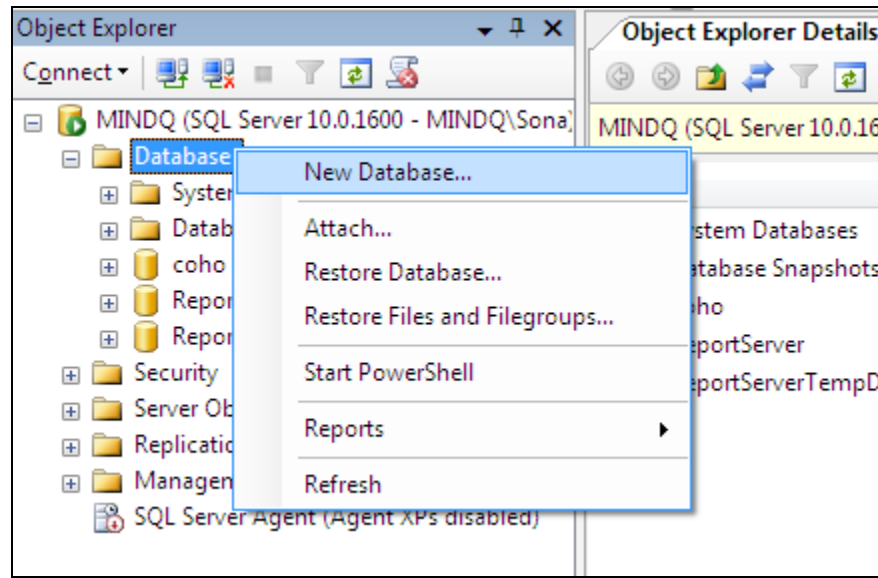
1. Open the **'Management Studio'** tool and connect to the required instance of SQL Server with desired authentication.



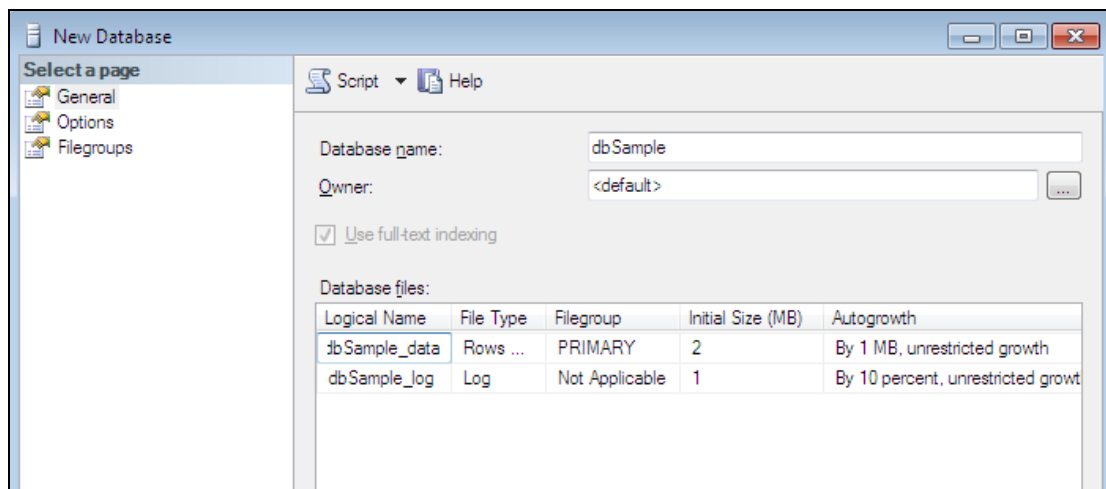
2. Expand '**Databases**' Item. It displays list of already existing databases.



3. Right Click on the '**Databases**' item and select '**New Database**' option.



4. Provide a name to the Database and specify the details of primary and log file and '**Database Files**' section.



## Creating a Database Using a Query:

Once the Query Analyzer is opened, in the command window we can write any command that has to be executed.

### Syntax:

```
CREATE DATABASE databasename
[ON]
(NAME=<logical file name>,
FILENAME=<os-filename>,
SIZE=<in mb's>,MAXSIZE=<in mb's>,FILEGROWTH=<in mb's>)]
[LOG ON]
(NAME=<logical file name>,
FILENAME=<os-filename>,
SIZE=<in mb's>,MAXSIZE=<in mb's>,FILEGROWTH=<in mb's>)]
```

**Note:** '*Logicalfilename*' is used to identify the file for further modifications and '*os-filename*' refers the original location where the file is stored in the hard disk.

### Example: 1. Creating a database without specifying Data Files

```
CREATE DATABASE dbNew
```

Note: Creates a database called 'dbNew' with a size of 1.68MB. It internally contains a default primary file of 1216KB and log file of 504KB.

### Example: 2. Creating a database Data Files

```
CREATE DATABASE dbSample
ON
(NAME='dbSampleData',
FILENAME='d:\backup\data\dbSample_data.mdf',
SIZE=20,MAXSIZE=50,FILEGROWTH=5)
LOG ON
(NAME=dbSampleLog',
FILENAME='d:\backup\log\dbSample_log.ldf',
SIZE=10,MAXSIZE=25,FILEGROWTH=5)
```

## Modifying the Database:

The changes that can be made to existing database are

- The data and log files can be added or removed.
- The database can be made to expand or shrink
- File groups can be added and removed from the database.
- The configuration settings for the database can be changed

The database can be expanded by allocating additional space to the existing database file or by allocating space to a new file. The size of the database must be increased by at least 1mb.

A maximum size should be specified using the MAXSIZE parameter of the ALTER DATABASE statement. This prevents the file from growing until the disk space is exhausted.

### Syntax:

```
ALTER DATABASE <database Name>
ADD FILE <file specifications>
|ADD LOG FILE <file specifications>
|REMOVE FILE <file name>
|ADD FILEGROUP <file group name>
|REMOVE FILEGROUP <file group name>
|MODIFY FILE <file specifications>
|MODIFY FILEGROUP <file group name>
```

### Example:

```
ALTER DATABASE dbSample
ADD FILE
(NAME= 'dbSampledata1',
FILE NAME='d:\backup\data\dbSample_data1.ndf',
SIZE=10mb,
MAXSIZE=30mb,
FILEGROWTH=1mb)
```

### **To add a file group:**

```
ALTER DATABASE dbSample  
ADD FILEGROUP frgp1
```

### **To add a secondary file in the file group:**

```
ALTER DATABASE dbSample  
ADD FILE  
(NAME= 'dbsample2_dat',  
FILE_NAME='d:\backup\data\dbSample2_dat.mdf',  
SIZE=4mb,MAXSIZE=30mb,FILEGROWTH=1mb)  
TO FILEGROUP frgp1
```

### **Viewing The Database:**

The information regarding the database such as owner, size, date and time of creation, status can be viewed using the following System Stored Procedure:

#### **Syntax:**

```
Sp_helpdb 'database name'
```

```
Example:  
sp_helpdb emp
```

Database options can be changed or displayed using:

#### **Syntax:**

```
Sp_dboption 'database name', 'option name', TRUE/FALSE
```

Different options that can be set to database are:

*Dbuseonly* – only owner of the database can use it.

*Offline* – the database is offline when this option is set.

*Readonly* – data can only read in the database

*Select into/bulk copy* – faster bulk copy

*Single user* – only one user can use it

Example:

```
sp_dboption 'dbSample', 'readonly', TRUE
```

### **Renaming the database:**

The name of the database can be changed as:

#### **Syntax:**

```
sp_renamedb 'old name', 'new name'  
Example:  
sp_renamedb 'dbSample ', 'SampleDB'
```

### **Deleting the database:**

#### **Syntax:**

```
DROP DATABASE <database name>
```

```
Example:  
DROP DATABASE dbSample
```

## TABLES

Tables are database objects that contain all the data in a database. A table definition is a collection of columns. In tables, data is organized in a row-and-column format similar to a spreadsheet. Each row represents a unique record, and each column represents a field within the record.

After you have designed the database, the tables that will store the data in the database can be created. The data is usually stored in permanent tables. Tables are stored in the database files until they are deleted and are available to any user who has the appropriate permissions.

### Types of Tables:

Corresponding to the type of data we are storing inside the tables, they are divided into the following ways:

- System Tables
- User Tables
- Temporary Tables

### System Tables:

These are the tables that will come along with the SQL Server database installation. All these tables will be stored under Master database. They will provide System level information. Ex: sysobjects, sysusers etc.,

### User Tables:

These are the tables that are created by a user. The user who creates the table will be the owner for that table. In order to get an access to that table, other users must get corresponding permission from the owner of the table.

### Temporary Tables

You can also create temporary tables. Temporary tables are similar to permanent tables, except temporary tables are stored in **tempdb** and are deleted automatically when no longer in use. They will be identified by “#” symbol.

### Table Properties

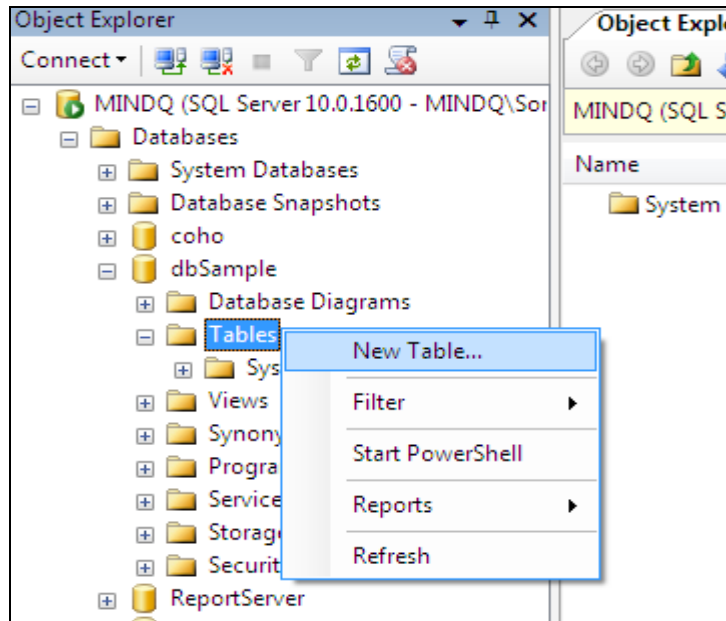
You can define up to 1,024 columns per table. Table and column names must follow the rules for identifiers; they must be unique within a given table, but you can use the same column name in different tables in the same database. You must also define a data type for each column.

Although table names must be unique for each owner within a database, you can create multiple tables with the same name if you specify different owners for each.



### Creating a Table using Navigation:

1. Open the '**Management Studio**' tool and navigate up to the database where we want to create the table.
2. Expand '**Tables**' item that displays list of existing tables.
3. Right click on 'Tables' item and select '**New Table**' option.



4. In the 'Data Grid' displayed enter the details of columns such as – column name, data type, size.

MINDQ.dbSample - dbo.Table_1* Object Explorer Details			
	Column Name	Data Type	Allow Nulls
	prodid	int	<input checked="" type="checkbox"/>
	prodname	char(10)	<input checked="" type="checkbox"/>
▶			<input type="checkbox"/>

5. Click on the 'Save' icon of the toolbar to provide tablename.

## Creating a Table using Query:

We can make use of the following syntax to create a table:

```
CREATE TABLE tablename  
(colname datatype (size) [NOT NULL | DEFAULT <constant  
value> | IDENTITY(seed,increment)], colname2 .....)
```

### NULL | NOT NULL:

Are keywords that determine if null values are allowed in the column. NULL is not strictly a constraint but can be specified in the same manner as NOT NULL.

### DEFAULT:

Specifies the value to be provided for the column when a value is not explicitly supplied during an insert.

*constant\_expression*

Is a constant, NULL, or a system function used as the default value for the column.

### IDENTITY:

Indicates that the new column is an identity column. When a new row is added to the table, Microsoft® SQL Server™ provides a unique, incremental value for the column. The IDENTITY property can be assigned to **tinyint**, **smallint**, **int**, **bigint**, **decimal(p,0)**, or **numeric(p,0)** columns. Only one identity column can be created per table. Bound defaults and DEFAULT constraints cannot be used with an identity column. You must specify both the seed and increment or neither. If neither is specified, the default is (1,1).

#### ***seed***

Is the value used for the very first row loaded into the table.

#### ***increment***

Is the incremental value added to the identity value of the previous row loaded.

- A table can have only one column defined with the IDENTITY property
- The seed and increment can be specified. The default for both is 1
- The identifier column must not allow NULL values and must not contain DEFAULT definition
- In order to insert the values explicitly into the identity column , SET IDENTITY\_INSERT option is used.

### Example:

1. Select the corresponding database in which we want to create the table from the available drop down button.
2. Type the following command in the query window.

```
CREATE TABLE employee
( empid VARCHAR(5),
  Ename char(10), basic_sal int DEFAULT 10000
)
```

3. click on Execute button to run the command.

### Example 2: Crating a table with identity column

```
CREATE TABLE dept
( dno int identity (10,1),
  dept_name varchar(20)
)
```

Syntax:

```
SET IDENTITY_INSERT tablename ON | OFF
```

### Modifying Tables:

After a table is created, you can change many of the options that were defined for the table when it was originally created, including:

- Columns can be added, modified, or deleted. For example, the column name, length, data type, precision, scale, and nullability can all be changed, although some restrictions exist. For more information.
- PRIMARY KEY and FOREIGN KEY constraints can be added or deleted.
- UNIQUE and CHECK constraints and DEFAULT definitions (and objects) can be added or deleted.
- An identifier column can be added or deleted using the IDENTITY or ROWGUIDCOL property.

## Renaming a Table:

### Syntax:

```
SP_RENAME <old_tableName>,<new_tableName>
```

### Example:

```
SP_RENAME employee.emp
```

## Renaming a Column:

### Syntax:

```
SP_RENAME <' tableName.columnname'>,<'newcolName'>
```

### Example:

```
SP_RENAME 'emp.basic_salary','sal'
```

## Syntax for ALTER TABLE command:

```
ALTER TABLE tablename  
ADD columnName datatype [not null |identity|default...]  
|ALTER COLUMN <column specifications>  
|DROP COLUMN columnName  
|ADD CONSTRAINT constraintName {CHECK(..) | UNIQUE | PRIMARY  
KEY..}  
|DROP CONSTRAINT constraintName
```

## Examples:

### Changing datatype

```
ALTER TABLE emp ALTER COLUMN empid int
```

### Changing NULLs

```
ALTER TABLE emp ALTER COLUMN empid int NOT NULL
```

## Adding Columns

```
ALTER TABLE emp ADD dno int
```

## Sp\_help:

This System Stored Procedure is used to provide description about the table structure.

Example:

```
SP_HELP emp
```

## Entering Table Data:

The **Insert** command is used to add rows to a table. If we are entering entire table data then it is optional to specify the column list. Where as if we are entering only particular column details or the order of columns then we must specify the column list.

## Syntax:

```
INSERT INTO <table_name>[ (column_list)] VALUES (data_list)
```

## Example:

```
1. INSERT INTO emp (empid,ename,sal,dno)
   VALUES (1,'E1',25000,20)
2. INSERT INTO emp values(2,'E2',30000,10)
```

- If we want to insert only certain columns data into the table, the column names have to be specified.

## Example:

```
3. INSERT INTO emp ( empid,ename,dno) VALUES(3,'E3',20)
```

- It is also possible to enter multiple records at a time into the table.

```
4. INSERT INTO emp
   Select 4,'E4',15000,10 UNION ALL
   Select 5,'E5',30000,30
```

## Retrieving Table Data:

The SELECT statement is used to retrieve the data from the table. It is possible to retrieve all the table data or particular column data. In order to retrieve complex data from the tables, we can add extra optional clauses to the select statement.

**Note:** A maximum of 4096 columns can be specified in a select statement.

Syn:

```
SELECT <column list> FROM <tablename>
[WHERE <condition>
[GROUP BY <column name>]
[HAVING <Condition based on the grouped data>]
[ORDER BY <column name> [ASC | DESC] ]
```

Ex:

1. Select \* from emp
2. Select empid,ename,sal,dno from emp
3. Select count(\*) from emp

## WHERE Clause:

It is used to provide a condition in the select statement. Only those records that satisfy the condition will be displayed as output. The condition can be specified based on any column of the table.

Ex:

**Q:** Get the details of employees who belongs to the department 20 and also having more than 20000 salary.

**Sol:**

```
Select * from emp
Where dno=20 AND sal>20000
```

**Output:**

Empid	ename	sal	dno
1	E1	25000	20

### GROUP BY Clause:

It is used to group the table data based on a column. After grouping the table data, we can able to apply an aggregate function on each group independently. The aggregations include: COUNT, MIN, MAX, AVG and SUM.

Ex:

**Q.1:** Get the no.of employees in each department.

**Sol:**

```
SELECT dno,count(*) from emp
GROUP BY dno
```

Output:

dno	NoOfEmployees
10	2
20	2
30	1

**Q.2:** Get the total amount paying to each department

**Sol:**

```
SELECT dno,SUM(sal) from emp
GROUP BY dno
```

Output:

dno	TotalSalary
10	45000
20	45000
30	30000

### HAVING Clause:

Like a 'Where' clause, HAVING clause is also used to provide condition based on the grouped data. The difference between them is :

With WHERE clause, condition can be specified on entire column data at a time. i.e., before grouping the table data. Whereas, with HAVING clause the condition can be specified after grouping the table data on each group independently.

When to use which clause depends on the criteria of the query. If the criteria doesn't require grouping, use where clause other wise use having clause.

### Examples:

**Q.1:** Get the no.of employees in each department with employees having more than 25000

**Sol:**

```
SELECT dno,Count(*) NoOfEmployees from emp
WHERE sal>25000 GROUP BY dno
```

Output:

dno	NoOfEmployees
10	1
30	1

**Q.2:** Get the department which are having more than one employee

**Sol:**

```
SELECT dno,Count(*) NoOfEmployees from emp
GROUP BY dno HAVING count(dno)>1
```

Output:

dno	NoOfEmployees
10	2
20	2

### Modifying Table Data:

To modify the values stored in a table, UPDATE statement is used. We can able to modify either a single column or multiple column data and a single record or multiple records data.

#### Syntax:

```
UPDATE <TableName> SET <col1>=Val1 [, Col2=val2,....]
[WHERE <condition>]
```

**Example:** the employee name Samuel with the emp\_no "E001" was wrongly entered as 'A' in the table. To correct it, the following statement is used.

```
UPDATE emp
SET ename='Samuel'
WHERE empid=1
```



### Deleting data from a Table:

To remove rows from a table, DELETE or TRUNCATE statements can be used.

To delete a particular row from the table we have to use DELETE command.

**Syntax:**

```
DELETE FROM tablename WHERE condition
```

**Example:**

```
DELETE FROM emp WHERE emp_no='E002'
```

To delete all rows from the table, TRUNCATE command can also be used.

**Syntax:**

```
TRUNCATE TABLE tablename
```

**Example:**

```
TRUNCATE TABLE emp
```

The TRUNCATE statement removes the whole table at a time. Whereas the delete table command removes each row from the table one by one. This is much faster than a DELETE statement, especially on large tables.

### Removing The Table:

To remove the table along with the structure, DROP statement is used.

**Syntax:**

```
DROP TABLE tablename
```

**Example:**

```
DROP TABLE dept1
```

## DATATYPES

The choice of the datatype determines the kind of data that can be stored in the column and the maximum length of data that can be stored in the column. SQL Server provides different categories of data types. They are:

Exact numerics	Unicode character strings
Approximate numerics	Binary strings
Date and time	Other data types
Character strings	

In SQL Server, based on their storage characteristics, some data types are designated as belonging to the following groups:

- Large value data types: **varchar(max)**, **nvarchar(max)**, and **varbinary(max)**
- Large object data types: **text**, **ntext**, **image**, **varchar(max)**, **nvarchar(max)**, **varbinary(max)**, and **xml**

### Exact Numeric Datatypes:

They will be used to store integer data.

Data type	Range	Storage
<b>bigint</b>	$-2^{63}$ (-9,223,372,036,854,775,808) to $2^{63}-1$ (9,223,372,036,854,775,807)	8 Bytes
<b>int</b>	$-2^{31}$ (-2,147,483,648) to $2^{31}-1$ (2,147,483,647)	4 Bytes
<b>smallint</b>	$-2^{15}$ (-32,768) to $2^{15}-1$ (32,767)	2 Bytes
<b>tinyint</b>	0 to 255	1 Byte

Numeric data types that have fixed precision and scale.

**decimal**[ (p[ , s] ) ] and **numeric**[ (p[ , s] ) ]

Fixed precision and scale numbers. When maximum precision is used, valid values are from  $-10^{38} + 1$  through  $10^{38} - 1$ . The ISO synonyms for **decimal** are **dec** and **dec(p, s)**. **numeric** is functionally equivalent to **decimal**.

p (precision)

The maximum total number of decimal digits that can be stored, both to the left and to the right of the decimal point. The precision must be a value from 1 through the maximum precision of 38. The default precision is 18.

s (scale)

The maximum number of decimal digits that can be stored to the right of the decimal point. Scale must be a value from 0 through  $p$ . Scale can be specified only if precision is specified. The default scale is 0; therefore,  $0 \leq s \leq p$ . Maximum storage sizes vary, based on the precision.

Precision	Storage bytes
1 - 9	5
10-19	9
20-28	13
29-38	17

### Character Datatype:

There are three valid datatypes for storing strings. They are:

**Char:** this is used for storing fixed-length strings. Columns defined as char will store blanks to fill out a fixed number of characters. The maximum length of a character column is 8000 bytes.

**Varchar:** this is used for storing *variable length strings*. Columns defined as varchar will truncate blanks to save space.

**Text:** this is used for storing of virtually unlimited size (upto 2 gigabytes of text per row).

### Binary Datatypes:

These datatypes store strings consisting of binary values. i.e, hexadecimal numbers instead of characters. They are:

**Binary:** Stores binary data of fixed length with a maximum length of 8,000 bytes

**Varbinary:** Stores variable length binary data with a maximum length of 8,000 bytes

**Image:** SQL Server provides a mechanism for storing binary data more than 8000 bytes using image datatypes. For Example, the photograph of the employees can be stored.

### **DateTime Datatype:**

SQL Server enables to store date and time values. Columns using **datetime** or **smalldatetime** will store both date and time.

### **Unicode Datatype:**

Unicode standard includes all the characters that are defined in the various character sets. Using Unicode data types, a column can store any character that is defined by the Unicode standard. Unicode data is stored using **nchar**, **nvarchar**, **ntext** datatypes.

### **Special DataTypes:**

#### **Cursor:**

This data type is used for variables or stored procedure OUTPUT parameters that contain a reference to a cursor. Any variables created with the **cursor** data type are nullable.

The operations that can reference variables and parameters having a **cursor** data type are:

- The DECLARE *@local\_variable* and SET *@local\_variable* statements.
- The OPEN, FETCH, CLOSE, and DEALLOCATE cursor statements.
- Stored procedure output parameters.
- The CURSOR\_STATUS function.
- The **sp\_cursor\_list**, **sp\_describe\_cursor**, **sp\_describe\_cursor\_tables**, and **sp\_describe\_cursor\_columns** system stored procedures.

**NOTE:** The cursor data type cannot be used for a column in a CREATE TABLE statement

#### **Sql\_variant .:**

**sql\_variant** can be used in columns, parameters, variables, and the return values of user-defined functions. **sql\_variant** enables these database objects to support values of other data types.

A column of type **sql\_variant** may contain rows of different data types. For example, a column defined as **sql\_variant** can store **int**, **binary**, and **char** values. The following table lists the types of values that cannot be stored by using **sql\_variant**:

<b>varchar(max)</b>	<b>varbinary(max)</b>
<b>nvarchar(max)</b>	<b>xml</b>
<b>text</b>	<b>ntext</b>
<b>image</b>	<b>timestamp</b>
<b>sql_variant</b>	User-defined types
<b>hierarchyid</b>	

**sql\_variant** can have a maximum length of 8016 bytes. This includes both the base type information and the base type value. The maximum length of the actual base type value is 8,000 bytes.

A **sql\_variant** data type must first be cast to its base data type value before participating in operations such as addition and subtraction.

**sql\_variant** can be assigned a default value. This data type can also have NULL as its underlying value, but the NULL values will not have an associated base type. Also, **sql\_variant** cannot have another **sql\_variant** as its base type.

A unique, primary, or foreign key may include columns of type **sql\_variant**, but the total length of the data values that make up the key of a specific row should not be more than the maximum length of an index. This is 900 bytes.

A table can have any number of **sql\_variant** columns.

### Table:

Is a special data type that can be used to store a result set for processing at a later time. **table** is primarily used is for temporary storage of a set of rows returned as the result set of a table-valued function.

**table** variables provide the following benefits:

- A **table** variable behaves like a local variable. It has a well-defined scope. This is the function, stored procedure, or batch that it is declared in.

Within its scope, a **table** variable can be used like a regular table. It may be applied anywhere a table or table expression is used in SELECT, INSERT, UPDATE, and DELETE statements. However, **table** cannot be used in the following statement:

```
SELECT select_list INTO table_variable
```

- **table** variables are automatically cleaned up at the end of the function, stored procedure, or batch in which they are defined.

- CHECK constraints, DEFAULT values and computed columns in the **table** type declaration cannot call user-defined functions.
- **table** variables used in stored procedures cause fewer recompilations of the stored procedures than when temporary tables are used.
- Transactions involving **table** variables last only for the duration of an update on the **table** variable. Therefore, **table** variables require less locking and logging resources.

### New Data Types

Data type	Description
xml	Used to store XML data.

### Enhanced Data Types

Data types	Description
varchar(max)	Indicates that the maximum storage size for the varchar data type is $2^{31}-1$ bytes.
nvarchar(max)	Indicates that the maximum storage size for the nvarchar data type is $2^{31}-1$ bytes.
varbinary(max)	Indicates that the maximum storage size for the varbinary data type is $2^{31}-1$ bytes.

## CONSTRAINTS

It is very important for the data in a database to be accurate, consistent and reliable. This implies that it is very crucial to enforce data integrity.

Data integrity ensures the consistency and correctness of data stored in a database. It is broadly classified into the following categories.

- **Entity integrity** – ensures that each row in a table is unique. It enforces integrity of the data contained in the columns, which uniquely identify the rows in a table.
- **Domain integrity** – ensures that only valid ranges of values are allowed to be stored in a column. It can be enforced, by restricting the type of data, the range of values and the format of the data.
- **Referential integrity** – ensures that the data in the database remains uniformly consistent, accurate and usable even after the data in it has been changed. It maintains the integrity of data by ensuring that the changes made in the parent table are also reflected in all the dependent tables.

### Enforcing Data Integrity

Incorporating business rules and specifying the relevant constraints can enforce data integrity.

Business rules refer to specific policies followed by an organization for the smooth running of its business. Business rules ensure that the database stores accurate data in relation to the business policies.

For example an organization may have a personnel policy, which states that the minimum salary of an employee is \$500. This is a business rule.

Constraints refer to rules, which restrict the values that are inserted in the columns of a table.

A Constraint can either be created at the time of creating a table or can be added later. When a constraint is created after table creation, it checks the existing data. If the existing data does not conform to the rule being enforced by the constraint, then the constraint is rejected.

A constraint can be created using either of the following statements:

- CREATE TABLE statement.
- ALTER TABLE statement.

#### **Using CREATE TABLE statement:**

A constraint can be defined on a column at the time of creating a table. It can be created with the CREATE TABLE statement:

**The syntax is :**

```
CREATE TABLE table_name
(Columnname1      datatype      CONSTRAINT      constraint_name
constraint_type, Columnname2 data type .....)
```

#### **Using ALTER TABLE statement:**

A constraint can also be defined on a column after a table has been created. This can be done using the ALTER TABLE statement.

**The syntax is:**

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name constraint_type (fieldName)
```

#### **CHECK CONSTRAINT:**

A CHECK constraint enforces domain integrity by restricting the values to be inserted in a column. It allows only valid values in a column.

It is possible to define multiple CHECK constraints on a single column.

**Syntax:**

```
CREATE TABLE table_name
(colName datatype [CONSTRAINT const_name] CHECK(<criteria>),
colName2 .....)
```



**Example:**

Ex:

```
1. create table dept
   (dno int,dname char(10),
   Dsize int constraint ck_dept_dsize check(dsize>20))

2. insert into dept values(10,'HR',30)

3. insert into dept values(20,'Admin',15)
   --Error, since dsize value is not greater than 20
```

The rules regarding the creation of CHECK constraint are as follows:

- It can be created at the column level as well as table level.
- It is used to limit the values that can be inserted into a column.
- It can contain user-specified search conditions.
- It cannot contain sub queries. A sub query is one or more select statements embedded within another select statement.
- It does not check the existing data into the table if created with the *No Check Option*.
- It can reference other columns of the same table.

A check constraint can also be added to an existing table by using the following syntax.

**Syntax:**

```
ALTER TABLE table_name
ADD CONSTRAINT const_name
CHECK(<criteria>)
```

### Examples:

```
1. insert into dept values(20,'Admin',15)
   --Correct, since we removed the restriction in the
   previous example.

2. ALTER TABLE dept
   ADD CONSTRAINT ck_dept_dsize
   CHECK(dsize>20)
   --Error, since one of the dsize value is not greater
   than 20

3. update dept set dsize=25 where dsize<20
   --Modifying the dsize value for those records which
   are having dsize value less than 20.

4.   Execute command.no.2 again. Now, it will execute
   successfully.
```

### UNIQUE CONSTRAINT:

UNIQUE constraints are used to enforce uniqueness on non-primary key columns.

A **primary key** constraint column automatically includes a restriction for uniqueness. However a unique constraint allows null values.

It is not advisable to have columns with null values, though these are allowed. A null value is also validated, hence there can only be one record in the table with a null value. The table can't contain another row with null value.

A unique column does not allow duplicate values but allows one null value.

### Syntax:

```
CREATE TABLE table_name
(colName datatype CONSTRAINT <const_name> UNIQUE,
colName2 .....)
```

### Examples:

```
1. create table emp
   (empid int constraint uk_emp_empid unique,
    ename char(10))

2. insert into emp values(100,'A')

3. insert into emp values(100,'B')
   --Error, duplicate value into empid column

4. insert into emp(ename) values('B')

5. insert into emp(ename) values('C')
   --Error, duplicate NULL value into empid column
```

### Adding a UNIQUE constraint to an existing table:

```
Syn: ALTER TABLE <tablename>
      ADD CONSTRAINT <constraintName>
      UNIQUE(<column Name>)

Ex:  Alter table emp
      Add constraint uk_emp_empid unique(empid)
```

### The rules regarding the creation of UNIQUE constraint are:

- It does not allow two rows to have the same non-null values in a table.
- It gets enforced automatically when a UNIQUE index is created.
- Multiple UNIQUE constraints can be placed on a table.

**PRIMARY KEY constraint:**

A primary key constraint is defined on a column or a set of columns whose values uniquely identify the rows in a table. These columns are referred to as primary key columns. A primary key column cannot contain NULL values since it is used to uniquely identify rows in a table.

While defining a PK constraint, you need to specify a name for the constraint. If no name is specified, SQL Server automatically assigns a unique name to the constraint.

Any column or set of columns that uniquely identifies a row in a table can be a candidate for the primary key. These set of columns are referred to as candidate keys. One of the candidate keys is chosen to be the primary key, based on familiarity and greater usage. The other key, which is not chosen as the primary key is called as alternate key.

A table should contain only one primary key.

Creating a primary key is very much similar to the unique constraint. In the syntax of unique, simply replace the keyword unique with a primary key.

**Syntax:**

```
CREATE TABLE table name
(ColumnName1 datatype CONSTRAINT constraint_name Primary key,
ColumnName2 data type .....)
```

OR

```
ALTER TABLE table name
ADD CONSTRAINT constraint_name PRIMARY KEY (field name)
```

**Example:**

```
1. create table dept
   (dno int constraint pk_dept_dno primary key,
    dname char(10), dsize int)

2. insert into dept values(10, 'HR', 30)

3. insert into dept values(10, 'Admin', 25)
   --Error, duplicate dno value

4. insert into dept (dname, dsize) values('Tester', 30)
   --Error, cannot enter a null value into dno column.
```

**Adding a Primary Key:**

It is not possible to directly add a primary key to an existing table's column if it is having any data. At first, we must add a 'NOT NULL' option to that column. Then only, we can add the primary key.

```
1. create table empl
   (eno int, ename char(10))

2. insert into empl values(100, 'A')

3. insert into empl values(101, 'B')

4. alter table empl
   add constraint pk_empl_eno primary key(eno)
   --Error, since there is no NOT NULL option for the
   eno column

5. alter table empl
   alter column eno int not null
   --Specifying the NOT NULL option to the eno column

6. Execute command no.4 again. Now, it will execute
   successfully.
```

### Composite Primary Key:

In some tables it is not possible to provide the primary key on any single column of the table. Then check the table data to identify any column combination, by using which, we can identify different records and other data of the table. If it is found, make that column as the primary key called as 'Composite Primary Key'. A maximum of 16 columns can be combined using this constraint.

Ex:

```
CREATE TABLE OrderDetails
(ordid int, ino char(2),orddate datetime, cno int, ,
Quantity int, constraint pk_od_ordid_ino
Primary Key(ordid,ino))
```

### FOREIGN KEY CONSTRAINT:

A foreign key is a column or combination of columns whose values match the primary key of another table.

A foreign key does not have to be unique. However, foreign key values must be copies of the primary key values of the master table.

The Foreign Key can be given on a column which is having the same data type, size and data as of the primary key column. The table with foreign key is called as 'Referencing/child' table and the table with primary key is called as 'Referenced/parent' table.

Once we apply a relation between the tables, lot of restrictions will be applied to the table operations. Some of them are:

1. The data to be entered in a foreign key column must have a matching value in the primary key column of the parent table.
2. It is not possible to delete the parent table record if it is having a reference in the child table.

### Syntax:

```
CREATE TABLE <TableName>
(<ColName1> <Datatype> [(Size)]
[CONSTRAINT <ConstraintName>]
REFERENCES <ParentTable>(<PrimarykeyCol>)
[ON DELETE CASCADE], <ColName2>.....)
```

**Example:**

```
1. create table emp
   (empid int,ename char(10),sal int,
   dno int constraint fk_emp_dno references dept(dno))

2. insert into emp values(100,'A',50000,20)

3. insert into emp values(101,'B',45000,50)
   --Error, dno 50 is not having any references in the
   parent table.

4. insert into emp
   select 101,'B',35000,10 union all
   select 102,'C',40000,20 union all
   select 103,'D',50000,30
   --Enterring multiple values at a time into the table.

5. delete from dept where dno=20
   --Error, since this record is having some references in
   the child table.
```

The rules regarding the creation of a FOREIGN KEY constraint are as follows:

- The no.of columns specified in the foreign key statement must match the no.of columns in the references clause and must be specified in the same order.
- The foreign key can reference the primary key or a UNIQUE constraint of another table.
- The foreign key constraint that is created using the WITH NOCHECK option will prevent the foreign key constraint from validating existing data.

**ON DELETE CASCADE:**

In general it is not possible to delete a record from the parent table if it is having any references in the child table. In this case, at first, we have to remove the child table records and then the parent table record.

With the use of 'ON DELETE CASCADE', whenever, we remove the parent table record, that record and all its child table records will be removed automatically.

### **Adding a Foreign key:**

#### **Syn:**

```
ALTER TABLE <TableName>
ADD CONSTRAINT <ConstraintName>
FOREIGN KEY(<ColName>)
REFERENCES <ParentTable>(<PKColumn>)
[ON DELETE CASCADE]
```

#### **Ex:**

```
alter table emp
  drop constraint fk_emp_dno
  --removing existing constraint

2. alter table emp
  add constraint fk_emp_dno
  foreign key(dno) references dept(dno)
  on delete cascade
  --adding the foreign key with new option

3. delete from dept where dno=20
  --Deletes the records with dno 20 both from dept and emp
  tables.
```

### **CONSTRAINTS vs TRIGGERS:**

Each category of data integrity is best enforced through the following constraints.

Entity integrity is best enforced through the PRIMARY KEY constraint, UNIQUE constraint and the IDENTITY property.

The IDENTITY property creates an identity column in the table. An identity column is a primary key column for which numeric values are generated automatically. Each generated value is different from the previous value, thus ensuring uniqueness of data in each row at the table. The IDENTITY property also ensures that while inserting data, the user need not explicitly insert a value for the identity column.

Domain integrity is best enforced through the DEFAULT constraint, CHECK constraint and the FOREIGN KEY constraint.



## RULES AND DEFAULTS

Rules and defaults are objects, which help enforce the data integrity. These objects are bound to columns or user defined data types to ensure that only valid values are allowed to insert into the tables.

### RULES:

A rule provides a mechanism for enforcing domain constraints for columns or user defined data types. The rule is applied before any modification is to be done on the table. In other words, a rule specifies the restriction on the values for a column or a user defined data type.

The syntax of the CREATE RULE statement is:

#### Syn:

```
CREATE RULE rule_name  
AS conditional_expression.
```

#### example:

```
CREATE RULE type_rule  
AS @typerule IN('business','mod_cook', 'trad_cook',  
'popular_comp', 'psychology')
```

### More Examples:

```
CREATE RULE dept_name_rule  
AS @deptname NOT IN ('accounts','stores')
```

```
CREATE RULE max_price_rule  
AS @maxprice >=$5000
```

```
CREATE RULE emp_code_rule  
AS @empcode LIKE '[F-M][A-Z]
```

Information on a rule can be obtained using the **sp\_help** system stored procedure. The text of a rule can be displayed using the **sp\_helptext** system stored procedure with the name of the rule as its parameter.

### **Binding Rules:-**

A rule can be bound using the **sp\_bindrule** system stored procedure.

#### **Syntax:**

```
Sp_bindrule rule_name, object_name.ColName
```

For example:

```
Sp_bindrule type_rule, 'titles.type'
```

Binds the rule, type\_rule, to the type column of the titles table

The restrictions on the use of the rules are as follows:

- Only one rule can be bound to a column or a user defined datatype.
- A rule cannot be bound to system datatypes.
- If a new rule is bound to a column or datatype that is already been inserted in the table, the existing values in the tables do not have to meet the criteria specified by the rule.
- A rule cannot be defined for a system-defined datatype

### **Unbinding Rules:**

A rule can be unbound from a column or user-defined datatype using the **sp\_unbindrule** system stored procedure.

The syntax of the **sp\_unbindrule** is:

```
Sp_unbindrule object_name
```

For example:

```
Sp_unbindingrule 'titles.type'
```

## DEFAULTS:

Default is a constant value assigned to a column, into which the user need not insert values. A default can be bound to a column or a user-defined datatype.

The syntax of the CREATE DEFAULT statement is:

### Syn:

```
CREATE DEFAULT default_name  
As constant_expression
```

Any constant, built-in function, mathematical expression or a global variable can be used in the constant expression. The character and date constants must be included in single quotation marks ('), whereas money, integer, and floating-point constants can be specified without quotation marks.

## Binding Defaults

The **sp\_bindefault** system stored procedure is used for binding a default.

The syntax of sp\_bindefault is:

```
Sp_bindefault default _name_, 'object_name.ColName'
```

For example:

1. **CREATE DEFAULT** city\_default AS 'Oakland'
2. **Sp\_default** city\_default, 'authors.city'

Creates a default, city\_default, and binds the default value, Oakland, to city column of the authors table.

The statement

```
Sp_bindefault city_default, city_datatype.
```

Binds the default city\_default to the user-defined datatype, city\_datatype.

### Unbinding Defaults:

Defaults can be unbound from a column or user defined datatype using the **sp\_unbinddefault** system stored procedure.

The syntax of sp\_unbinddefault is:

```
Sp_unbinddefault object_name  
For example:  
Sp_unbinddefault 'authors.city'
```

Unbinds the default specified on the city column of the authors table.

### Renaming Rules And Defaults:

The **sp\_rename** system stored procedure can be used for renaming rules and defaults.

The syntax of sp\_rename is:

```
Sp_rename old_object_name, new_object_name
```

### Dropping Rules And Defaults:

The DROP RULE and DROP DEFAULT statement can be used to drop a rule and default respectively. A rule or default must be unbound from the column or the user-defined data type before it is dropped.

The syntax for dropping a rule is:

```
DROP RULE rule_name  
(Or)  
DROP DEFAULT default_name
```

## JOINS

By using joins, you can retrieve data from two or more tables based on logical relationships between the tables. Joins indicate how Microsoft SQL Server should use data from one table to select the rows in another table.

A join condition defines the way two tables are related in a query by:

- Specifying the column from each table to be used for the join. A typical join condition specifies a foreign key from one table and its associated key in the other table.
- Specifying a logical operator (=, <>, and so on) to be used in comparing values from the columns.

Joins can be specified in either the FROM or WHERE clauses. The join conditions combine with the WHERE and HAVING search conditions to control the rows that are selected from the base tables referenced in the FROM clause.

Specifying the join conditions in the FROM clause helps separate them from any other search conditions that may be specified in a WHERE clause, and is the recommended method for specifying joins. A simplified SQL-92 FROM clause join

### Syntax:

```
SELECT columnList
FROM first_table join_type second_table [ON (join_condition)]
```

*Join\_type* specifies what kind of join is performed: an inner, outer, or cross join. *join\_condition* defines the predicate to be evaluated for each pair of joined rows. This is an example of a FROM clause join specification:

Although join conditions usually have equality comparisons (=), other comparison or relational operators can be specified, as can other predicates

When SQL Server processes joins, the query engine chooses the most efficient method (out of several possibilities) of processing the join. Although the physical execution of various joins uses many different optimizations, the logical sequence is:

- The join conditions in the FROM clause are applied.
- The join conditions and search conditions from the WHERE clause are applied.
- The search conditions from the HAVING clause are applied.

### Types of Joins:

Join conditions can be specified in either the FROM or WHERE clauses; specifying them in the FROM clause is recommended. WHERE and HAVING clauses can also contain search conditions to further filter the rows selected by the join conditions.

Based on the way the select statements are provided and the records they return, joins are categorized into the following ways.

1. Inner Join
2. Outer Join
  - Left Outer Join
  - Right Outer Join
  - Full Outer Join
3. Cross Join
4. Self Join

### Inner Join:

This join returns only the matching data between the tables. It is mainly used to retrieve the related data between the tables.

**Ex:** Get the details of the students along with the class names from the following student and class tables.

#### Class:

CID	CNAME	CSIZE
100	MCA	30
101	MTECH	25
102	BTECH	40

#### Student:

SID	SNAME	CID	MARKS
1	A	101	300
2	B	100	200
3	C	101	150
4	D	102	150
5	E	NULL	300

**Sol:**

```
SELECT sid,sname,s.cid stdCID,c.cid clsCID,cname
FROM Student s INNER JOIN Class c
ON s.cid=c.cid
```

Output:

sid	sname	stdCID	clsCID	cname
1	A	101	101	MTECH
2	B	100	100	MCA
3	C	101	101	MTECH
4	D	102	102	BTECH

### Outer Joins:

This join returns the unmatched records between the tables. Which tables unmatched data to be displayed, depends on the type of outer join we are using. Sometimes the unmatched data will be displayed along with matched data and some other times the unmatched data will be displayed alone.

#### Left Outer Join:

This join returns the unmatched data from the left table. The corresponding values from the right table will be NULL.

**Ex:**

a. *Get the details of all students along with classes if any student is having a class.*

**Sol:**

```
SELECT sid,sname,s.cid stdCID,c.cid clsCID,cname
FROM Student s LEFT OUTER JOIN Class c
ON s.cid=c.cid
```

**Output:**

sid	sname	stdCID	clsCID	cname
1	A	101	101	MTECH
2	B	100	100	MCA
3	C	101	101	MTECH
4	D	102	102	BTECH
5	E	NULL	NULL	NULL

b. Get the details of students who are not assigned to any class.

**Sol:**

```
SELECT sid,sname
FROM Student s LEFT OUTER JOIN Class c
ON s.cid=c.cid WHERE c.cid IS NULL
```

**Output:**

sid	sname
5	E

**Right Outer Join:**

This join returns the unmatched data from the right table. The corresponding values from the left table will be null.

**Ex:**

a. Get the details of all classes along with students.

**Sol:**

```
SELECT sid,sname,s.cid stdCID,c.cid clsCID,cname
FROM Student s RIGHT OUTER JOIN Class c
ON s.cid=c.cid
```



**Output:**

sid	sname	stdCID	clsCID	cname
2	B	100	100	MCA
1	A	101	101	MTECH
3	C	101	101	MTECH
4	D	102	102	BTECH
NULL	NULL	NULL	103	BCA

*b. Get the details of classes which are not having students.*

**Sol:**

```
SELECT c.cid,cname
FROM Student s RIGHT OUTER JOIN Class c
ON s.cid=c.cid WHERE s.cid IS NULL
```

**Output:**

cid	cname
103	BCA

**Full Outer Join:**

It is used to get the unmatched data both from the left table as well as from the right table.

**Cross Join:**

A join without common column criteria is called as a cross join. This join is used to find the different combinations of data between the tables. The output will be the product of number of records between the tables.

**Ex:** Get the different possibilities how a student can be assigned to a class only when they are having more than 200 marks

**Sol:**

```
Select sid,sname,c.cid,cname
From student s cross join class c
where marks>200
```

**NOTE:**

In case of **ORACLE** database, Inner join is called as EQUI/NATURAL Join and Outer Join is called as Non-EQUI Join.

**Self Join:**

Joining a table to it is called as a self-join. In a self-join, since we are using the same table twice, to distinguish the two copies of the table, we will create table aliases.

In the syntax of self join we never use the keyword 'self'. Internally, we perform an inner join only.

**Ex:** Get the employee details along with manager names from the EMP table. In this table one the employee will be the manager for other employee.

empid	ename	mgrid	sal	comm	hiredate
100	Anil	105	15000	10	1993-01-12
101	Balu	103	35000	NULL	1992-04-06
102	Avinash	104	39000	20	1993-02-05
103	Sumit	NULL	38000	10	1990-10-11
104	Chandu	100	50000	30	1993-01-01
105	Santosh	103	30000	NULL	1994-04-05
106	vivek	101	35000	15	2000-06-07

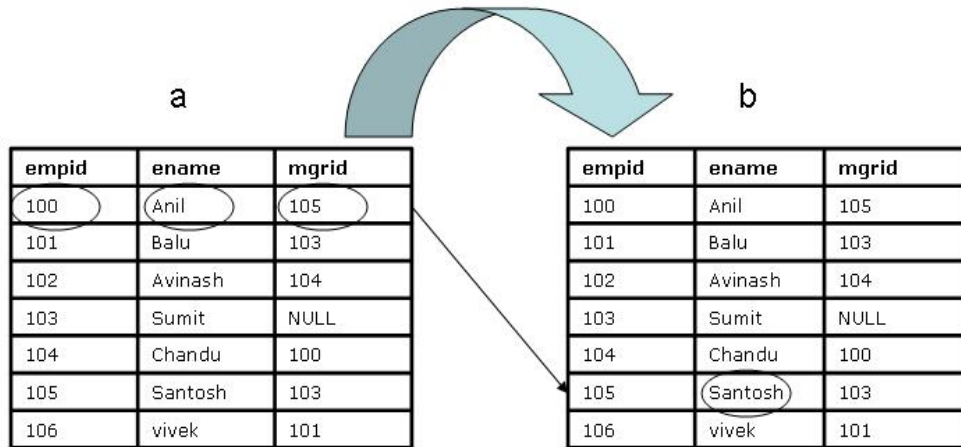
**Sol:**

```
Select a.empid,a.ename,a.mgrid,b.ename mgrName
From emp a inner join emp b
On a.mgrid=b.empid
```

**Output:**

empid	ename	mgrid	mgrName
100	Anil	105	Santosh
101	Balu	103	Sumit
102	Avinash	104	Chandu
104	Chandu	100	Anil
105	Santosh	103	Sumit
106	vivek	101	Balu

### Internal Execution Process:



### Joining more than two tables:

It is also possible to join more than two tables. In this case each table must have a relation with other table.

Syn:

```
Select <column list>
FROM <table1> INNER JOIN <table2> ON <criteria>
INNER JOIN <table3> ON <criteria>
INNER JOIN <table4> ON <criteria>
.....
..... . .
```

**Ex: Get the full details of the orders made by the customers.**

```
select cu.customerid,companyname,o.orderid,orderdate,
od.productid,productname,od.unitprice,quantity
from customers cu inner join orders o
on cu.customerid=o.customerid
inner join [order details] od
on o.orderid=od.orderid
inner join products p
on od.productid=p.productid
```

## SUBQUERIES

A SubQuery is a SELECT statement containing another SELECT statement. In these SubQueries, at first, the inner query will be executed and based on the result next higher query will be executed. Whenever, it happened to compare a column data with the output of another select statement then we use sub queries.

### Types Of SubQueries:

Based on the way the Select statements are provided and the way they will execute, SQL Server provides the following types of subqueries.

- Simple SubQuery
- Nested SubQuery
- Multiple SubQuery
- Correlated SubQuery

### Simple SubQuery:

It is a select statement containing another select statement. At first inner query will be executed and based on the result next higher query will be executed.

**Ex:** *Get the Second maximum salary from the following EMP table.*

Sol:

```
SELECT max(Sal) FROM EMP  
WHERE sal < (SELECT max(Sal) FROM EMP)
```

**Output:** 39000

### Neseted SubQueries:

A Nested SubQuery is a select statement, containing another select statement, which contains another select statement and so on. Such type of nesting nature of select statements is called as a nested subquery. In these subqueries, at first, the innermost query will be executed and based on the result next higher query will be executed and so on. A maximum of 32 select statements can be combined in these nested subqueries.

**Ex:** *Get the Third maximum salary from the emp table*

**Sol:**

```
SELECT max(sal) FROM EMP
WHERE sal<(SELECT max(sal) FROM EMP
WHERE sal<(SELECT max(sal) FROM EMP))
```

**Output:** 38000

### Multiple SubQueries:

It is select statement containing different select statements at different places.

**Ex:** *List out the employees who are having salary less than the maximum salary and also having hiredate greater than the hiredate of an employee who is having the maximum salary.*

**Sol:**  

```
SELECT empid,ename,sal,hiredate FROM EMP
WHERE sal<(SELECT max(sal) FROM EMP) AND
hiredate>(SELECT hiredate FROM EMP
WHERE sal=(SELECT max(Sal) FROM EMP))
```

**Output:**

Empid	Ename	Sal	Hiredate
100	Anil	15000	1993-01-12
102	Avinash	39000	1993-02-05
105	Santosh	30000	1994-04-05
106	vivek	35000	2000-06-07

### Correlated SubQueries:

In a normal subquery, at first, the inner query will be executed and only once. Based on the result next higher query will be executed.

Whereas in a correlated subquery, the inner query will be executed for each record of the parent statement table. The internal execution process of this subquery will be as follows:

1. A record value from the parent table will be passed to the inner query.
2. The inner query execution will be done based on that value.
3. The result of the inner query will be sent back to the parent statement.
4. The parent statement finishes the processing for that record.

The above 4 steps will be executed for each record of the parent table.

### Examples:

1. *Get the nth maximum salary from the emp table.*

**Sol:**

```
SELECT a.empid,a.ename,a.sal FROM EMP a
WHERE 2=(SELECT count(distinct(b.sal)) FROM EMP b
WHERE a.sal<b.sal)
```

**Note:** The above query is to calculate 3<sup>rd</sup> maximum salary. To calculate nth maximum salary specify the number "n-1" in place of "2".

2. *Get the Top 3 maximum salaries from emp table.*

**Sol:**

```
SELECT a.empid,a.ename,a.sal FROM EMP a
WHERE 3>(SELECT count(distinct(b.sal)) FROM EMP b
WHERE a.sal<b.sal)Order by a.sal desc
```

### Output:

Empid	Ename	Sal
104	Chandu	50000
102	Avinash	39000
103	Sumit	38000

3. *Get the details of employees who are not the managers*

**Sol:**

```
SELECT a.empid,a.ename FROM EMP a
Where 0=(SELECT count(*) FROM EMP b WHERE a.empid=b.mgrid)
```

**Output:**

Empid	Ename
102	Avinash
106	vivek

4. *Get the details of employees who are the managers for more than one employee*

**Sol:**

```
SELECT a.empid,a.ename FROM EMP a
Where 1<(SELECT count(*) FROM EMP b WHERE a.empid=b.mgrid)
```

**Output:**

Empid	Ename
103	Sumit

**More Examples:**

**SubQueries With IN:**

A subquery introduced with IN returns zero or more values. Consider the following example where all author ID'S, from the TITLEAUTHOR table, are displayed whose books are sold:

```
SELECT au_id
FROM titleauthor
WHERE title_id IN (SELECT title_id FROM sales)
```

SQL Server returns a list of all title IDs to the main query then lists all the authors, whose books are sold, in the result set.

Consider the following example where the server returns a list of publisher IDs to the main query, and then determines whether each publisher's pub\_id is in that list:

```
SELECT publisher=pub_name
FROM publishers
WHERE pub_id IN ( SELECT pub_id FROM titles WHERE
type='business')
```

The inner query is evaluated first and then the result set is sent to the outer query.

Consider another subquery with the IN clause:

```
SELECT type=type, Average=AVG(ytd_sales)
FROM titles
WHERE type IN (SELECT type FROM titles
WHERE title=" the busy executive's database guide" or
title='Is Anger the Enemy?') GROUP BY type
```

The inner query is evaluated first and then the result set is sent to the outer query.

The NOT IN clause is used in the same way as the IN clause. Consider the following example.

```
SELECT pub_id, pub_name
FROM publishers
WHERE pub_id NOT IN (SELECT pub_id FROM titles
WHERE type='mod_cook')
```



### **Sub Queries with EXISTS:**

The subquery, when used with the EXISTS clause, always returns data in terms of TRUE OR FALSE and passes the status to the outer query to produce the results set. The subquery returns a TRUE value if the result set contains any rows.

The query introduced with the EXISTS keyword differs from other queries. The EXISTS keyword is not preceded by any column name, constant or there expression and it contains an asterisk (\*) in the SELECT list.

```
1. SELECT pub_name
FROM publishers
WHERE EXISTS (SELECT * FROM titles WHERE type='business')

2. SELECT pub_name
FROM publishers
WHERE EXISTS (SELECT * FROM publishers WHERE
City='Paris')
```

```
SELECT Title=title
FROM titles
WHERE advance>(SELECT AVG (advance) from titles
                WHERE type='business')
```

### **Subquery Restrictions:**

The restrictions imposed are:

- The column list of the select statement of a subquery introduced with the comparison operator can include only one column.
- The column used in the WHERE clause of the outer query should be compatible with the column used in the select list of the inner query.
- The DISTINCT keyword cannot be used with the subqueries that include the GROUP BY clause.
- The ORDER BY clause, GROUP BY clause and INTO keyword cannot be used in a subquery, because a subquery cannot manipulate its result internally.
- Do not specify more than one column name in the subquery introduced with the EXISTS keyword.
- A view created with a subquery cannot be updated.

## BUILT IN FUNCTIONS

SQL Server provides a lot of functions, which can be used as calculated fields as part of column lists in a SELECT statement. Such functions are called as Built-in Functions.

### Types of Built-in Functions:

Based on the type of data we are using inside the functions, built-in functions are categorized into the following ways.

- Aggregate Functions
- Numeric Functions
- Date and Time Functions
- String Functions

### Aggregate Functions:

Aggregate functions are used to produce summary data using tables.

Function	Parameters	Description
AVG	(ALL/DISTINCT] expression	Returns the average of values specified in the expression, either all records or distinct records
SUM	(ALL/DISTINCT] expression)	Returns the sum of values specified in the expression, either all records or distinct records.
MIN	(expression)	Returns the minimum of a value specified in the expression.
MAX	(expression)	Returns the maximum of a value specified in the expression.
COUNT	(ALL  DISTINCT expression)	Returns the number of unique or all records specified in

		expression.
--	--	-------------

### Examples of Aggregate functions

SELECT 'Average Price'=AVG(price) FROM titles	Returns the average value of all the price values in the titles table with user-defined heading.
SELECT 'Sum'=SUM(DISTINCT advance) FROM titles	Returns the sum value of all-the unique advance values in the titles table with user-defined heading.
SELECT 'Minimum Ytd Sales'=MIN(ytd_sales) FROM titles	Returns the minimum value of ytd_sales value in the titles table with user-defined heading.
SELECT 'Maximum Ytd Sales'=MAX(ytd_sales) FROM titles	Returns the maximum value of ytd_sales in the titles table with user-defined heading.
SELECT 'Unique Price'= COUNT(DISTINCT price) FROM titles	Returns the number of unique price values in the titles table with user-defined heading.
SELECT 'Price=COUNT(price) FROM titles	Returns the number of total number of price values in the titles with user-defined heading.

### NUMERIC FUNCTIONS:

- a. SQRT:** Gets the square root of a number

```
EX: Select SQRT(25) --o/p: 5
```

- b. SQUARE:** Gets the square of a number

```
Ex: select SQUARE(4) --o/p: 16
```

- c. POWER:** Gets the power a number

```
Ex: select POWER(2,3) --o/p: 8
```

- d. **ROUND:** Used to round a floating point number to the required no.of decimal places

```
Ex:  1. Select ROUND(12.347,2)  --o/p: 12.35
      2. select ROUND(12.343,2)  --o/p: 12.34
```

- e. **CEILING:** Returns the nearest higher integer value for the given number

```
Ex: select CEILING(12.345)  --o/p:13
```

- f. **FLOOR:** Returns the nearest smaller integer value for the given number

```
Ex: select FLOOR(12.345)    --o/p:12
```

## Date and Time Functions:

- a. **GETDATE:** Returns system date and time

```
Ex: select GETDATE()        --o/p: 2010-06-10 22:44:18.187
```

- b. **DATEPART:** Returns the part of given date or time.

Syn: DATEPART(Format,Date)

Format	Output	Format	Output
dd	Days	mi	Minutes
mm	Months	ss	Seconds
yy	Years	ms	Milliseconds
hh	Hours	dw	Day of Week
mi	Minutes	dy	Day of Year
ss	Seconds		

```
ex:  1. select datepart(dd,getdate()) --10
      2. select datepart(hh,getdate()) --22
```

- c. DATENAME:** It Returns the name of the corresponding datepart. If the specified datepart is not having a name, this function will return that value directly.

Syn: DATENAME(datepart,date)

```
Ex:
1. select datename(dw,getdate()) --Friday
2. select datename(mm,getdate()) --June
3. select datename(dd,getdate()) --10
4. select datename(dw,'10/21/77') --Friday
5. calculate how many employees joined in each
   weekday from the emp table
      select datename(dw,hiredate),count(*) from emp
      group by datename(dw,hiredate)
```

- d. DATEADD:** It is used to add a number to a part of a date. If we specify a negative value, the number will be subtracted from the given datepart.

Syn: DATEADD(datepart,n,date)

```
Ex:
1. select dateadd(dd,3,getdate()) --2010-06-13

2. Getting the weekday name which was 10 days before
   today's date
sol:-select datename(dw,dateadd(dd,-10,getdate()))
--o/p: Saturday
```

- e. DATEDIFF:** Returns the difference between two given dates. The output will be the subtraction of date1 from date2. The difference can be in days or months or years etc., based on the interval specified.

**Syn: DATEDIFF(interval, data1, date2)**

**Ex:** Get the experience of each employee from the employee table.

```
ex:  Get the experience of each employee from the
      employee table.
SQL: SELECT      empid,ename,datediff(yy,hiredate,getdate())
      FROM EMP
```

- f. CONVERT:** SQL server handles certain datatype conversion automatically. If a character expression is compared with an int expression, SQL server makes the conversion automatically for the comparison(implicit conversion).

The CONVERT function is used to change data from one type to another when SQL server cannot implicitly perform a conversion. Using the CONVERT function, data can be modified in variety of styles.

The syntax is:

```
CONVERT(datatype[(length), expression[,style])
```

**Ex:**

```
1. SELECT Ytd_Sales=CONVERT(char(10),ytd_sales)
   FROM titles.

2. SELECT CONVERT(char,getdate(),101)  --06/10/2010
```

**STRING FUNCTIONS:**

<b>FUNCTION</b>	<b>DESCRIPTION</b>
CHARINDEX('pattern', expression)	Returns the starting position of the specified pattern.
LOWER (character_expression)	Converts two string and evaluates the similarity between them on a scale of 1 to 4.
LTRIM(character_expression)	Returns the data without leading brackets
REPLICATE(char_expression, integer_expression)	Repeats a character expression a specified number of times.
REVERSE(character_expression)	Returns the reverse of character expression.
RIGHT(character_expression, integer_expression)	Returns the part of the character string from the right.
RTRIM(character_expression)	Returns the data without trailing blanks.
SPACE(numeric_expression)	Returns a string of repeated spaces. The number of spaces is equal to the integer expression.
UPPER(character_expression)	Converts the character expression into upper case.

## OPERATORS

An operator is a symbol specifying an action that is performed on one or more expressions. SQL Server provides few methods of searching rows in a table. These methods can be broadly categorized into the following categories.

- Arithmetic Operators like +, -, /, \*, %
- Comparison operators like =, >, <, >=, <=, !=, !< and !>
- Range operators like **BETWEEN** and **NOT BETWEEN**.
- List operators like **IN** and **NOT IN**.
- String operators like **LIKE** and **NOT LIKE**.
- Unknown values like **IS NULL** and **NOT NULL**.
- Logical operators like **AND**, **OR** and **NOT**.

### Arithmetic operators:

The arithmetic operators supported by SQL server are:

- + for addition
- - for subtraction
- / for division
- \* for multiplication
- % for modulo

The *modulo* arithmetic operator is used to obtain the remainder of two divisible numeric integer values. It cannot be used with money data type columns.

All the arithmetic operators can be used in the SELECT list with the column names and numeric constants in a combination.

Example:

```
SELECT title_id, price, price+5, ytd_sales*5 FROM titles.
```



When any arithmetic operation is performed on a NULL value, the result is always NULL because NULL values have no explicitly assigned values. Arithmetic operations can be performed on more than one column at a time. Consider the following query code:

```
SELECT Title_Id=title_id, sValue=ytd_sales*price FROM titles
```

The above query computes the product of ytd\_sales and price from the titles table and displays the output with the user-defined headings.

Some rules regarding the usage of arithmetic operators are:

- Arithmetic operations can be performed on numeric columns or numeric constants.
- The modulo (%) operator cannot be used with columns of money, smallmoney, float or real datatypes.

### Comparison Operators:

The command syntax is:

```
SELECT column_list FROM table_name WHERE expression1  
comparison_operator expression2
```

### Valid Comparison operators

Operator	Description
=	Equal to
>	Greater than
<	Less than
>=	Greater than or Equal to
<=	Less than or Equal to
<>, !=	Not Equal to
!>	Not Greater than
!<	Not Less than
()	Controls Precedence

### Examples:

```
SELECT title FROM titles WHERE type='business'
SELECT stor_id, ord_num, title_id FROM sales WHERE qty>20
SELECT type, title_id, price FROM titles WHERE price *
vtd sales<advance.
```

### Range Operator:

The range operator is used to retrieve data that can be extracted in ranges. The range operations are:

- BETWEEN
- NOT BETWEEN

The syntax is:

```
SELECT column_list FROM table_name WHERE expression1
range_operator expression2 AND expression2
```

#### Examples:

1. SELECT title from titles  
WHERE advance BETWEEN 2000 AND 5000
2. SELECT title FROM titles  
WHERE advance NOT BETWEEN 4000 AND 5000

### List Operator

The syntax is:

```
SELECT column_list FROM table_name WHERE expression
list_operator('value_list')
```

#### Examples:

1. SELECT pub\_name, city, state FROM publishers  
WHERE state IN ('MA', 'DC')
2. SELECT pub\_name, city, state FROM publishers  
WHERE state NOT IN ('MA', 'DC')

## LIKE Operator:

SQL Server provides a pattern-matching method for string expressions using the LIKE keyword with the wildcard characters. The LIKE keyword is used to select those rows that match the specified portion of character string. The LIKE keyword allows wildcard characters that can be used as a part of an expression.

Wild card	Description
%	Represents any string of zero or more characters(s)
-	Represents a single character
[]	Represents any single character within the specified range.
[^]	Represents any single character not within the specified range.

Examples of the LIKE operator with wildcards.

Example	Description
SELECT title FROM titles WHERE type LIKE 'bus%'	Returns all titles from titles table where first three characters of the column type are 'bus'
SELECT * FROM publishers WHERE country LIKE 'US_'	Returns all rows from publishers table where country name is three characters long and starts with US where the third character can be anything.
SELECT title_id, price FROM titles WHERE title_id LIKE 'P[SC]]%'	Returns all columns from the titles table where title_id starts with the character P and contains S or C in the second position followed by any number of characters.
SELECT title_id, price FROM titles WHERE title_id, LIKE 'P[^C]]%'	Returns all title_id and price from the titles table where title_id starts with P and does not contain and S as the second character and the third position onwards can contain any characters.

### Unknown Values:

Unknown values refer to the data entered in the form of the NULL keyword. In SQL server terms, NULL is an unknown value or the value for which data is not available. The rows containing the NULL values can be retrieved by using the IS NULL keyword in the WHERE clause.

The Syntax is:

```
SELECT column_list FROM table_name  
WHERE column_name unknown_value_operator.
```

```
SELECT title, ytd_sales FROM titles WHERE ytd_sales IS NULL
```

### Logical Operator

- **OR** – Returns the result when any of the specified search conditions is true.
- **AND** – returns the result when all of the specified search conditions are true.
- **NOT** – **Negates the expression that follows it.**

```
SELECT column_list FROM table_name  
WHERE conditional_expression{AND\OR}[NOT] conditional_expression.
```

### Examples of Logical operators

Example	Description
SELECT * FROM publishers WHERE city='Boston' OR city='Paris'	Returns all the rows specific to the conditions, even if any one of the conditions is true.
SELECT publishers WHERE city='Boston' AND city='MA'	Returns all the rows specific to the conditions, when both the conditions are true.
SELECT * FROM publishers WHERE city='Boston' OR NOT city='Paris'	Returns all the rows specific to the conditions, except the rows specified with the condition after NOT operator.

## VIEWS

A view is a virtual table, which gives access to a subset of columns from one or more tables. Hence, a view is an object that derives its data from one or more tables. These tables are referred to as the base tables or the underlying tables.

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

- ❖ Specific rows of the table
- ❖ Specific columns of the table.
- ❖ Rows fetched by using joins.
- ❖ Statistical summary of data in a given table.
- ❖ Subsets of another view or a subset of views and tables

Common examples of views include:

- ✓ A subset of rows or columns of a base table.
- ✓ A union of two or more tables
- ✓ A join of two or more tables
- ✓ A statistical summary of a base tables
- ✓ A subset of another view, or some combination of views and base tables.

### Creating Views:

Using the CREATE VIEW statement we can create a view.

The syntax of the CREATE VIEW statement is:

```
CREATE VIEW view_name  
[(column_name [,column_name]...)]  
[WITH ENCRYPTION][WITH SCHEMABINDING]  
AS select_statement[WITH CHECK OPTION]
```

The restrictions imposed on views are as follows:

- ✓ A view can be created only in the current database.
- ✓ A view can be created only if there is the SELECT permission on its base table.
- ✓ The SELECT INTO statement cannot be used in a view declaration statement.
- ✓ A view cannot derive its data from temporary tables.

### Example:

```
USE NORTHWIND
GO

CREATE VIEW custview
AS
SELECT customerid, companyname, phone FROM customers
GO
```

The above statements create a view named custview, which contains the customerid, companyname and the phone columns of the customer table. The following statement can be used to execute a view:

```
SELECT * from custview
```

### Getting View Information

SQL Server stores information in a view in the following system tables.

- ✓ **Sysobjects**-stores the name of the view
- ✓ **Syscolumns**-stores the names of the columns defined in the view.
- ✓ **Sysdepends**-stores information on the view dependencies
- ✓ **Syscomments**-stores the text of the view definition

**Altering a view:**

A view can be modified without dropping it. It ensures that the permissions on the view are not lost. A view can be altered without affecting its dependent objects, such as, triggers and stored procedures.

A view can be modified using the ALTER VIEW statement.

The syntax of ALTER VIEW statement is:

```
ALTER VIEW view_name([column_name])  
[WITH ENCRYPTION]  
AS  
Select_statement  
[WITH CHECK OPTION]
```

**Example:**

```
ALTER VIEW custview  
AS  
SELECT customerid, companyname, phone, fax  
FROM customers
```

*SELECT \* FROM custview*

**Dropping a View:**

A view can be dropped from a database using the DROP VIEW statement. When a view is dropped, it has no effect on the underlying table(s). Dropping a view removes its definition and all permissions assigned to it. Furthermore if users query any views that refer to a dropped view, they receive an error message. However, dropping a table that refers to a view does not drop the view automatically. You must drop it explicitly.

The syntax of DROP VIEW statement is:

*DROP VIEW view\_name*

Where view\_name is the name of the view to be dropped.

It is possible to drop multiple views with a single DROP VIEW. A comma in the DROP statement separates each view that needs to be dropped.

### Renaming a View:

A view can be renamed without dropping it. This also ensures that the permissions on the view are not lost.

The guidelines to be followed for renaming a view as follows.

- ✓ The view must be in the current database.
- ✓ The new name for the view must follow the rules for renaming identifiers.
- ✓ Only the owner of the view and the database in which view is created can rename a view.

A view can be renamed by using the `sp_rename` system stored procedure.

The syntax of `sp_rename` is

```
Sp_rename old_viewname ,new_viewname
```

**Example:**

```
sp_rename custview,customerView
```

Note: All view are not modifiable. If the modification to the view is affecting multiple base tables, then we can't perform such modifications. If the modification to the view is affecting a single base table, then we can perform that modification

### With Encryption:

Without this option in the view, the view design can be viewed by anybody who have access on the view. In order to stop this we can encrypt the view with this option. Once we encrypted the view we can't decrypt again. So, it is important to maintain a copy of the script.

Ex:

```
Create view vSample  
WITH ENCRYPTION  
AS  
    Select sid,sname,student.cid,cname  
    From student inner join class  
    On student.cid=class.cid
```



## With SchemaBinding: (Materialized Views)

A view with an index is called a Materialized view. In order to create an index on the view we must follow these points:

- ✓ View must be created using 'WITH SCHEMABINDING'
- ✓ At first we must have a 'UNIQUE CLUSTERED' index on the view
- ✓ After that only we can create a 'NONCLUSTERED' index.

```
1. Create view vSample
   WITH SCHEMABINDING
   AS
       Select sid,sname,student.cid,cname
       From student inner join class
       On student.cid=class.cid
2. CREATE UNIQUE NONCLUSTERED INDEX isid on vsample(sid)
3. CREATE INDEX iCid on vsample(cid)
```

## With CheckOption:

In general even the modification on the view affects the view criteria, we can perform that modification. To stop this we can use 'WITH CHECK OPTION'.

```
1. Create view vEmp
   As
       Select empid,ename,sal from emp
       Where sal>35000
2. Update vemp set sal=25000 where empid=101
   --no error, even empid 101 contains salary 35000.

3. alter view vemp
   as
       Select empid,ename,sal from emp
       Where sal>35000 WITH CHECK OPTION
4. UPDATE vemp set sal=25000 where empid=104;
   --error, since this employee salary is 39000. If we
   modify this, it will violate the view condition
```

## INDEXES

Indexes in databases are similar to indexes in books. In a book, an index allows you to find information quickly without reading the entire book. In a database, an index allows the database program to find data in a table without scanning the entire table. An index in a book is a list of words with the page numbers that contain each word. An index in a database is a list of values in a table with the storage locations of rows in the table that contain each value. Indexes can be created on either a single column or a combination of columns in a table and are implemented in the form of B-trees. An index contains an entry with one or more columns (the search key) from each row in a table. A B-tree is sorted on the search key, and can be searched efficiently on any leading subset of the search key. For example, an index on columns **A**, **B**, **C** can be searched efficiently on **A**, on **A, B**, and **A, B, C**.

Most books contain one general index of words, names, places, and so on. Databases contain individual indexes for selected types or columns of data: this is similar to a book that contains one index for names of people and another index for places. When you create a database and tune it for performance, you should create indexes for the columns used in queries to find data.

In the **pubs** sample database provided with Microsoft® SQL Server™ 2000, the **employee** table has an index on the **emp\_id** column. The following illustration shows how the index stores each **emp\_id** value and points to the rows of data in the table with each value.

When SQL Server executes a statement to find data in the **employee** table based on a specified **emp\_id** value, it recognizes the index for the **emp\_id** column and uses the index to find the data. If the index is not present, it performs a full table scan starting at the beginning of the table and stepping through each row, searching for the specified **emp\_id** value.

SQL Server automatically creates indexes for certain types of constraints (for example, PRIMARY KEY and UNIQUE constraints). You can further customize the table definitions by creating indexes that are independent of constraints.

The performance benefits of indexes, however, do come with a cost. Tables with indexes require more storage space in the database. Also, commands that insert, update, or delete data can take longer and require more processing

time to maintain the indexes. When you design and create indexes, you should ensure that the performance benefits outweigh the extra cost in storage space and processing resources.

## **Table Indexes**

Microsoft® SQL Server™ 2000 supports indexes defined on any column in a table, including computed columns.

If a table is created with no indexes, the data rows are not stored in any particular order. This structure is called a heap.

The two types of SQL Server indexes are:

- **Clustered**

Clustered indexes sort and store the data rows in the table based on their key values. Because the data rows are stored in sorted order on the clustered index key, clustered indexes are efficient for finding rows. There can only be one clustered index per table, because the data rows themselves can only be sorted in one order. The data rows themselves from the lowest level of the clustered index.

The only time the data rows in a table are stored in sorted order is when the table contains a clustered index. If a table has no clustered index, its data rows are stored in a heap.

- **Nonclustered**

Nonclustered indexes have a structure completely separate from the data rows. The lowest rows of a nonclustered index contain the nonclustered index key values and each key value entry has pointers to the data rows containing the key value. The data rows are not stored in order based on the nonclustered key.

The pointer from an index row in a nonclustered index to a data row is called a row locator. The structure of the row locator depends on whether the data pages are stored in a heap or are clustered. For a heap, a row locator is a pointer to the row. For a table with a clustered index, the row locator is the clustered index key.

The only time the rows in a table are stored in any specific sequence is when a clustered index is created on the table. The rows are then stored in sequence on the clustered index key. If a table only has nonclustered indexes, its data rows are stored in an unordered heap.

Indexes can be unique, which means no two rows can have the same value for the index key. Otherwise, the index is not unique and multiple rows can share the same key value.

There are two ways to define indexes in SQL Server. The CREATE INDEX statement creates and names an index. The CREATE TABLE statement supports the following constraints that create indexes:

- PRIMARY KEY creates a unique index to enforce the primary key.
- UNIQUE creates a unique index.
- CLUSTERED creates a clustered index.
- NONCLUSTERED creates a nonclustered index.

This example shows the Transact-SQL syntax for creating indexes on a table.

```
USE pubs
GO
CREATE TABLE emp_sample
    (emp_id      int      PRIMARY KEY,
     emp_name    char(50),
     emp_address char(50),
     emp_title   char(25)  UNIQUE )
GO

CREATE NONCLUSTERED INDEX sample_nonclust ON
emp_sample(emp_name)
GO
```

## SYNONYMS

A synonym is a database object that serves the following purposes:

- It provides an alternative name for another database object, referred to as the base object, that can exist on a local or remote server.
- It provides a layer of abstraction that protects a client application from changes made to the name or location of the base object.

Synonyms can be created for the following types of objects:

Assembly (CLR) Stored Procedure	Assembly (CLR) Table-valued Function
Assembly (CLR) Scalar Function	Assembly Aggregate (CLR) Aggregate Functions
Replication-filter-procedure	Extended Stored Procedure
SQL Scalar Function	SQL Table-valued Function
SQL Inline-table-valued Function	SQL Stored Procedure
View	Table <sup>1</sup> (User-defined)

### Creating a Synonym:

#### Syn:

```
CREATE SYNONYM [ schema_name_1. ] synonym_name FOR < object >
< object > :: =
{
    [ server_name.[ database_name ] . [ schema_name_2 ] .|
    database_name . [ schema_name_2 ] .| schema_name_2. ]
    object_name
}
```

**Creating a synonym for a local object:**

The following example first creates a synonym for the base object, Product in the AdventureWorks database, and then queries the synonym.

```
USE tempdb;
```

```
GO
```

```
-- Create a synonym for the Product table in AdventureWorks.
```

```
CREATE SYNONYM MyProduct
```

```
FOR AdventureWorks.Production.Product;
```

```
GO
```

```
-- Query the Product table by using the synonym.
```

```
USE tempdb;
```

```
GO
```

```
SELECT ProductID, Name
```

```
FROM MyProduct
```

```
WHERE ProductID < 5;
```

# **NORMALIZATION**

Normalization is a technique for producing a set of relations with desirable properties, given the data requirements of an enterprise.

The process of normalization is a formal method that identifies relations based on their primary or candidate / foreign keys and the functional dependencies among their attributes.

## **The Process of Normalization:**

- Normalization is often executed as a series of steps. Each step corresponds to a specific normal form that has known properties.
- As normalization proceeds, the relations become progressively more restricted in format, and also less vulnerable to update anomalies.
- For the relational data model, it is important to recognize that it is only first normal form (1NF) that is critical in creating relations. All the subsequent normal forms are optional.

Basically 6 normal forms are available. They are:

- 1NF
- 2NF
- 3NF
- BCNF(Boyce-codd NF)
- 4NF
- 5NF

Mostly the realtime database designs ends with 3NF. It means, if we satisfy the rules of 3NF.

## **Case Study:**

There is a consulting company which acts a consultant for property owners and clients who require property for rent.

## **Unnormalized form (UNF)**

A table that contains one or more repeating groups

## ClinetRental

ClientNo	cName	property No	pAddress	rentStart	rentFinish	rent	ownerNo	oName
CR76	John kay	PG4	6 lawrence St,Glasgow	1-Jul-00	31-Aug-01	350	CO40	Tina Murphy
		PG16	5 Novar Dr, Glasgow	1-Sep-02	1-Sep-02	450	CO93	Tony Shaw
CR56	Aline Stewart	PG4	6 lawrence St,Glasgow	1-Sep-99	10-Jun-00	350	CO40	Tina Murphy
		PG36	2 Manor Rd, Glasgow	10-Oct-00	1-Dec-01	370	CO93	Tony Shaw
		PG16	5 Novar Dr, Glasgow	1-Nov-02	1-Aug-03	450	CO93	Tony Shaw

## Definition of 1NF

**First Normal Form** is a relation in which the intersection of each row and column contains one and only one value.

There are two approaches to removing repeating groups from unnormalized tables:

1. Removes the repeating groups by entering appropriate data in the empty columns of rows containing the repeating data.
2. Removes the repeating group by placing the repeating data, along with a copy of the original key attribute(s), in a separate relation. A primary key is identified for the new relation



### 1NF ClientRental relation with the first approach

With the first approach, we remove the repeating group (property rented details) by entering the appropriate client data into each row.

The ClientRental relation is defined as follows,

**ClientRental** ( clientNo, propertyNo, cName, pAddress, rentStart, rentFinish, rent, ownerNo, oName)

ClientNo	Property No	CName	PAddress	RentStart	RentFinish	Rent	OwnerNo	oName
CR76	PG4	John Kay	6 Lawrence, Glasgow	1-Jul-00	31-Aug-01	350	C040	Tina Murphy
CR76	PG16	John Kay	5 NovarDr, Glasgow	1-Sep-02	1-Sep-02	450	C093	Tony Shaw
CR56	PG4	Aline Stewart	6 Lawrence, Glasgow	1-Sep-99	10-Jun-00	350	C040	Tina Murphy
CR56	PG36	Aline Stewart	2 ManorRd, Glasgow	10-Oct-00	1-Dec-01	370	C093	Tony Shaw
CR56	PG16	Aline Stewart	5 NovarDr, Glasgow	1-Nov-02	1-Aug-03	450	C093	Tony Shaw

### 1NF ClientRental relation with the second approach

With the second approach, we remove the repeating group (property rented details) by placing the repeating data along with a copy of the original key attribute (clientNo) in a separate relation.

**Client** (clientNo, cName)

**PropertyRentalOwner** (clientNo, propertyNo, pAddress, rentStart, rentFinish, rent, ownerNo, oName)

#### Client

ClientNo	cName
CR76	PG4
CR76	PG16

#### PropertyRentalOwner

ClientNo	PropertyNo	PAddress	RentStart	RentFinish	Rent	OwnerNo	oName
CR76	PG4	6 Lawrence, Glasgow	1-Jul-00	31-Aug-01	350	C040	Tina Murphy
CR76	PG16	5 NovarDr, Glasgow	1-Sep-02	1-Sep-02	450	C093	Tony Shaw
CR56	PG4	6 Lawrence, Glasgow	1-Sep-99	10-Jun-00	350	C040	Tina Murphy
CR56	PG36	2 ManorRd, Glasgow	10-Oct-00	1-Dec-01	370	C093	Tony Shaw
CR56	PG16	5 NovarDr, Glasgow	1-Nov-02	1-Aug-03	450	C093	Tony Shaw

## Second Normal Form (2NF)

Second normal form (2NF) is a relation that is in first normal form and every non-primary-key attribute is fully functionally dependent on the primary key.

The normalization of 1NF relations to 2NF involves the removal of partial dependencies. If a partial dependency exists, we remove the function dependent attributes from the relation by placing them in a new relation along with a copy of their determinant.

The ClientRental relation has the following functional dependencies:

- |     |   |                         |
|-----|---|-------------------------|
| fd1 | clientNo, propertyNo & rentStart, rentFinish                                    | (Primary Key)           |
| fd2 | clientNo & cName  | (Partial dependency)    |
| fd3 | propertyNo & pAddress, rent, ownerNo, oName                                     | (Partial dependency)    |
| fd4 | ownerNo & oName   | (Transitive Dependency) |
| fd5 | clientNo, rentStart → propertyNo, pAddress,<br>rentFinish, rent, ownerNo, oName | (Candidate key)         |
| fd6 | propertyNo, rentStart & clientNo, cName, rentFinish                             | (Candidate key)         |

## 2NF ClientRental relation

After removing the partial dependencies, the creation of the three new relations called Client, Rental, and PropertyOwner

**Client**            (clientNo, cName)

**Rental** (clientNo, propertyNo, rentStart, rentFinish)

**PropertyOwner** (propertyNo, pAddress, rent, ownerNo, oName)

**Client**

ClientNo	cName
CR76	PG4
CR76	PG16

**Rental**

ClientNo	Property No	RentStart	RentFinish
CR76	PG4	1-Jul-00	31-Aug-01
CR76	PG16	1-Sep-02	1-Sep-02
CR56	PG4	1-Sep-99	10-Jun-00
CR56	PG36	10-Oct-00	1-Dec-01
CR56	PG16	1-Nov-02	1-Aug-03

**Property Owner**

Property No	PAddress	Rent	OwnerNo	oName
PG4	6 Lawrence, Glasgow	350	C040	Tina Murphy
PG16	5 NovarDr, Glasgow	450	C093	Tony Shaw
PG4	6 Lawrence, Glasgow	350	C040	Tina Murphy
PG36	2 ManorRd, Glasgow	370	C093	Tony Shaw
PG16	5 NovarDr, Glasgow	450	C093	Tony Shaw

## Third Normal Form (3NF)

A Relation that is in first and second normal form, and in which no non-primary key attribute is **Transitively** dependent on the primary key

**Transitive dependency** A condition where A, B, and C are attributes of a relation such that if A & B and B & C, then C is transitively dependent on A via B (provided that A is not functionally dependent on B or C).

The normalization of 2NF relations to 3NF involves the removal of transitive dependencies by placing the attribute(s) in a new relation along with a copy of the determinant.

The functional dependencies for the Client, Rental and PropertyOwner relations are as follows:

Client

```
fd2  clientNo  cName                                     (Primary Key)
```

Rental

fd1	clientNo, propertyNo & rentStart, rentFinish	(Primary Key)
-----	--	---------------

fd5	clientNo, rentStart & propertyNo, rentFinish	(Candidate key)
-----	--	-----------------

fd6	propertyNo, rentStart & clientNo, rentFinish	(Candidate key)
-----	--	-----------------

PropertyOwner

fd3	propertyNo & pAddress, rent, ownerNo, oName	(Primary Key)
-----	---	---------------

fd4      ownerNo & oName      (Transitive Dependency)

### 3NF ClientRental relation

The resulting 3NF relations have the forms:

Client                    (clientNo, cName)

Rental (clientNo, propertyNo, rentStart, rentFinish)

```
PropertyOwner (propertyNo, pAddress, rent, ownerNo)
```

Owner (ownerNo, oName)

**Client**

ClientNo	cName
CR76	PG4
CR76	PG16

**Owner**

OwnerNo	OName
C040	Tina Murphy
C093	Tony Shaw

**Rental**

ClientNo	Property No	RentStart	RentFinish
CR76	PG4	1-Jul-00	31-Aug-01
CR76	PG16	1-Sep-02	1-Sep-02
CR56	PG4	1-Sep-99	10-Jun-00
CR56	PG36	10-Oct-00	1-Dec-01
CR56	PG16	1-Nov-02	1-Aug-03

**Property Owner**

Property No	PAddress	Rent	OwnerNo	oName
PG4	6 Lawrence, Glasgow	350	C040	Tina Murphy
PG16	5 NovarDr, Glasgow	450	C093	Tony Shaw
PG4	6 Lawrence, Glasgow	350	C040	Tina Murphy
PG36	2 ManorRd, Glasgow	370	C093	Tony Shaw
PG16	5 NovarDr, Glasgow	450	C093	Tony Shaw

## T-SQL Programming

T-SQL stands for TRANSACT-SQL. It is a programming language, which can be used to develop logic on the database.

Whenever we are trying to execute a set of statements in the query window, those statements will be executed independently. There will be no relation between the statements. In order to have some kind of relation and application logic to those statements, they can be specified under a T-SQL program.

T-SQL is a procedural language which contains a set of SQL statements as a unit. Like any other programming language, T-SQL also follows a predefined structure to develop the programs. This structure contains a 'DECLARE' block used to declare variables and 'BEGIN...END' block to specify the statements or logic to be executed.

```
Syn:
DECLARE
    <Variable Declarations>
BEGIN
    -----
    <Executable Statements>
    -----
END
```

T-SQL supports 2 types of variables. They are:

- Local Variables
- Global Variables

### **Local Variables:**

A local variable can be created by using the DECLARE statement. An initial value can be assigned to the variable with the help of the SELECT statement and can be used within the trigger or procedure where it is created or assigned the value.

### **Global Variables:**

Global variables are pre-defined and maintained by the system. The server to track server-wide and session-specific information uses them. They cannot be explicitly set or declared. Global variables cannot be defined by users and are not used to pass information across processors by applications. Many of the global variables report on system activity since the last time SQL server was started, other report information about a connection.

Some common global variables are:

Global Variable	Description
@@rowcount	Returns the number of rows processed by preceding command.
@@error	Returns the error number of the last error generated.
@@trancount	Returns the transaction nesting level status.
@@servername	Returns the name of the local SQL server.
@@version	Returns the version of the SQL server using.
@@spid	Returns the current process ID.
@@identity	Returns the last identity value used in an insert.
@@nestlevel	Returns the number of level nested in a stored procedure/trigger.
@@fetch_status	Returns a value corresponding to the status of the previous fetch statement in a cursor

### Declaring Variables:

The declare statement can be used to provide variable declaration by specifying the name of the variable and its corresponding data type.

Even though we are declaring two variables of same type, each variable must be declared independently.

```
Syn: DECLARE <varname> <datatype>
```

Ex:

1. Declare @i int
2. Declare @i,@j int --Error
3. Declare @i int,@j int

### Initializing Variables:

We can make use of either SELECT or SET statements to initialize variables.

### Syntax:

```
SELECT @varname=value (or)  
SET @varname=value
```



**Example:**

```
SELECT @name_variable='AKINOVA KURENDIOL', @age=22
```

**PRINT Statement**

The PRINT statement is used to pass a message to the client program's message handler. It is used to display user-defined messages.

The syntax is:

```
PRINT character_string|@local_variable|@@global_variable
```

The message to be displayed using PRINT statement can be up to 255 characters long.

**CONTROL-OF-FLOW LANGUAGE:****BEGIN...END**

When series of statements need to be executed it is better to enclose them in blocks. SQL server provides the BEGIN...END block for this purpose and the statements enclosed between BEGIN and END block are known as statement block. Statement blocks are used with IF...ELSE and WHILE control -of-flow language. If BEGIN and END are not used, only the first statement that immediately follows IF...ELSE or WHILE is executed.

The command syntax is:

```
BEGIN
    {sql_statement | statement_block}
END
```

## IF...ELSE Block

Statements to be executed conditionally are identified with the IF...ELSE construct. The IF...ELSE block allows a statement or statement block to be executed when a condition is TRUE or FALSE.

The command syntax is:

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

IF...ELSE constructs can be used in batches, in stored procedures and in ad hoc queries. IF...ELSE constructs can be nested. There is no limit to the number of nesting levels.

Consider the following example:

```
USE pubs
IF (SELECT SUM(price) FROM titles WHERE type='business') < 600
BEGIN
    PRINT ' The sum of the following BUSINESS books is
less than 600
        Dollars :'
    PRINT ''
    Select * FROM titles WHERE type='business'
END
```

## CASE CONSTRUCT

SQL Server provides a CASE statement where you need a large number of IF statements. The CASE statement enables multiple possible conditions to be managed within a SELECT statement.

The syntax is:

```
CASE
    WHEN Boolean_expression THEN expression1
    [[WHEN Boolean_expression THEN expression][...]]
    [ELSE expression]
END
```

Example:

```
SELECT
    CASE
        WHEN type='BUSINESS' THEN 'BUSINESS BOOK'
        WHEN type='mod_cook' THEN 'MODERN COOKING'
        WHEN type='trad_cook' THEN 'TRADITIONAL COOKING'
        WHEN type='psychology' THEN 'PSYCHOLOGY BOOK'
        ELSE 'No category assigned as yet'
    END
FROM titles
WHERE title_id LIKE 'bu%' OR title_id LIKE 'MC%' OR
title_id LIKE 'PC%' OR title_id LIKE 'PS%' group by type
```

## WHILE CONSTRUCT

The WHILE construct is used for the repeated execution of SQL Statements. The statements are executed repeatedly as long as the specified condition is true.

SQL Server provides BREAK and CONTINUE statements to control the loop from inside of the WHILE construct.

The command syntax is:

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

### Example:

```
WHILE (SELECT AVG (price) FROM titles) < $60
BEGIN
    SELECT title FROM titles
    IF (SELECT MAX (price) FROM titles) > $20
        BREAK
    ELSE
        CONTINUE
END
```

## STORED PROCEDURES

A stored procedure is a group of Transact-SQL statements compiled into a single execution plan.

Stored procedures in SQL Server are similar to procedures in other programming languages in that they can:

- Accept input parameters and return multiple values in the form of output parameters to the calling procedure or batch.
- Contain programming statements that perform operations in the database, including calling other procedures.
- Return a status value to a calling procedure or batch to indicate success or failure (and the reason for failure).

You can use the Transact-SQL EXECUTE statement to run a stored procedure. Stored procedures are different from functions in that they do not return values in place of their names and they cannot be used directly in an expression.

The benefits of using stored procedures in SQL Server rather than Transact-SQL programs stored locally on client computers are:

- They allow modular programming.
- They allow faster execution.
- They can reduce network traffic.
- They can be used as a security mechanism.

### Type of Stored Procedures:

SQL Server supports five types of stored procedures. They are:

#### System Stored Procedures (sp\_)

Many administrative and informational activities SQL Server can be performed through system stored procedures. These system stored procedures are stored in the *Master* database and are identified by the sp\_prefix. They can be executed from any database.

#### Local Stored Procedures

These procedures will be created in the user database. The user who creates the procedure will become the owner for that procedure.

## Temporary Stored Procedures

Temporary stored procedures are stored in **tempdb** database. They can be used in the case where an application builds dynamic Transact-SQL statements that are executed several times. Instead of recompiling the T-SQL statements each time, a temporary stored procedure can be created and compiled on the first execution, then execute the precompiled plan multiple times. The temporary stored procedures can be local or global.

## Remote Stored Procedures

They are legacy feature of SQL Server. Their functionality in T-SQL is limited to executing a stored procedure on a remote SQL Server installation. The distributed queries in SQL Server support this ability along with the ability to access tables on linked OLEDB data sources directly from local T-SQL statements.

## Extended Stored Procedures

These are dynamic link libraries (DLLs) that SQL Server can dynamically load and execute. These procedures run directly in the address space of SQL Server and are programmed using the SQL Server Open Data Services API. They are identified by the xp\_prefix.

## Creating a Stored Procedure:

The stored procedures can be created using the **CREATE PROCEDURE** statement.

### Syntax:

```
CREATE PROCEDURE procedure_name
[( @parameter1 data_type [OUTPUT] [, @parameter2 ....])]
AS
BEGIN
    SQL-Statements
    [RETURN]
END
```

### Types of Parameters:

SQL Server provides two types of parameters.

- Input parameters
- Output parameters

Input parameters will be used to pass a value using the procedure call. Based on this value, the procedure execution will be done.

Output parameters will be used to assign a value inside the procedure definition which can be used by the calling program. Before execution of the stored procedure, output parameter doesn't contain any value. Once the procedure execution completes, now the output parameter gets some value.

### Example: creating a procedure to insert values into Products table.

```
Create procedure pInsertProductst
    (@pid int,@pn char(10),@pqty int)
AS
BEGIN
    Insert into products values (@pid,@pn,@pqty)
END
```

### Executing the Procedure:

We can execute a procedure with EXEC statement by specifying the procedure name and parameters list.

If procedure call is only the statement to be executed, then no need to use exec statement. We can directly use procedure name and parameter list.

Whereas, if we are calling the procedure from another procedure or T-SQL program, then we must use the EXEC statement to call the procedure.

```
Syn: EXEC <Proc.Name> [val1,val2,var1 output,...]
```

```
Ex: exec pInsertProducts(1,'P1',200)
```

**Ex: using output parameters**

Write a program to calculating the following details for a given employee number.

- Basic Salary
- Benefits(Commission percentage on salary)
- Net Salary

Solution:

1. Creating a procedure to calculate Benefits

```
create procedure emp_benefits(@eno int,@add int output)
as
    select @add=comm*sal*0.01 from emp where empid=@eno
```

2. Creating a procedure to calculate Net Salary

```
create procedure emp_netsalary(@eid int)
as
BEGIN
    declare @bsal int,@add int,@nsal int
    exec emp_benefits @eid,@add output
    select @bsal=sal from emp
        where empid=@eid
    set @nsal=@bsal+@add
    select Empid=@eid,BasicSalary=@bsal,
        Benefits=@add,NetSalary=@nsal
END
```

**Execution:**

Exec emp\_netsalary 100

Output:

Empid	BasicSalary	Benefits	NetSalary
100	15000	1500	16500

### Error Handling in Stored Procedures:

In order to not get any runtime errors it is important to handle errors in the stored procedure. Using a 'Begin Try' and 'Begin Catch' blocks error handling can be done in SQL Server.

Syn:

```
create procedure <procedurename> (Prameter1,...)
as
BEGIN
    Begin Try
        ----
        <Executable statements>
    End Try
    Begin Catch
        --error handling
    End Catch
END
```

Example:

```
Create procedure pInsertProductst
    (@pid int,@pn char(10),@pqty int)
AS
BEGIN
    Begin Try
        Insert into products values(@pid,@pn,@pqty)
    End Try
    Begin Catch
        If ERROR_NUMBER()=2627
            PRINT 'Duplicate product id value entered'
        Else
            PRINT ERROR_MESSAGE()
    End Catch
END
```



## TRIGGERS

A trigger is a special type of stored procedure that automatically takes effect when the data in a specified table is modified. Triggers are invoked in response to an INSERT, UPDATE, or DELETE statement. A trigger can query other tables and can include complex Transact-SQL statements. The trigger and the statement that fires it are treated as a single transaction, which can be rolled back from within the trigger. If a severe error is detected (for example, insufficient disk space), the entire transaction automatically rolls back.

Triggers are useful in these ways:

- Triggers can cascade changes through related tables in the database; however, these changes can be executed more efficiently using cascading referential integrity constraints.
- Triggers can enforce restrictions that are more complex than those defined with CHECK constraints.

Unlike CHECK constraints, triggers can reference columns in other tables. For example, a trigger can use a SELECT from another table to compare to the inserted or updated data and to perform additional actions, such as modify the data or display a user-defined error message.

- Triggers can also evaluate the state of a table before and after a data modification and take action(s) based on that difference.
- Multiple triggers of the same type (INSERT, UPDATE, or DELETE) on a table allow multiple, different actions to take place in response to the same modification statement.

### Creating a Trigger:

```
CREATE TRIGGER trigger_name
ON { table | view }
{
    { { FOR | AFTER | INSTEAD OF } { [ INSERT ] [ , ] [ UPDATE
] }
    AS
        sql_statement [ ...n ]
}
}
```

You can use the FOR clause to specify when a trigger is executed:

- AFTER

The trigger executes after the statement that triggered it completes. If the statement fails with an error, such as a constraint violation or syntax error, the trigger is not executed. AFTER triggers cannot be specified for views, they can only be specified for tables. You can specify multiple AFTER triggers for each triggering action (INSERT, UPDATE, or DELETE). If you have multiple AFTER triggers for a table, you can use **sp\_settriggerorder** to define which AFTER trigger fires first and which fires last. All other AFTER triggers besides the first and last fire in an undefined order which you cannot control.

AFTER is the default in SQL Server 2000. You could not specify AFTER or INSTEAD OF in SQL Server version 7.0 or earlier, all triggers in those versions operated as AFTER triggers.

- INSTEAD OF

The trigger executes in place of the triggering action. INSTEAD OF triggers can be specified on both tables and views. You can define only one INSTEAD OF trigger for each triggering action (INSERT, UPDATE, and DELETE). INSTEAD OF triggers can be used to perform enhance integrity checks on the data values supplied in INSERT and UPDATE statements. INSTEAD OF triggers also let you specify actions that allow views, which would normally not support updates, to be updatable.

### Triggers Compared to Constraints

Constraints and triggers each have benefits that make them useful in special situations. The primary benefit of triggers is that they can contain complex processing logic that uses Transact-SQL code. Therefore, triggers can support all of the functionality of constraints; however, triggers are not always the best method for a given feature.

*Entity integrity* should always be enforced at the lowest level by indexes that are part of PRIMARY KEY and UNIQUE constraints or are created independently of constraints. Domain integrity should be enforced through CHECK constraints, and *referential integrity (RI)* should be enforced through FOREIGN KEY constraints, assuming their features meet the functional needs of the application.

Triggers are most useful when the features supported by constraints cannot meet the functional needs of the application. For example:

- FOREIGN KEY constraints can validate a column value only with an exact match to a value in another column, unless the REFERENCES clause defines a cascading referential action.

- A CHECK constraint can validate a column value only against a logical expression or another column in the same table. If your application requires that a column value be validated against a column in another table, you must use a trigger.
- Constraints can communicate about errors only through standardized system error messages. If your application requires (or can benefit from) customized messages and more complex error handling, you must use a trigger.

Triggers can cascade changes through related tables in the database; however, these changes can be executed more efficiently through cascading referential integrity constraints.

- Triggers can disallow or roll back changes that violate referential integrity, thereby canceling the attempted data modification. Such a trigger might go into effect when you change a foreign key and the new value does not match its primary key.
- If constraints exist on the trigger table, they are checked after the INSTEAD OF trigger execution but prior to the AFTER trigger execution. If the constraints are violated, the INSTEAD OF trigger actions are rolled back and the AFTER trigger is not executed.

#### **Example:**

While entering the details of a new employee, the salary has to be entered according to the grade. If the grade it changed, the basic salary should also be changed accordingly. Instead of manually doing this, the HR manager wanted the basic salary of the employees to be entered automatically whenever an employee is added or grade is changed.

**Solution:** Create sal\_info and emp tables as follows.

```
CREATE TABLE sal_info(grade char(1), bsal numeric(18,0))
```

Insert some values into the sal\_info table:

```
INSERT INTO sal_info
SELECT 'A',1000 UNION ALL
SELECT 'B',2000 UNION ALL
SELECT 'C',3000
```

Now, create the emp table as follows:

```
CREATE TABLE emp(emp_no int,emp_name varchar(10),dept_no int,
grade char(1),bsal numeric(18,0),doj datetime)
```

Now, create the required trigger as follows:

```
CREATE TRIGGER tr_emp ON emp
FOR INSERT,UPDATE
AS
DECLARE @sal numeric(18,0)
SELECT @sal=sal_info.bsal from sal_info,inserted
Where inserted.grade=sal_info.grade

UPDATE emp set bsal=@sal from emp,inserted
Where emp.emp_no=inserted.emp_no
```

An *insert/update* is written for the table *emp*. Whenever a new record is inserted or updated, the new grade is obtained from the inserted table. The corresponding basic salary is obtained from the table *sal\_info* and the basic salary in the *emp* table is set to this value.

This trigger can be checked by inserting a record with a null value for the field *bsal*. When a select statement is given, the value for *bsal* will also be present.

```
INSERT INTO emp VALUES(100,'Arvind',30,'B',null,getdate())
```

```
SELECT * FROM EMP
```

Emp_no	emp_name	dept_no	grade	bsal	doj
100	Arvind	30	B	2000	2006-06-15

## Programming DDL Triggers:

### Syntax:

```
CREATE TRIGGER trigger_name
ON { ALL SERVER | DATABASE }
[ WITH <ddl_trigger_option> [ ,...n ] ]
{ FOR | AFTER } { event_type | event_group } [ ,...n ]
AS { sql_statement [ ; ] [ ...n ] | EXTERNAL NAME < method
specifier > [ ; ] }
```

### Example:

The following example illustrates how a DDL trigger can be used to prevent any table in a database from being modified or dropped:

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

## Altering a Trigger:

The definition of an existing trigger can be altered without dropping it. The altered definition replaces the definition of the existing trigger with the new definition.

### Syntax

```
ALTER TRIGGER trigger_name
ON { table | view }
{
    { { FOR | AFTER | INSTEAD OF } { [ INSERT ] [ , ] [ UPDATE
] }
    AS
        sql_statement [ ...n ]
    }
}
```

**Example:**

This example creates a trigger that prints a user-defined message to the client when a user tries to add or change data in the emp table. Then, the trigger is altered using ALTER TRIGGER to apply the trigger only on INSERT activates. This trigger is helpful because it reminds the user who updates or inserts rows into this table.

```
CREATE TRIGGER trig1
ON emp
WITH ENCRYPTION
FOR INSERT, UPDATE
AS RAISERROR(40008,16,10)
```

```
ALTER TRIGGER trig1
ON emp
FOR INSERT
AS RAISERROR(40008,16,10)
```

**Dropping a Trigger:**

A trigger can be dropped by using, DROP TRIGGER statement. A trigger gets dropped automatically when its associated table is dropped. Permissions to drop a trigger defaults to the table owner. But the members of the system administrators(*sysadmin*) and the database owner(*db\_owner*) can drop any object by specifying the owner in the DROP TRIGGER statement.

**Syntax:**

```
DROP TRIGGER trigger_name
```

**Disabling or Enabling a Trigger:**

Triggers can be enabled or disabled using ALTER TABLE statement. On disabling a trigger, the actions in the trigger are not performed until the trigger is re-enabled.

**Syntax:**

```
ALTER TABLE table_name
{ENABLE | DISABLE } TRIGGER
{ALL | trigger_name[, --n]}
```

## Setting the Trigger Order:

The **'sp\_settriggerorder'** is used to specify the order of firing the trigger.

The AFTER triggers that are fired between the first and last triggers are executed in undefined order

Syntax:

```
sp_settriggerorder [ @triggername = ] '[ triggerschema. ]  
triggername'  
    , [ @order = ] 'value'  
    , [ @stmttype = ] 'statement_type'  
    [ , [ @namespace = ] { 'DATABASE' | 'SERVER' | NULL  
    } ]
```

The following are the values for the order

Value	Description
First	Trigger is fired first.
Last	Trigger is fired last.
None	Trigger is fired in undefined order.

Example:

1. Setting the order of DML triggers

```
USE AdventureWorks;  
GO  
sp_settriggerorder @triggername= 'Sales.uSalesOrderHeader',  
@order='First', @stmttype = 'UPDATE';
```

2. Setting the order of DDL triggers

```
USE AdventureWorks;  
GO  
sp_settriggerorder @triggername= 'ddlDatabaseTriggerLog',  
@order='First', @stmttype = 'ALTER_TABLE', @namespace =  
'DATABASE';
```

## USER DEFINED FUNCTIONS

A User-Defined Function is, much like a Stored Procedure, an ordered set of T-SQL statements that are pre-optimized and compiled and can be called to work as a single unit. The primary difference between them is how results are returned.

With a Stored Procedure, you can pass parameters in, and also get values in parameters passed back out. You can return a value, but that value is really intended to indicate success or failure rather than return data. You can also return result sets, but you can't really use those result sets in a query without first inserting them into some kind of table (usually a temporary table) to work with them further.

With a UDF, however, you can pass parameters *in*, but not *out*. Instead, the concept of output parameters has been replaced with a much more robust return value. As with system functions, you can return a scalar value. Another advantage is that this value is not limited to just the integer data type as it would be for a Stored Procedure. Instead, you can return most SQL Server data types.

### Types of UDF's:

- Those that return a scalar value
- Those that return a table

### General Syntax for creating a UDF:

```
CREATE FUNCTION function_name
(<@parameter name> <data type>[=default value][,.....n])
RETURNS {<data type>|TABLE}
AS
BEGIN
    [<function statements>]
    {RETURN <type as defined in RETURNS clause> |
    RETURN (<SELECT statement>)}
END
```



### UDFs Returning a Scalar Value:

Much like SQL Server's own built-in functions, they will return a scalar value to the calling script or procedure. One of the truly great things about a UDF is that you are not limited to an integer for a return value – instead, it can be of any valid SQL Server data type, except for BLOBs, cursors, and timestamps. Even if you wanted to return an integer, a UDF should be advantage to you for two different reasons:

1. Unlike Stored Procedures, the whole purpose of the return value is to serve a meaningful piece of data – for Stored Procedures, a return value is meant as an indication of success or failure, and, in the event of failure, to provide some specific information about the nature of that failure.
2. You can perform functions in-line to your queries (for instances, include it as part of your SELECT statement) – you can't do that with a Stored Procedure.

### Case study:

Let's create a table called ORDERS with 3 columns named as OrderID, CustID and OrderDate as follows

```
CREATE TABLE orders(ordered int, custid int,orderdate datetime)
```

Lets insert some data into the table with the use of a simple T-SQL program.

```
DECLARE @Counter int

SET @Counter =1
WHILE @Counter <= 10
BEGIN
    INSERT INTO Orders
        VALUES(1,1,DATEADD(mi, @Counter, GETDATE()))
    SET @Counter = @Counter + 1
END
```

So, this gets us 10 rows inserted, with each row being inserted with today's date, but one minute apart from each other.

So, now we're ready to run a simple query to see what orders we have today. We might try something like:

```
SELECT *  
FROM Orders  
WHERE orderdate=GETDATE()
```

Unfortunately, this query will not get us anything back at all. This is because GETDATE() gets the current time not just the day.

The solution is to convert the date to a string and back in order to truncate the time information, then perform comparison.

```
SELECT *  
FROM Orders  
WHERE CONVERT(varchar(12), orderdate,  
101)=CONVERT(varchar(12), GETDATE(), 101)
```

This time, we will get back every row with today's date in the OrderDate column – regardless of what time of day the order was taken. Unfortunately, this isn't exactly the most readable code. Imagine you had a large series of dates you needed to perform such comparisons against – it can get very ugly indeed.

So now let's look at doing the same thing with a simple user-defined function. First, we'll need to create the actual function. This is done with the new CREATE FUNCTION command, and it's formatted much like a Stored Procedure. For example, we might code this function like this:

```
CREATE FUNCTION dbo.DateOnly(@dt datetime)  
RETURNS varchar(12)  
AS  
BEGIN  
    RETURN CONVERT(varchar(12), @dt, 101)  
END
```

Whether the date returned from GETDATE() is passed in as the parameter and the task of converting the date is included in the function body and the truncated date is returned.

To see this function in action, let's reformat our query as follows:

```
SELECT *
FROM Orders
WHERE dbo.DateOnly(OrderDate) = dbo.DateOnly(GETDATE())
```

We get back the same set as with the stand-alone query. Even for a simple query like this one, the new code is quite a bit more readable. There is, however, one requirement for this type. The owner name is required in the function call. SQL Server will, for some reason, not resolve functions the way it does with other objects.

### **UDFs that Return a Table:**

SQL Server's new user-defined functions are not limited to just returning scalar values. They can return something far more interesting – tables.

To make the change to using a table, as a return value is not hard at all – a table is just like any other SQL Server data type as far as a UDF is concerned. To illustrate this, we'll build a relatively simple one:

```
USE pubs
GO

CREATE FUNCTION dbo.fnListOfAuthors()
RETURNS TABLE
AS
RETURN ( SELECT au_id,
                au_lname + ', ' + au_fname AS au_name,
                address AS address1,
                City + ', ' + state + ' ' + zip AS address2
          FROM authors)
GO
```

This function returns a table of SELECTEd records and does a little formatting: joining the last and first names, separating them with a comma, and concatenating the three components to fill the address2 column.

At this point, we're ready to use our function just as we would use a table – the only exception is that as was discussed with scalar functions, we must use the two-part naming convention:

```
SELECT *  
FROM dbo.fnListOfAuthors()
```

# TRANSACTIONS

A transaction is a sequence of operations performed as a single logical unit of work. A logical unit of work must exhibit four properties, called the ACID (Atomicity, Consistency, Isolation, and Durability) properties, to qualify as a transaction:

## **Atomicity**

A transaction must be an atomic unit of work; either all of its data modifications are performed, or none of them is performed.

## **Consistency**

When completed, a transaction must leave all data in a consistent state. In a relational database, all rules must be applied to the transaction's modifications to maintain all data integrity. All internal data structures, such as B-tree indexes or doubly-linked lists, must be correct at the end of the transaction.

## **Isolation**

Modifications made by concurrent transactions must be isolated from the modifications made by any other concurrent transactions. A transaction either sees data in the state it was in before another concurrent transaction modified it, or it sees the data after the second transaction has completed, but it does not see an intermediate state. This is referred to as serializability because it results in the ability to reload the starting data and replay a series of transactions to end up with the data in the same state it was in after the original transactions were performed.

## **Durability**

After a transaction has completed, its effects are permanently in place in the system. The modifications persist even in the event of a system failure.

## **Controlling Transactions**

Applications control transactions mainly by specifying when a transaction starts and ends. This can be specified using either Transact-SQL statements or database API functions. The system must also be able to correctly handle errors that terminate a transaction before it completes.

Transactions are managed at the connection level. When a transaction is started on a connection, all Transact-SQL statements executed on that connection are part of the transaction until the transaction ends.

## Starting Transactions

You can start transactions in Microsoft® SQL Server™ as explicit, autocommit, or implicit transactions.

### Explicit transactions

Explicitly start a transaction by issuing a *BEGIN TRANSACTION* statement.

### Autocommit transactions

This is the default mode for SQL Server. These transactions will contain only one statement. Each individual Transact-SQL statement is committed when it completes. You do not have to specify any statements to control transactions.

### Implicit transactions

Set implicit transaction mode on through either an API function or the Transact-SQL *SET IMPLICIT\_TRANSACTIONS ON* statement. When that transaction is completed either by using *COMMIT* / *ROLLBACK*, the next Transact-SQL statement starts a new transaction.

## Ending Transactions

You can end transactions with either a *COMMIT* or *ROLLBACK* statement.

### COMMIT

If a transaction is successful, commit it. A *COMMIT* statement guarantees all of the transaction's modifications are made a permanent part of the database. A *COMMIT* also frees resources, such as locks, used by the transaction.

#### Syntax:

```
COMMIT TRAN[SACTION] transactionName
```

### ROLLBACK

If an error occurs in a transaction, or if the user decides to cancel the transaction, then roll the transaction back. A *ROLLBACK* statement backs out all modifications made in the transaction by returning the data to the state it was in at the start of the transaction. A *ROLLBACK* also frees resources held by the transaction.

#### Syntax:

```
ROLLBACK TRAN[SACTION] transactionName [savepointname]
```

## Savepoints

Savepoints offer a mechanism to roll back portions of transactions. You create a savepoint using the `SAVE TRANSACTION savepoint_name` statement, and then later execute a `ROLLBACK TRANSACTION savepoint_name` statement to roll back to the savepoint instead of rolling back to the start of a transaction.

**Example:** Let us suppose emp table contains the following data.

Empid	Ename	Sal
100	Anil	1500
101	Balu	2000
102	Vivek	3500

```
1. BEGIN TRAN
2. SELECT * FROM emp
3. DELETE FROM emp WHERE empid=102
4. SELECT * FROM emp
5. SAVE TRAN sp1
6. UPDATE emp SET sal=4000 WHERE empid=101
7. SELECT * FROM emp
8. ROLLBACK TRAN sp1
9. SELECT * FROM emp
10. ROLLBACK TRAN
11. SELECT * FROM emp
```

102	Vivek	3500
100	Anil	1500
101	Balu	2000
100	Anil	1500
101	Balu	4000
100	Anil	1500
101	Balu	2000
100	Anil	1500
101	Balu	2000
102	Vivek	3500

## LOCKS

Microsoft® SQL Server™ 2000 has locking that allows different types of resources to be locked by a transaction. To minimize the cost of locking, SQL Server locks resources automatically at a level appropriate to the task. Locking at a smaller granularity, such as rows, increases concurrency, but has a higher overhead because more locks must be held if many rows are locked. Locking at a larger granularity, such as tables, are expensive in terms of concurrency because locking an entire table restricts access to any part of the table by other transactions, but has a lower overhead because fewer locks are being maintained.

SQL Server can lock these resources (listed in order of increasing granularity).

Resource	Description
RID	Row identifier. Used to lock a single row within a table.
Key	Row lock within an index. Used to protect key ranges in serializable transactions.
Page	8 kilobyte –(KB) data page or index page.
Extent	Contiguous group of eight data pages or index pages.
Table	Entire table, including all data and indexes.
DB	Database.

SQL Server locks resources using different lock modes that determine how the resources can be accessed by concurrent transactions.

SQL Server uses these resource lock modes.

Lock mode	Description
Shared (S)	Used for operations that do not change or update data (read-only operations), such as a SELECT statement.
Update (U)	Used on resources that can be updated. Prevents a common form of deadlock that occurs when multiple sessions are reading, locking, and potentially updating resources later.



Exclusive (X)	Used for data-modification operations, such as INSERT, UPDATE, or DELETE. Ensures that multiple updates cannot be made to the same resource at the same time.
Intent	Used to establish a lock hierarchy. The types of intent locks are: intent shared (IS), intent exclusive (IX), and shared with intent exclusive (SIX).
Schema	Used when an operation dependent on the schema of a table is executing. The types of schema locks are: schema modification (Sch-M) and schema stability (Sch-S).
Bulk Update (BU)	Used when bulk-copying data into a table and the <b>TABLOCK</b> hint is specified.

## Shared Locks

Shared (S) locks allow concurrent transactions to read (SELECT) a resource. No other transactions can modify the data while shared (S) locks exist on the resource. Shared (S) locks on a resource are released as soon as the data has been read, unless the transaction isolation level is set to repeatable read or higher, or a locking hint is used to retain the shared (S) locks for the duration of the transaction.

## Update Locks

Update (U) locks prevent a common form of deadlock. A typical update pattern consists of a transaction reading a record, acquiring a shared (S) lock on the resource (page or row), and then modifying the row, which requires lock conversion to an exclusive (X) lock. If two transactions acquire shared-mode locks on a resource and then attempt to update data concurrently, one transaction attempts the lock conversion to an exclusive (X) lock. The shared-mode-to-exclusive lock conversion must wait because the exclusive lock for one transaction is not compatible with the shared-mode lock of the other transaction; a lock wait occurs. The second transaction attempts to acquire an exclusive (X) lock for its update. Because both transactions are converting to exclusive (X) locks, and they are each waiting for the other transaction to release its shared-mode lock, a deadlock occurs.

To avoid this potential deadlock problem, update (U) locks are used. Only one transaction can obtain an update (U) lock to a resource at a time. If a transaction

modifies a resource, the update (U) lock is converted to an exclusive (X) lock. Otherwise, the lock is converted to a shared-mode lock.

## Exclusive Locks

Exclusive (X) locks prevent access to a resource by concurrent transactions. No other transactions can read or modify data locked with an exclusive (X) lock.

## Intent Locks

An intent lock indicates that SQL Server wants to acquire a shared (S) lock or exclusive (X) lock on some of the resources lower down in the hierarchy. For example, a shared intent lock placed at the table level means that a transaction intends on placing shared (S) locks on pages or rows within that table. Setting an intent lock at the table level prevents another transaction from subsequently acquiring an exclusive (X) lock on the table containing that page. Intent locks improve performance because SQL Server examines intent locks only at the table level to determine if a transaction can safely acquire a lock on that table. This removes the requirement to examine every row or page lock on the table to determine if a transaction can lock the entire table.

Intent locks include intent shared (IS), intent exclusive (IX), and shared with intent exclusive (SIX).

Lock mode	Description
Intent shared (IS)	Indicates the intention of a transaction to read some (but not all) resources lower in the hierarchy by placing S locks on those individual resources.
Intent exclusive (IX)	Indicates the intention of a transaction to modify some (but not all) resources lower in the hierarchy by placing X locks on those individual resources. IX is a superset of IS.
Shared with intent exclusive (SIX)	Indicates the intention of the transaction to read all of the resources lower in the hierarchy and modify some (but not all) resources lower in the hierarchy by placing IX locks on those

	individual resources. Concurrent IS locks at the top-level resource are allowed. For example, an SIX lock on a table places an SIX lock on the table (allowing concurrent IS locks), and IX locks on the pages being modified (and X locks on the modified rows). There can be only one SIX lock per resource at one time, preventing updates to the resource made by other transactions, although other transactions can read resources lower in the hierarchy by obtaining IS locks at the table level.
--	---

## Schema Locks

Schema modification (Sch-M) locks are used when a table data definition language (DDL) operation (such as adding a column or dropping a table) is being performed.

Schema stability (Sch-S) locks are used when compiling queries. Schema stability (Sch-S) locks do not block any transactional locks, including exclusive (X) locks. Therefore, other transactions can continue to run while a query is being compiled, including transactions with exclusive (X) locks on a table. However, DDL operations cannot be performed on the table.

## Bulk Update Locks

Bulk update (BU) locks are used when bulk copying data into a table and either the **TABLOCK** hint is specified or the **table lock on bulk load** table option is set using **sp\_tableoption**. Bulk update (BU) locks allow processes to bulk copy data concurrently into the same table while preventing other processes that are not bulk copying data from accessing the table.

## Deadlocking

A deadlock occurs when there is a cyclic dependency between two or more threads for some set of resources.

Deadlock is a condition that can occur on any system with multiple threads, not just on a relational database management system. A thread in a multi-threaded system may acquire one or more resources (for example, locks). If the resource being acquired is currently owned by another thread, the first thread may have to

wait for the owning thread to release the target resource. The waiting thread is said to have a dependency on the owning thread for that particular resource.

If the owning thread wants to acquire another resource that is currently owned by the waiting thread, the situation becomes a deadlock: both threads cannot release the resources they own until their transactions are committed or rolled back, and their transactions cannot be committed or rolled back because they are waiting on resources the other owns. For example, thread T1 running transaction 1 has an exclusive lock on the **Supplier** table. Thread T2 running transaction 2 obtains an exclusive lock on the **Part** table, and then wants a lock on the **Supplier** table. Transaction 2 cannot obtain the lock because transaction 1 has it. Transaction 2 is blocked, waiting on transaction 1. Transaction 1 then wants a lock on the **Part** table, but cannot obtain it because transaction 2 has it locked. The transactions cannot release the locks held until the transaction is committed or rolled back. The transactions cannot commit or roll back because they require a lock held by the other transaction to continue.

## CURSORS

A cursor is a pointer to the result of a SELECT statement. The following are the features provided by a cursor.

- Allowing positioning at specific rows of the result set.
- Retrieving one row or block of rows from the current position in the result set.
- Supporting data modifications to the rows at the current position in the result set.
- Supporting different levels of visibility to changes made by other users to the database data that is presented in the result set.
- Providing Transact-SQL statements in scripts, stored procedures, and triggers access to the data in a result set.

### Types of Cursors:

- Static cursors
- Dynamic cursors
- Forward-only cursors
- Keyset-driven cursors

Static cursors detect few or no changes but consume relatively few resources while scrolling, although they store the entire cursor in **tempdb**. Dynamic cursors detect all changes but consume more resources while scrolling, although they make the lightest use of **tempdb**. Keyset-driven cursors lie in between, detecting most changes but at less expense than dynamic cursors.

### Cursor Statements:

We can control the records of a result set with the use of these cursor statements. They are:

- DECLARE
- OPEN
- FETCH
- CLOSE
- DEALLOCATE

**DECLARE** Statement is used to provide cursor declarations such as name Of the cursor, the select statement to which result set the cursor should point etc.,

**OPEN** Statement starts using the cursor. It executes the select statement in the cursor and points the cursor to the first record of the result set.

**FETCH** statement is used to retrieve the data in the current location and navigate the cursor to the required position.

**CLOSE** statement stops using the cursor. But the resources used by the cursor are still open.

**DEALLOCATE** statements removes entire resources that are used by the cursor

**@@FETCH\_STATUS**:Returns the status of the last cursor FETCH statement issued against any cursor currently opened by the connection.

Return value	Description
0	FETCH statement was successful.
-1	FETCH statement failed or the row was beyond the result set.
-2	Row fetched is missing.

Example:

Suppose there is a table with some data and indexes. After the index creation, if any modifications are performed on the table, those changes will not reflect directly into the index structure. We have to refresh the index structure on all table using the following statement.

```
DBCC DBREINDEX (<tablename>)
```

The above command should be executed for all tables of the database. To perform this task we can make use of the following cursor program:

```
declare @tblname varchar(255)

declare tblcursor cursor
for select table_name from information_schema.tables
where table_type='base table'

open tblcursor

fetch next from tblcursor into @tblname

while @@fetch_status=0
begin
    dbcc dbreindex(@tblname)
    fetch next from tblcursor into @tblname
end

close tblcursor
deallocate tblcursor
```

## **SECURITY**

SQL Server validates users at two levels of security on database user accounts and roles.

- Login Authentication
- Permissions Validation

The authentication stage identifies the user using a login account and verifies only the ability to connect with SQL Server. If the authentication is successful, the user is connected to SQL Server. The user then needs permissions to access database on the server, which is done by using an account in each database, mapped to the user login.

The permission validation stage controls the activities the user is allowed to perform in the SQL Server database.

### **Login Authentications:**

A user must have a login account to connect to SQL Server. SQL Server provides two types of Login Authentications.

- Windows NT Authentication
- SQL Server Authentication

### **Windows NT Authentication:**

When using Windows NT Authentication, the user is not required to specify a login ID or password to connect to SQL Server. The user's access to SQL Server is controlled by the Windows NT account, which is authenticated when he/she logs on to the Windows Operating System.

### **SQL Server Authentication:**

When using SQL Server authentication, the users must supply the SQL Server login and password to connect to SQL Server. The users are identified in SQL Server by their SQL Server login.

### **Authentication Modes:**

SQL Server can operate in two security modes:

**Windows NT Authentication Mode:** Only Windows NT Authentication is allowed. Users cannot specify SQL Server Authentication.



**Mixed Mode:** It allows users to connect to SQL Server using Windows NT Authentication or SQL Server Authentication.

### Create a User Login:

- Expand the server
- Expand security
- Right click on "**Logins**" and select "**New Login...**"
- In the window displayed enter a name to the user under **Name** Textbox
- Click the **SQL Server Authentication** option button to provide the password
- Enter "Password" and "Confirm Password"
- Uncheck "Enforce Password Policy" option.

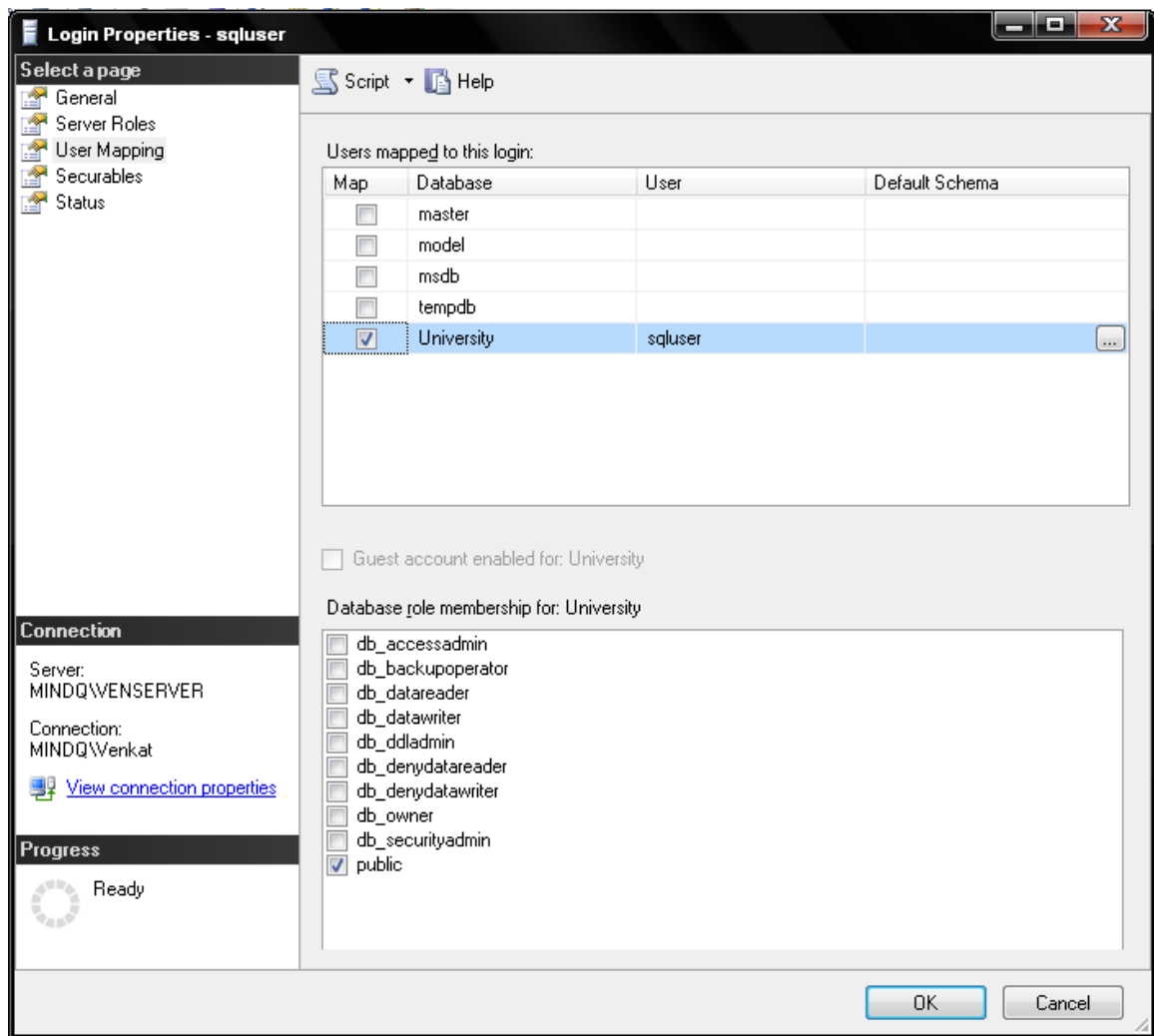
The screenshot shows the 'Login - New' dialog box with the following details:

- General Tab:**
  - Login name:** sqluser
  - Authentication:** ☒ SQL Server authentication
  - Password:** [masked]
  - Confirm password:** [masked]
  - Enforce password policy:** ☐ (unchecked)
  - Enforce password expiration:** ☐ (unchecked)
  - User must change password at next login:** ☐ (unchecked)
  - Mapped to certificate:** ☒ (selected, indicated by a red arrow)
  - Certificate name:** [empty field]
  - Mapped to asymmetric key:** ☐ (unselected)
  - Key name:** [empty field]
  - Default database:** master
  - Default language:** <default>
- Connection:**
  - Server: MINDQ\WENSERVER
  - Connection: MINDQ\Wenkat
  - [View connection properties](#)
- Progress:** Ready
- Buttons:** OK, Cancel

- Click "**OK**"

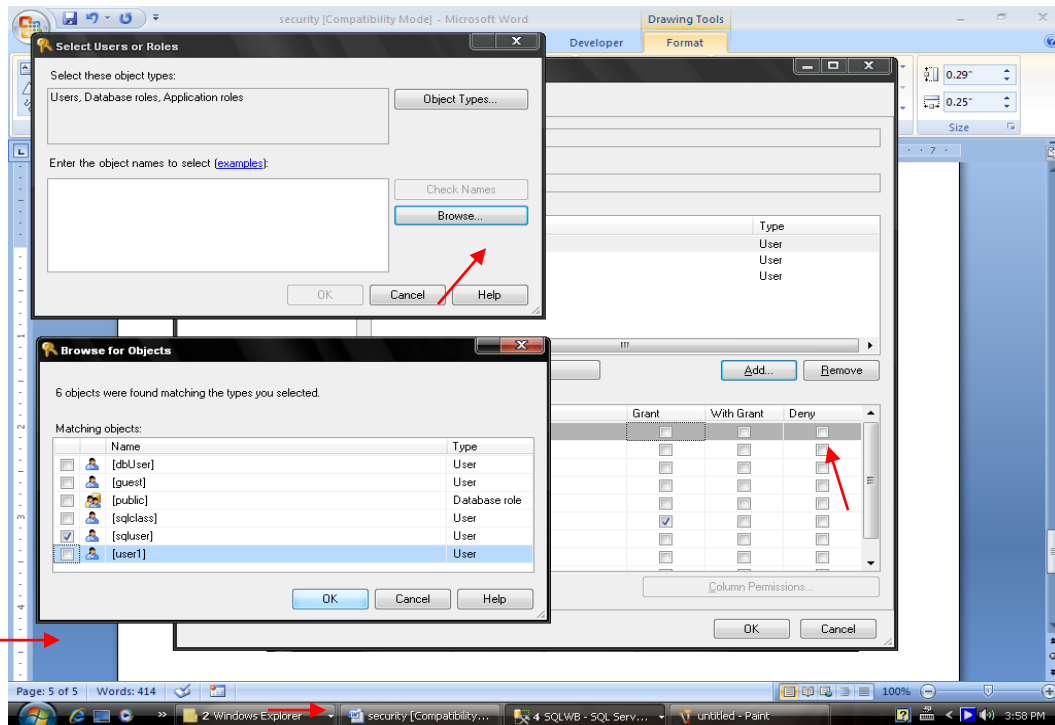
## Providing access to a database for a user:

- Right click on the login name and select properties
- In the window displayed click on **"User Mappings"** tab
- Make a check mark on the database for which the access has to be given.
- Click **"OK"**



## Granting Permissions on a table:

- Right click on table name and select **"Properties"**
- In the window displayed click on **"Permissions"**
- In the window displayed click on "Add" button, "Browse" for the user, and make a check mark for the required user name and click "ok"



- Make a check mark for the desired operation.
- Click "OK"

## BACKING UP AND RESTORING DATABASES

The backup and restore component of Microsoft® SQL Server™ 2000 provides an important safeguard for protecting critical data stored in SQL Server databases.

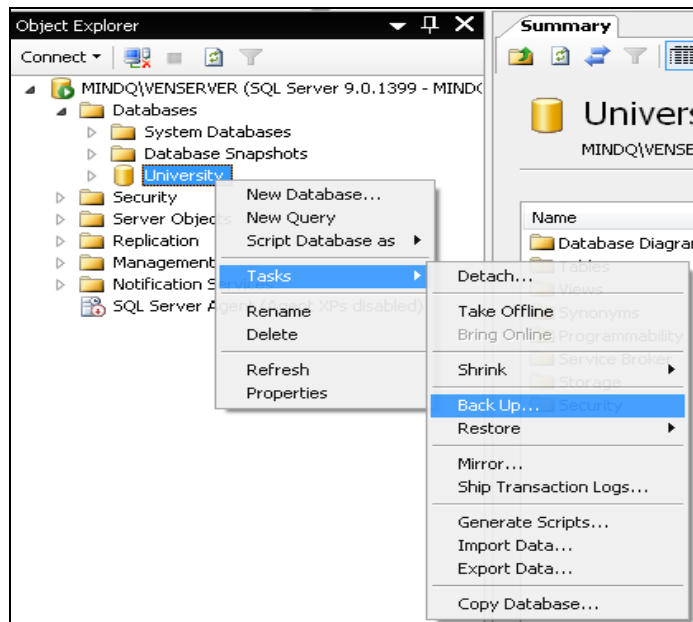
With proper planning, you can recover from many failures, including:

- Media failure.
- User errors.
- Permanent loss of a server.

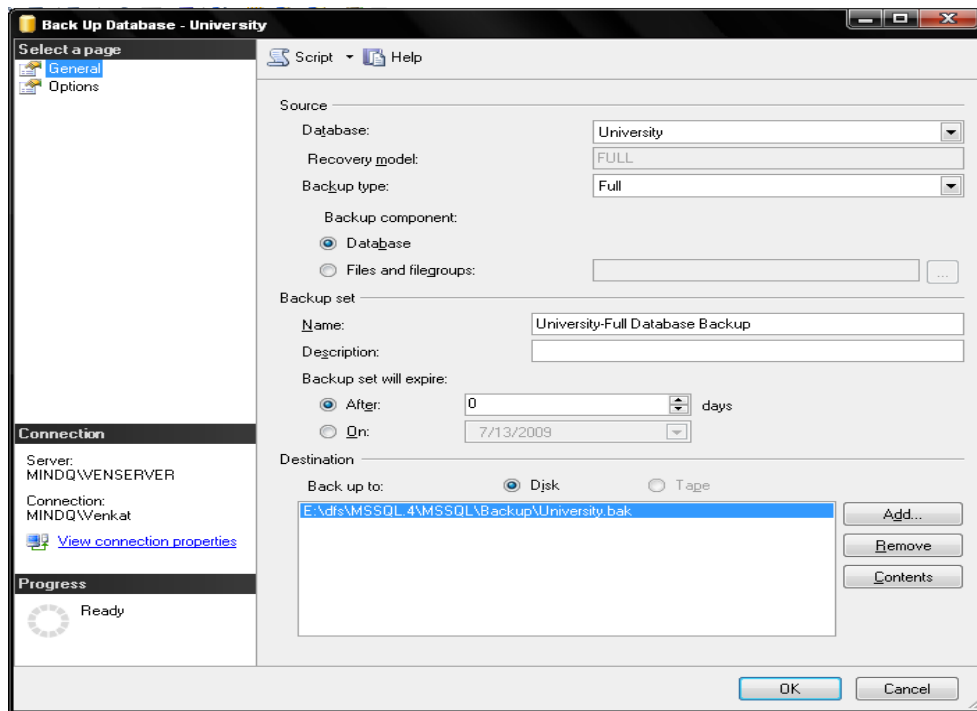
Additionally, backing up and restoring databases is useful for other purposes, such as copying a database from one server to another. By backing up a database from one computer and restoring the database to another, a copy of a database can be made quickly and easily.

### Using the Enterprise Manager to take backup of database:

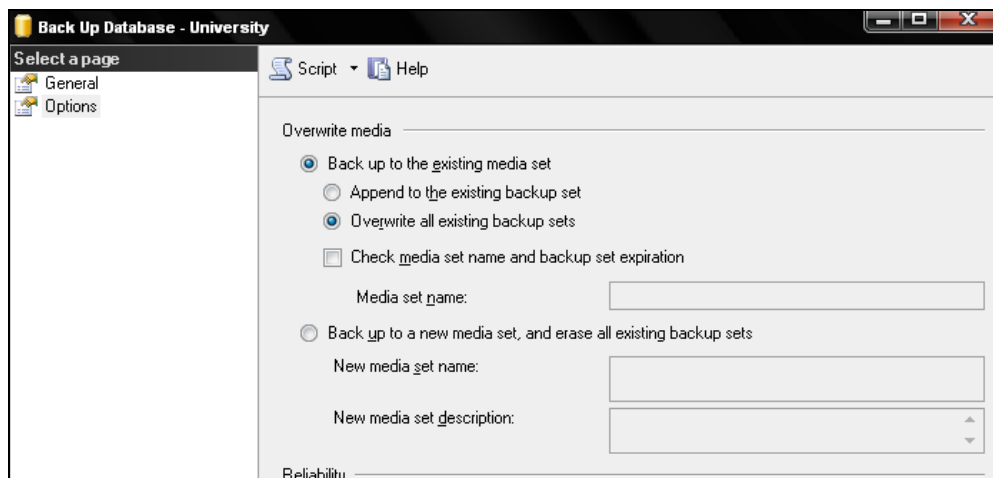
1. Open the Management Studio tool and expand "Databases" item.
2. Right click on the corresponding database and choose "Tasks -> Backup..."



3. In the window displayed click on ADD button and choose the location where the backup to be stored along with name.



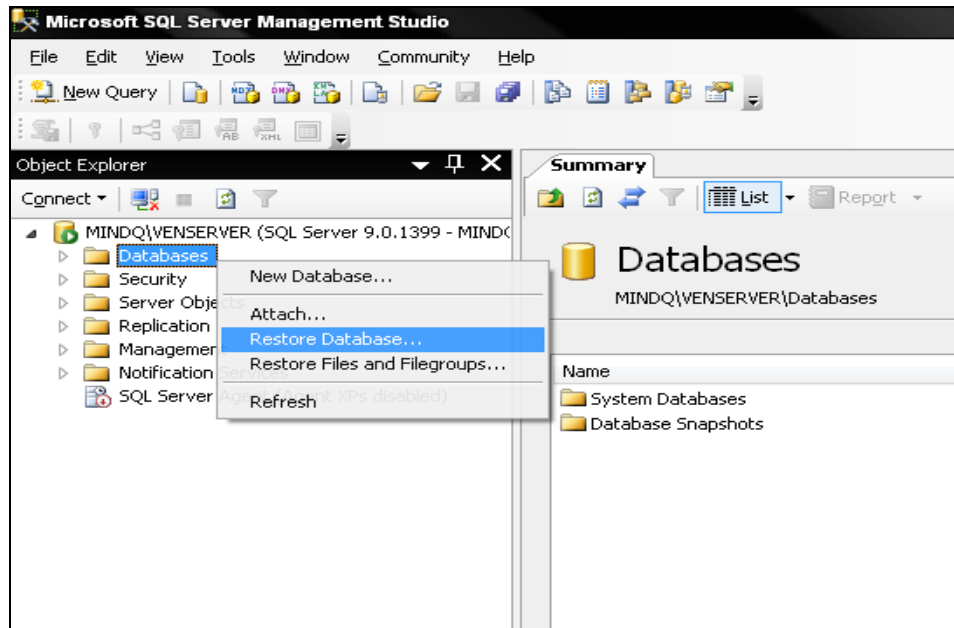
4. Click on "Options" tab and choose "Overwrite existing backup sets" option




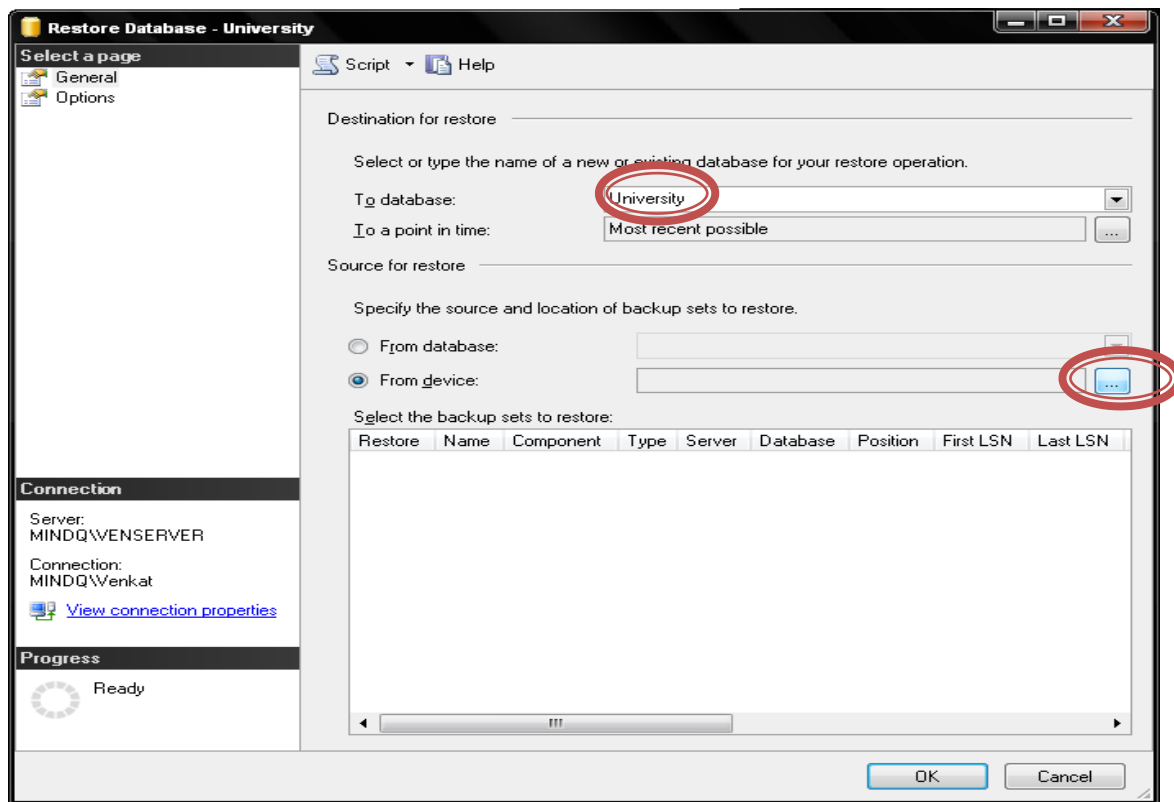
5. Click "OK"

## Recovering Database :

1. Open Management Studio Tool
2. Right click on the "Databases" item and select "Restore Database..." option.



3. In the window displayed enter a name to the database(ex: University) under "To database". Select "From Device" option button and choose  (browse) button



4. Choose the file from the "Locate Backup file" window.
  5. Select "ok"
  6. Make a check mark under "Restore" item for the corresponding backup database file.
  7. Click on "Options" tab and choose "Overwrite Existing Database" option
  8. Click "OK" button.
- Note:** If any error occurred regarding path, change the path under "Restore As" to new path.

## CASE STUDY

Create the following Tables:

LOCATION	
Location_ID	Regional_Group
122	NEW YORK
123	DALLAS
124	CHICAGO
167	BOSTON

DEPARTMENT		
Department_ID	Name	Location_ID
10	ACCOUNTING	122
20	RESEARCH	124
30	SALES	123
40	OPERATIONS	167

JOB	
Job_ID	Function
667	CLERK
668	STAFF
669	ANALYST
670	SALESPERSON
671	MANAGER
672	PRESIDENT



EMPLOYEE									
EMPLOYEE_ID	LAST_NAME	FIRST_NAME	MIDDLE_NAME	JOB_ID	MANAGER_ID	HIREDATE	SALARY	COMM	DEPARTMENT_ID
7369	SMITH	JOHN	Q	667	7902	17-DEC-84	800	NULL	20
7499	ALLEN	KEVIN	J	670	7698	20-FEB-85	1600	300	30
7505	DOYLE	JEAN	K	671	7839	04-APR-85	2850	NULL	30
7506	DENNIS	LYNN	S	671	7839	15-MAY-85	2750	NULL	30
7507	BAKER	LESLIE	D	671	7839	10-JUN-85	2200	NULL	40
7521	WARK	CYNTHIA	D	670	7698	22-FEB-85	1250	500	30

Queries based on the above tables:

### Simple Queries:

1. List all the employee details
2. List all the department details
3. List all job details
4. List all the locations
5. List out first name,last name,salary, commission for all employees
6. List out employee\_id,last name,department id for all employees and rename employee id as "ID of the employee", last name as "Name of the employee", department id as "department ID"
7. List out the employees annual salary with their names only.

### Where Conditions:

8. List the details about "SMITH"
9. List out the employees who are working in department 20
10. List out the employees who are earning salary between 3000 and 4500

11. List out the employees who are working in department 10 or 20
12. Find out the employees who are not working in department 10 or 30
13. List out the employees whose name starts with "S"
14. List out the employees whose name start with "S" and end with "H"
15. List out the employees whose name length is 4 and start with "S"
16. List out the employees who are working in department 10 and draw the salaries more than 3500
17. list out the employees who are not receiving commission.

**Order By Clause:**

18. List out the employee id, last name in ascending order based on the employee id.
19. List out the employee id, name in descending order based on salary column
20. list out the employee details according to their last\_name in ascending order and salaries in descending order
21. list out the employee details according to their last\_name in ascending order and then on department\_id in descending order.

**Group By & Having Clause:**

22. How many employees who are working in different departments wise in the organization
23. List out the department wise maximum salary, minimum salary, average salary of the employees
24. List out the job wise maximum salary, minimum salary, average salaries of the employees.
25. List out the no.of employees joined in every month in ascending order.
26. List out the no.of employees for each month and year, in the ascending order based on the year, month.
27. List out the department id having atleast four employees.
28. How many employees in January month.
29. How many employees who are joined in January or September month.
30. How many employees who are joined in 1985.

31. How many employees joined each month in 1985.
32. How many employees who are joined in March 1985.
33. Which is the department id, having greater than or equal to 3 employees joined in April 1985.

### **Sub-Queries**

34. Display the employee who got the maximum salary.
35. Display the employees who are working in Sales department
36. Display the employees who are working as "Clerk".
37. Display the employees who are working in "New York"
38. Find out no.of employees working in "Sales" department.
39. Update the employees salaries, who are working as Clerk on the basis of 10%.
40. Delete the employees who are working in accounting department.
41. Display the second highest salary drawing employee details.
42. Display the Nth highest salary drawing employee details

### **Sub-Query operators: (ALL,ANY,SOME,EXISTS)**

43. List out the employees who earn more than every employee in department 30.
44. List out the employees who earn more than the lowest salary in department 30.
45. Find out whose department has not employees.
46. Find out which department does not have any employees.

### **Co-Related Sub Queries:**

47. Find out the employees who earn greater than the average salary for their department.

## **Joins**

### **Simple join**

- 48. List our employees with their department names
- 49. Display employees with their designations (jobs)
  - 50. Display the employees with their department name and regional groups.
- 51. How many employees who are working in different departments and display with department name.
- 52. How many employees who are working in sales department.
- 53. Which is the department having greater than or equal to 5 employees and display the department names in ascending order.
- 54. How many jobs in the organization with designations.
- 55. How many employees working in "New York".

### **Non – Equi Join:**

- 56. Display employee details with salary grades.
- 57. List out the no. of employees on grade wise.
- 58. Display the employ salary grades and no. of employees between 2000 to 5000 range of salary.

### **Self Join:**

- 59. Display the employee details with their manager names.
- 60. Display the employee details who earn more than their managers salaries.
- 61. Show the no. of employees working under every manager.

### **Outer Join:**

- 61. Display employee details with all departments.
- 62. Display all employees in sales or operation departments.

### **Set Operators:**

63. List out the distinct jobs in Sales and Accounting Departments.
64. List out the ALL jobs in Sales and Accounting Departments.
65. List out the common jobs in Research and Accounting Departments in ascending order.

### **Solutions**

1. Select \* from employee;
2. Select \* from department;
3. Select \* from job;
4. Select \* from loc;
5. Select first\_name, last\_name, salary, commission from employee;
6. Select employee\_id "id of the employee", last\_name "name", department\_id as "department id" from employee;
7. Select last\_name, salary\*12 "annual salary" from employee
8. Select \* from employee where last\_name='SMITH';
9. Select \* from employee where department\_id=20
10. Select \* from employee where salary between 3000 and 4500
11. Select \* from employee where department\_id in (20,30)
12. Select last\_name, salary, commission, department\_id from employee where department\_id not in (10,30)
13. Select \* from employee where last\_name like 'S%'
14. Select \* from employee where last\_name like 'S%H'
15. Select \* from employee where last\_name like 'S\_\_\_\_'
16. Select \* from employee where department\_id=10 and salary>3500
17. Select \* from employee where commission is Null
18. Select employee\_id, last\_name from employee order by employee\_id

19. Select employee\_id, last\_name, salary from employee order by salary desc
20. Select employee\_id, last\_name, salary from employee order by last\_name, salary desc
21. Select employee\_id, last\_name, salary from employee order by last\_name, department\_id desc
22. Select department\_id, count(\*), from employee group by department\_id
23. Select department\_id, count(\*), max(salary), min(salary), avg(salary) from employee group by department\_id
24. Select job\_id, count(\*), max(salary), min(salary), avg(salary) from employee group by job\_id
25. select datepart(mm,hire\_date) month,count(\*) from employee group by datepart(mm,hire\_date) order by month
26. select datepart(yy,hire\_date) year,datepart(mm,hire\_date) month,count(\*) from employee group by datepart(yy,hire\_date),datepart(mm,hire\_date) order by year
27. Select department\_id, count(\*) from employee group by department\_id having count(\*)>=4
28. select datename(mm,hire\_date) month,count(\*) NoOfEmployees from employee group by datename(mm,hire\_date) having datename(mm,hire\_date)='January'
29. select datename(mm,hire\_date) month,count(\*) NoOfEmployees from employee group by datename(mm,hire\_date) having datename(mm,hire\_date) in ('January','September')
30. select datename(yy,hire\_date) year,count(\*) NoOfEmployees from employee group by datename(yy,hire\_date) having datename(yy,hire\_date)=1985
31. Select datepart(yy,hire\_date)Year, datepart(mm,hire\_date) Month, count(\*) [No. of employees] from employee where datepart(yy,hire\_date)=1985 group by datepart(yy,hire\_date),datepart(mm,hire\_date)
32. Select datepart(yy,hire\_date) Year, datename(mm,hire\_date) Month, count(\*) [No. of employees] from employee where datepart(yy,hire\_date)=1993 and datename(mm,hire\_date)='March' group by datepart(yy,hire\_date),datename(mm,hire\_date)

33. Select department\_id, count(\*) [No. of employees] from employee  
where datepart(yy,hire\_date)=1985 and datename(mm,hire\_date)='April'  
group by datepart(yy,hire\_date), datename(mm,hire\_date),  
department\_id having count(\*)>=3
34. Select \* from employee where salary=(select max(salary) from  
employee)
35. Select \* from employee where department\_id IN (select department\_id  
from department where name='SALES')
36. Select \* from employee where job\_id in (select job\_id from job where  
function='CLERK')
37. Select \* from employee where department\_id=(select department\_id  
from department where location\_id=(select location\_id from location  
where regional\_group='New York'))
38. Select \* from employee where department\_id=(select department\_id  
from department where name='SALES' group by department\_id)
39. Update employee set salary=salary\*10/100 where job\_id=(select job\_id  
from job where function='CLERK')
40. delete from employee where department\_id=(select department\_id from  
department where name='ACCOUNTING')
41. Select \* from employee where salary=(select max(salary) from employee  
where salary <(select max(salary) from employee))
42. Select distinct e.salary from employee where & no-1=(select  
count(distinct salary) from employee where sal>e.salary)
43. Select \* from employee where salary > all (Select salary from employee  
where department\_id=30)
44. Select \* from employee where salary > any (Select salary from employee  
where department\_id=30)
45. Select employee\_id, last\_name, department\_id from employee e where  
not exists (select department\_id from department d where  
d.department\_id=e.department\_id)
46. Select name from department d where not exists (select last\_name from  
employee e where d.department\_id=e.department\_id)
47. Select employee\_id, last\_name, salary, department\_id from employee e  
where salary > (select avg(salary) from employee where  
department\_id=e.department\_id)
48. Select employee\_id, last\_name, name from employee e, department d  
where e.department\_id=d.department\_id

49. Select employee\_id, last\_name, function from employee e, job j where e.job\_id=j.job\_id
50. Select employee\_id, last\_name, name, regional\_group from employee e, department d, location l where e.department\_id=d.department\_id and d.location\_id=l.location\_id
51. Select name, count(\*) from employee e, department d where d.department\_id=e.department\_id group by name
52. Select name, count(\*) from employee e, department d where d.department\_id=e.department\_id group by name having name='SALES'
53. Select name, count(\*) from employee e, department d where d.department\_id=e.department\_id group by name having count (\*)>=5 order by name
54. Select function, count(\*) from employee e, job j where j.job\_id=e.job\_id group by function
55. Select regional\_group, count(\*) from employee e, department d, location l where e.department\_id=d.department\_id and d.location\_id=l.location\_id and regional\_group='NEW YORK' group by regional\_group
56. Select employee\_id, last\_name, grade\_id from employee e, salary\_grade s where salary between lower\_bound and upper\_bound order by last\_name
57. Select grade\_id, count(\*) from employee e, salary\_grade s where salary between lower\_bound and upper\_bound group by grade\_id order by grade\_id desc
58. Select grade\_id, count(\*) from employee e, salary\_grade s where salary between lower\_bound and upper\_bound and lower\_bound>=2000 and lower\_bound<=5000 group by grade\_id order by grade\_id desc
59. Select e.last\_name emp\_name, m.last\_name, mgr\_name from employee e, employee m where e.manager\_id=m.employee\_id
60. Select e.last\_name emp\_name, e.salary emp\_salary, m.last\_name, mgr\_name, m.salary mgr\_salary from employee e, employee m where e.manager\_id=m.employee\_id and m.salary<e.salary
61. Select m.manager\_id, count(\*) from employee e, employee m where e.employee\_id=m.manager\_id group by m.manager\_id
62. Select last\_name, d.department\_id, d.name from employee e, department d where e.department\_id\*=d.department\_id



63. Select last\_name, d.department\_id, d.name from employee e, department d where e.department\_id=d.department\_id and d.department\_id in (select department\_id from department where name IN ('SALES','OPERATIONS'))
64. Select function from job where job\_id in (Select job\_id from employee where department\_id=(select department\_id from department where name='SALES')) union Select function from job where job\_id in (Select job\_id from employee where department\_id=(select department\_id from department where name='ACCOUNTING'))
65. Select function from job where job\_id in (Select job\_id from employee where department\_id=(select department\_id from department where name='SALES')) union all Select function from job where job\_id in (Select job\_id from employee where department\_id=(select department\_id from department where name='ACCOUNTING'))
66. Select function from job where job\_id in (Select job\_id from employee where department\_id=(select department\_id from department where name='RESEARCH')) intersect Select function from job where job\_id in (Select job\_id from employee where department\_id=(select department\_id from department where name='ACCOUNTING')) order by function

## More Practical Examples

1. Assuming system date as current date, solve the following
  - i) Get the last day of week
  - ii) Get the first day of week
  - iii) Get the last day of the month

Note: The solution you provided should work for any given date.

2. In the Northwind database, get the customerid, companyname and no. of orders made by each customer.
3. A table is having the following information-

id	name	Age
1	A	25
2	B	26
3	C	24
2	B	26

- Write a query to delete duplicate records.(only one record with sid 2 should exist).
4. In my database I have a table with a column name 'MyColumn1'. Write a query to get the names of the tables that contains this column.

A department store has many sections such as Toys, Cosmetics, Clothing, Household Items, and Electronics etc. Each section has many employees. Employees can belong to only one section. In addition, each section also has a head that is responsible for the section's performance.

The department store also has many customers who purchase goods from various sections. Customers can be of two types Regular and Ad-hoc. Regular customers get credit at the department store. Maximum credit limit allowed is Rs.10000.

The store procures goods from various suppliers. The goods are stored in a warehouse and transferred to the store as and when requirement comes up. Quantity of goods supplied cannot be less than 0 and cannot be greater than 10000 for a particular supply.

The store has a computerized system for all its operations. Given below are the tables in the database:

- **Section** ( SectionNo int, Description Varchar(30), Section\_Head\_EmpNo int)
- **Employees**( EmpNo int, First\_Name Varchar(30), Last\_Name Varchar(30), Address Varchar(30), Grade int, Salary Numeric(9,2), SectionNo int, Date\_Joined Datetime)
- **Customers**( CustNo int, First\_Name Varchar(30), Last\_Name Varchar(30), CustType char(1) (with check constraint R/A), Credit\_Limit Numeric(9,2), Credit\_Card\_No bigint, Credit\_Card\_Type Varchar(8))
- **Suppliers** (SuppNo int, Name Varchar(30), Address Varchar(30), City Varchar(30))
- **Products** (ProductNo int identity, Product\_Category\_Code Char(3), Description Varchar(30), Price Numeric(9,2), Qty\_On\_Hand int)
- **Product\_CategoryMaster**(Product\_Category\_Code Char(3), Description Varchar(15))
- **Suppliers\_Products** ( SuppNo int, ProductNo int, Date\_Supplied Datetime, Qty\_Supplied int)
- **Customers\_Products**( TransactionID int, CustNo int, ProductNo int, Date\_of\_Purchase Datetime, Qty\_Purchased int)

The column names which have been underlined are the primary keys for the tables.

Create the tables with all appropriate constraints. Use the constraints UNIQUE, NOT NULL, CHECK, PRIMARY KEY, FOREIGN KEY etc. wherever necessary

5. Find all employees whose names begin with 'A' and end with 'A'.
6. Find the total salary paid by each section to employees.
7. Display the section name and the name of the person who heads the section.
8. Display supplier names and cities. If the city is null, display 'LOCAL'.
9. Display the customer names and the customer type. If the customer type is 'R' displays 'Regular', if the customer type is 'A' display 'Ad-hoc'.
10. Try creating a view which when used, will display the supplier names, product names, the quantity supplied and the date supplied for all records in Supp\_Products. The listing should be in alphabetical order of supplier names.
11. List all suppliers who have not supplied any products. Use an outer join to do this.
12. Display the average salary of employees in each section.
13. Display the customer no and the total number of products purchased by him, for those customers who have purchased at least 2 or more different products. Sort the listing in ascending order of customer number.

14. Display the customer name and product names which have been purchased by him. The customer and product names should be in upper case.
15. Display the products that have been supplied more than a month ago.
16. Display employee names and the date joined for employees. Date should be in the format 'DD-MMM-YYYY'.
17. Display employee names and the number of months between today's date and the joining date.
18. Display product names and the price rounded to the nearest integer.
19. Find the product which has the greatest price in each category.

#### T-SQL programming

20. Write a Procedure code which will let you insert a record into the Section table. Use variables to represent the values of section\_no,description and section\_head\_empno. What happens if you try to insert a value of section\_no which already exists in the section table? Write proper error handler.
21. Write a stored procedure to get the total quantity supplied for a particular product. The procedure should total the quantity, and the calling block should print the product no, product description and the total quantity supplied. The procedure should raise an exception if the total quantity supplied is 0, or the product does not exist.
22. Write a procedure to insert a new record into the Suppliers\_Products table, and also update the corresponding Qty\_On\_Hand for the product using a trigger.

Assume an Employee and department tables with following description

Create table employee (empNo int,empName varchar(20),  
empSalary numeric(8,2),grade char(1),location varchar(20))

create table department(deptno int,location varchar(20))

23. Write a procedure to display the following information to the given employee number: Employee name (first\_name + last\_name) Employee section no Employee Designation Obtain the designation as follows:

Grade	Designation
1	Vice President
2	Senior Manager
3	Assistant Manager
4	Section Supervisor
5	Sales Assistant

24. Whenever the location of the department changes, update the employee location also. (Use an AFTER UPDATE trigger for this).
25. Add a new column called Performance\_Measure Number(3,1) to the EMP table. The column can take values between 1 and 10(Use a check constraint to implement this). Whenever the performance measure of an employee is 8 or more, insert a record into a table called EXCEPTIONAL\_EMPLOYEES, which contains the columns EMPNO and PERFORMANCE\_LINKED\_BONUS. Write a [AFTER INSERT OR AFTER UPDATE] trigger for this.
26. Write a Trigger that gets fired whenever employee salaries are updated for a particular grade. The trigger inserts a record into the table SALARY\_UPDATE\_LOG that contains the USERNAME, DATE OF UPDATION of the user who updated the salary.
27. Using northwind database, create a view that gets the information about orderid, orderdate, productid, proudctname , quantityorderd. Enter a new order using the view.(used instead of insert trigger)
28. Write a user-defined function that gets the details of orders made by a given customerid. Call this function from a t-SQL program.(function should return a table).

### Solutions:

1.
  - i) DECLARE @Date datetime  
     select @Date = GETDATE()  
     SELECT DATEADD(dd,-(DATEPART(dw, @Date) - 1),@Date) AS 'First day of the week'
  - ii) SELECT DATEADD(dd,-(DATEPART(dw, @Date) - 7),@Date) AS 'Last day of the week'
  - iii) SELECT DAY(DATEADD(d, -DAY(DATEADD(m,1,@Date))),  
     DATEADD(m,1,@Date))) AS 'Last day of the month'

2.      `select c.customerid,companyname,count(*) nooforderes from customers c  
inner join orders o  
on c.customerid=o.customerid  
group by c.customerid,companyname`

3.

```
create table t1  
(id int,name char(10),age int)
```

```
insert into t1  
select 1,'A',25 union all  
select 2,'B',26 union all  
select 3,'C',24 union all  
select 2,'B',26
```

First, run the above GROUP BY query to determine how many sets of duplicate PK values exist, and the count of duplicates for each set.

Select the duplicate key values into a holding table. For example:

```
SELECT id,name,count=count(*)  
INTO holdkey  
FROM t1  
GROUP BY id,name  
HAVING count(*) > 1
```

Select the duplicate rows into a holding table, eliminating duplicates in the process. For example:

```
SELECT DISTINCT t1.*  
INTO holddups  
FROM t1, holdkey  
WHERE t1.id = holdkey.id  
AND t1.name = holdkey.name
```

At this point, the holddups table should have unique PKs, however, this will not be the case if t1 had duplicate PKs, yet unique rows (as in the SSN example above). Verify that each key in holddups is unique, and that you do not have duplicate keys, yet unique rows. If so, you must stop here and reconcile which of the rows you wish to keep for a given duplicate key value. For example, the query:

```
SELECT id, name, count(*)
FROM holddups
GROUP BY id, name
```

should return a count of 1 for each row. If yes, proceed to step 5 below. If no, you have duplicate keys, yet unique rows, and need to decide which rows to save. This will usually entail either discarding a row, or creating a new unique key value for this row. Take one of these two steps for each such duplicate PK in the holddups table.

Delete the duplicate rows from the original table. For example:

```
DELETE t1
FROM t1, holdkey
WHERE t1.id = holdkey.id
```

Put the unique rows back in the original table. For example:

```
INSERT t1 SELECT * FROM holddups
```

4. select o.name from syscolumns c,sysobjects o where c.id=o.id and c.name='MyColumn1'

(or)

select name from sysobjects where ID IN(select ID from syscolumns where name='MyColumn1')

5. select empno,First\_Name,Last\_name from employees where first\_name like 'A%A'

6. select sectionno,SUM(salary) TotalSal from employees group by sectionno

7. select sec.sectionno,description,e.empno,First\_name from section sec  
inner join employees e  
on sec.section\_head\_empno=e.empno

8.     select suppno,name,isnull(city,'Local') from suppliers
  
9.     select Custno,First\_name,Last\_name,  
       case CustType  
         when 'R' then 'Regular'  
         when 'A' then 'Ad-hoc'  
       end from customers
  
10.    create view vSupOrders  
       as  
       select Sup.suppno,Sup.name,Sp.productno,p.description,qty\_supplied,  
       date\_supplied from Suppliers sup inner join Suppliers\_Products sp  
       on sup.SuppNo=sp.SuppNo  
       inner join Products p on sp.ProductNo=p.ProductNo  
       order by sup.Name  
  
       select \* from vsuporders
  
11.    select   sup.suppno,Name   from   suppliers   sup   left   outer   join  
       suppliers\_products sp  
       on sup.suppno=sp.suppno where sp.productno is null
  
12.    select sectionno,avg(salary) from employees group by sectionno
  
13.    select    custno,productno,COUNT(productno)       NoOfOrders       from  
       customers\_products group by custno,productno order by custno
  
14.    select upper(first\_name),upper(description) from customers cu inner join  
       customers\_products cp on cu.custno=cp.custno  
       inner join products p on cp.productno=p.productno
  
15.    select   productno,date\_supplied   from   suppliers\_products   where  
       date\_supplied<DATEADD(mm,-1,getdate())
  
16.    select first\_name,last\_name,convert(char,date\_joined,105) from  
       employees
  
17.    select first\_name,last\_name,DATEDIFF(mm,date\_joined,getdate()) from  
       employees
  
18.    select productno,description,ceiling(price) from products



```

19.  select p.productno,p.product_category_code,p.price
      from products p inner join (select product_category_code,max(price)
      maxPrice from products group by product_category_code) a
      on p.product_category_code=a.product_category_code and
      p.Price=maxPrice order by p.Product_Category_Code

20.  create procedure pInsertSection
      (@sno int,@desc varchar(30),@shempno int)
      as
      begin
          begin try
              insert into Section values(@sno,@desc,@shempno)
          end try
          begin catch
              if ERROR_NUMBER()=2627
                  print 'Duplicate SectionNo specified. Violated Primary key"
              else
                  print Convert(varchar(30),Error_Number()) + ' ' +
                      ERROR_MESSAGE()
          end catch
      end

exec pinsertsection 3,'Sports',2003

21.
alter procedure pGetProductQtySupplied(@pid int)
as
begin
    declare @count int
    select @count= COUNT(*) from Suppliers_Products where ProductNo=@pid
    if @count=0
        raiserror('Productno doenot exist',10,1)
    else
        select  p.productno,p.description,sum(qty_supplied)  from  products
p,suppliers_products su
        where p.productno=su.productno and su.productno=@pid group by
p.productno,p.description

end
exec pGetProductQtySupplied 10

```

22.

```
create procedure pInsertSupplierProducts(@sno int,@pid int,@dtSupp
datetime,@qtySupp int)
as
begin
    insert into Suppliers_Products values(@sno,@pid,@dtSupp,@qtySupp)
end
```

```
create trigger tUpdateProdcutQty
on Suppliers_products
for insert
as
begin
    declare @qSupp int,@pid int
    select @qSupp=qty_supplied from inserted
    select @pid=productno from inserted
    update products set qty_on_hand=Qty_on_hand+@qsupp where
productno=@pid
end
```

```
exec pinsertsupplierproducts 102,1,'3/13/10',100
```

Assume an Employee and department tables with following description

Create table emp (empNo int,empName varchar(20),  
empSalary numeric(8,2),grade char(1),location varchar(20))

create table department(deptno int,location varchar(20))

23.

```
select first_name + ' ' + last_name,sectionno,
    case Grade
        when 1 then 'Vice President'
        when 2 then 'Senior Manager'
        when 3 then 'Assistant Manager'
        when 4 then 'Section Supervisor'
        when 5 then 'Sales Assistant'
    end Designation from employees
```

```

24.
create trigger trgUpdateLocation
on department
for update
as
begin
    declare @oldloc varchar(20),@newloc varchar(20)
    select @newloc=location from inserted
    select @oldloc=location from deleted
    update emp set location=@newloc where location=@oldloc
end

testing:
select * from emp
select * from department
update department set location='Sec' where location='Secunderabad'

25.  alter table emp add Performance_measure numeric(3,1) constraint
ck_emp_pm check(performance_measure between 1 and 10)

create table EXCEPTIONAL_EMPLOYEES
(EMPNO int,PERFORMANCE_LINKED_BONUS numeric(7,2))

create trigger trgPeromance
on emp
for insert,update
as
begin
    declare @eid int,@pm numeric(3,1)
    select @eid=empno from inserted
    select @pm=performance_measure from emp where empno=@eid
    if @pm>=8
        insert into exceptional_employees values(@eid,10000)
end

select * from emp
select * from EXCEPTIONAL_EMPLOYEES

update emp set performance_measure=9 where empno=100

```

```

26.  create table SALARY_UPDATE_LOG
      (username varchar(20),DateofUpdate datetime)

create trigger trgUpdateSalary
on emp
for update
as
begin
    insert into SALARY_UPDATE_LOG values(System_user,getdate())
end

testing:
select * from emp
select * from SALARY_UPDATE_LOG
update emp set empsalary=empsalary+50000 where empNo=101

27.
CREATE VIEW Customerorders_vw
with schemabinding
as
select o.orderid,o.orderdate,oi.productid,oi.quantity,oi.unitprice
from dbo.orders o inner join dbo.[order details] oi
on o.orderid=oi.orderid
inner join dbo.products p on oi.productid=p.productid

select * from customerorders_vw

insert into customerorders_Vw values(1,GETDATE(),2,10,150)

create trigger trgCustOrdInsert on customerorders_vw
instead of insert
as
begin
    if(select count(*) from inserted)>0
        insert into dbo.[order details](orderid,productid,unitprice,quantity)
        select i.orderid,i.productid,i.unitprice,i.quantity
        from inserted i join orders o
        on i.orderid=o.orderid

        if @@rowcount=0
            raiserror('no matching orders. cannot perform insert',10,1)
end
insert into customerorders_Vw values(1,GETDATE(),2,10,150)

```

28.

create function fnGetOrders(@cid NCHAR(5))

returns table

as

```
return(select Cu.Customerid,o.orderid,o.orderdate,od.productid,  
        p.productname, od.unitprice,od.quantity  
        from customers cu inner join orders o on cu.customerid=o.customerid  
        inner join [order details] od on o.orderid=od.orderid  
        inner join products p on od.productid=p.productid  
        where cu.customerid=@cid)
```

Testing:

select \* from dbo.fnGetOrders('HANAR')

dbcc checkident('customers',reseed,30)

## **FREQUENTLY ASKED QUESTIONS**

1. What are the new data types in SQL Server?
2. How to get the version of SQL Server?
3. What are system Databases?
4. What happens if we delete any one of either Master or Model databases?
5. What are data files?
6. What is the Maximum number of databases that can be created for an instance of SQL Server?
7. What is identity property?
8. How to insert explicit values into identity column?
9. What is the maximum size of data that can be entered in a single row of a table?
10. How to get the table names in the database?
11. How to insert multiple rows using insert statement?
12. Difference between DELETE and TRUNCATE?
13. Difference between WHERE and HAVING clauses?
14. What are the types of Integrity Constraints?
15. Difference between UNIQUE and PRIMARY KEY?
16. What is composite primary key?
17. How many columns can be given in a composite primary key?
18. What is ON DELETE CASCADE?
19. How many references can be given to a table?
20. What are Joins? Types of Joins?
21. What is a correlated sub query?
22. What is the difference between IN and ANY Operators?
23. What is a View?
24. What is an Index? What are the types of indexes?
25. How many Clustered Indexes can be created on a table?
26. How many NonClustered Indexes can be created on a table?
27. What is Normalization?
28. What are the Normal Forms?
29. What is Denormalization?
30. What is a cursor?
31. What are the types of cursors?
32. What is @@FETCH\_STATUS?
33. What is a stored procedure?
34. How many stored procedures can be nested?
35. How to raise errors inside a stored procedure?
36. What is a trigger?
37. What are Instead of Triggers?
38. What is the difference between a Stored Procedure and a Trigger?
39. What is User Defined Function?

40. What is the difference between Stored Procedure and User Defined Function?
41. What are ACID properties?
42. What is Lock?
43. What is a Deadlock?
44. Write a query to get n<sup>th</sup> maximum salary?
45. How to delete duplicate rows from a table?
46. What is the use of DBCC commands
47. What are the authentications available in sql sever?
48. What are the authentication modes available?
49. How to increase the performance of a query?

## Solutions

1. SQL Server 2000 introduces three new data types. **bigint** is an 8-byte integer type. **sql\_variant** is a type that allows the storage of data values of different data types. **table** is a type that allows applications to store results temporarily for later use. It is supported for variables, and as the return type for user-defined functions.
2. The following 3 commands can be used to get the version of SQL Server
  - Select @@Version
  - EXEC sp\_msgetversion
  - Exec xp\_msver
3. With the installation of SQL Server 4 databases will be created automatically, called as System Databases. They are:
  - MASTER
  - MODEL
  - MSDB
  - TEMPDB

**Master:** The **master** database records all of the system level information for a SQL Server system. It records all login accounts and all system configuration settings. **Master** is the database that records the existence of all other databases, including the location of the database files. **Master** records the initialization information for SQL Server.

**Model:** The **model** database is used as the template for all databases created on a system. When a CREATE DATABASE statement is issued, the first part of the database is created by copying in the contents of the **model** database, then the remainder of the new database is filled with empty pages.

**MSDB:** The **msdb** database is used by SQL Server Agent for scheduling alerts and jobs, and recording operators.

**TempDB:** While performing some complex operations, sometimes, we may need to store the results temporarily somewhere. To do this task we use temporary tables. Whatever database we are using, if we create a temporary table that will be created in tempdb database only. **tempdb** is re-created every time SQL Server is started so the system starts with a clean copy of the database.

4. SQL Server won't work properly if we delete any one of Master or Model databases.
5. Microsoft® SQL Server maps a database over a set of operating-system files. There are the locations where the original database data will be stored. Data and log information are never mixed on the same file, and individual files are used only by one database.

SQL Server databases have three types of files:

- Primary data files

The primary data file is the starting point of the database and points to the other files in the database. Every database has one primary data file. The recommended file name extension for primary data files is .mdf.

- Secondary data files

Secondary data files comprise all of the data files other than the primary data file. Some databases may not have any secondary data files, while others have multiple secondary data files. The recommended file name extension for secondary data files is .ndf.

- Log files

Log files hold all of the log information used to recover the database. There must be at least one log file for each database, although there can be more than one. The recommended file name extension for log files is .ldf.



6. 32,767

7. Identity:

Auto number generated columns can be implemented using the IDENTITY property, which allows the application developer to specify both an identity number for the first row inserted into the table (**Seed** property) and an increment (**Increment** property) to be added to the seed to determine successive identity numbers. When inserting values into a table with an identifier column, Microsoft® SQL Server™ 2000 automatically generates the next identity value by adding the increment to the seed.

8. In general we can't enter values into an identity column directly. To do this we have to follow the two steps

- Set the IDENTITY\_INSERT to ON for that table
- Specify the column list in the INSERT statement  
EX: Suppose there is a table called STD with two columns SID and SNAME. SID is an identity column.
- set IDENTITY\_INSERT std ON
- Insert into std(Sid,sname) values (5,'Anil')

9. 8060 bytes

10. To get the name of all the tables in a database you can use one of the following 3 methods.

- SELECT \* FROM INFORMATION\_SCHEMA.TABLES  
WHERE TABLE\_TYPE='BASE TABLE'
- SELECT NAME FROM SYSOBJECTS WHERE TYPE='U'
- EXEC SP\_TABLES → returns all tables (system, user)
  
- SELECT \* FROM INFORMATION\_SCHEMA.TABLES  
WHERE TABLE\_TYPE='BASE TABLE'
- SELECT NAME FROM SYSOBJECTS WHERE TYPE='U'
- EXEC SP\_TABLES → returns all tables (system, user)

11. sample table: create table tName(firstName varchar(20))  
INSERT INTO TNAME  
SELECT 'RAJ' UNION ALL  
SELECT 'GOPI' UNION ALL  
SELECT 'VIVEK'

12. Both TRUNCATE and DELETE commands are used to delete the data from the table. The difference between them is :
- With TRUNCATE entire table data will be deleted. Whereas with DELETE command we can delete entire table data as well as certain row data, which is not possible with TRUNCATE.
  - Even though, we are deleting entire table data, the deletion will be done row by row. Where as, with TRUNCATE, the deletion will be done at once.
  - DELETE maintains log information about the deleted records. Where as, TRUNCATE does not maintain log details. Instead of this, it maintains deallocation of Data Pages.
13. Both WHERE and HAVING clauses are used to provide a restriction to the result set. Only, those records, which satisfy the criteria, will be returned. The difference between them is:
- With WHERE clause the condition can be provided based on any column of table. Whereas, by using HAVING clause, the condition can be specified either by using an Aggregate Function or a GROUP BY column.
  - Suppose, if the SELECT statement contains WHERE and GROUP BY clauses, the WHERE clause should precede GROUP BY clause. Whereas, if the SELECT statement contains GROUP BY and HAVING clauses, the HAVING clause should come after GROUP BY clause.
14. Integrity Constraints are used to maintain Data Integrity in the database. It is a restriction that can be applied on a column or a table. 3 types of Integrity constraints are available
- DOMAIN INTEGRITY: They can be applied on a column of a table. It ensures that a column should satisfy specified criteria. Ex: CHECK
  - ENTITY INTEGRITY: they can be provided on a column or on a table. It ensures that the column must have unique (non=duplicate) values.  
Ex. UNIQUE, PRIMARY KEY
  - REFERENCIAL INTEGRITY: Referential integrity preserves the defined relationships between tables when records are entered or deleted. In Microsoft® SQL Server™ 2000, referential integrity is based on relationships between foreign keys and primary keys or between foreign keys and unique keys (through FOREIGN KEY and CHECK constraints). Referential integrity ensures that key values are consistent across tables. Such consistency requires that there be no references to nonexistent values and that if a key value changes, all references to it change consistently throughout the database.

When you enforce referential integrity, SQL Server prevents users from:

- Adding records to a related table if there is no associated record in the primary table.
- Deleting records from a primary table if there are matching related records.

15. The differences between UNIQUE and PRIMARY KEY are:

- UNIQUE allows NULL values whereas, a PRIMARY KEY does not allow a NULL value.
- A table can contain more than one UNIQUE constraint whereas, a table can have only one PRIMARY KEY.
- An UNIQUE constraint on a column by default creates a NONCLUSTERED INDEX. Whereas, a PRIMARY KEY on a column by default creates a CLUSTERED INDEX

16. In general a PRIMARY KEY can be given on a column by which we can able to identity different rows of the table. Suppose, it is not possible to identity different rows of the table based on a single column which can be done with combination of columns, then we will make that combination as the primary key which is called as COMPOSITE PRIMARY KEY.

17. 16 columns.

**18.** In general if the parent table record is having a reference In the child table, we can't delete the record directly. With the use of "**ON DELETE CASCADE**", when we delete the parent table record, that record and all its child references will be deleted at once.

19. A table can have as many as **253** references.

20. A **JOIN** is a select statement used to get the data distributed among different tables.

**Types of Joins:**

1. **Inner Join:** Returns only the matching records between the two tables based on the criteria.
2. **Outer Join:** Returns left alone records along with matching records.
  - **Left Outer Join:** Returns all the matching records between the two tables and all the left alone records from left table.

- **Right Outer Join:** Returns all the matching records between the two tables and all the left alone records from Right table.
  - **Full Outer Join:** Returns all the matching records between the two tables and all the left alone data from left table as well as all the left alone data from the right table.
- 3. Cross Join:** A cross join that does not have a WHERE clause produces the Cartesian product of the tables involved in the join. The size of a Cartesian product result set is the number of rows in the first table multiplied by the number of rows in the second table. This is an example of a Transact-SQL cross join. Cross joins return all rows from the left table, each row from the left table is combined with all rows from the right table.
- 4. Self Join:** Joining a table to itself is called as self Join. In a self join we have to use two copies of the same table to join them. To distinguish the two copies, we use duplicate names to the tables called as *'table aliases'*.

**21. Correlated Sub Query:** In a normal sub query, at first the inner query will be executed and based on the result the parent query will be executed. Both queries will execute only once.

Whereas, in a Correlated Sub Query, the inner query will execute for each and every row of the parent query. The internal process in the execution of a correlated sub query is as follows.

- The outer query passes the value of the first record into the inner query.
- The inner query then executes based on the values passed in. and it passes the resultant values back to the parent query.
- The parent query uses those values to finish the processing.

The above 3 steps should be performed for each row of the parent query.

**22. IN operator** can be used to provide the list of values to checked, either by using explicit list of values and also with the resultant values of a SELECT statement, which is not possible with ANY operator, where we can use only the SELECT statement to specify the list of values.

**23.** A view can be thought of as either a virtual table or a stored query. The data accessible through a view is not stored in the database as a distinct object. What is stored in the database is a SELECT statement. The result set of the SELECT statement forms the virtual table returned by the view. A user can use this virtual table by referencing the view name in Transact-SQL statements the same way a table is referenced. A view is used to do any or all of these functions:

- Restrict a user to specific rows in a table.

For example, allow an employee to see only the rows recording his or her work in a labor-tracking table.

- Restrict a user to specific columns.

For example, allow employees who do not work in payroll to see the name, office, work phone, and department columns in an employee table, but do not allow them to see any columns with salary information or personal information.

- Join columns from multiple tables so that they look like a single table.
- Aggregate information instead of supplying details.

For example, present the sum of a column, or the maximum or minimum value from a column.

Views are created by defining the SELECT statement that retrieves the data to be presented by the view. The data tables referenced by the SELECT statement are known as the base tables for the view

24. **Indexes:** In a database, an index allows the database program to find data in a table without scanning the entire table. An index in a database is a list of values in a table with the storage locations of rows in the table that contain each value. Indexes can be created on either a single column or a combination of columns in a table and are implemented in the form of B-trees.

SQL Server provides two types of indexes.

- Clustered
- NonClustered

**Clustered:** In this case, the index structure will occupy the space in the same data page where the original data is stored. The index structure order is similar to the physical order of data in the data page.

**NonClustered:** This is the default index created by SQL Server. In this case, the index structure will occupy space at one place and the data page at other place. The order of index structure is not similar to the physical order of data in the data page.

25. **1**

26. **249**

27. Normalization is the process of breaking up data into a logical non-repetitive format that can easily be reassembled as a whole.

**28. Normal Forms:**

Basically there are 6 Normal Forms. A fully normalized database is the one that is normalized to the Third Normal Form.

**1NF:** A table is said to be in 1NF if it satisfies the following rules.

- The table must not contain any redundant groups of data
- The table must contain atomic data (at the intersection of row and column only one value should be specified)

**2NF:** A table is said to be in 2NF if it satisfies the following rules.

- The table must be in 1NF
- All the non-key column must depend on Whole key (primary key)

**3NF:** A table is said to be in 3NF if it satisfies the following rules.

- The table must be in 2NF
- All the non-key columns must depend only on the whole key. They should not depend on other non-key columns. (There should be no TRANSITIVE DEPENDENCY)
- The data should not be derived further.

**29. De-Normalization:** It is the reverse process of Normalization. Sometimes, by including just one de-normalized column in a table, you can eliminate or significantly cut down the number of joins necessary to retrieve information.

30. A cursor is a pointer to the result of a SELECT statement.

- Allowing positioning at specific rows of the result set.
- Retrieving one row or block of rows from the current position in the result set.
- Supporting data modifications to the rows at the current position in the result set.
- Supporting different levels of visibility to changes made by other users to the database data that is presented in the result set.

- Providing Transact-SQL statements in scripts, stored procedures, and triggers access to the data in a result set.

### 31.Types of Cursors:

- Static cursors
- Dynamic cursors
- Forward-only cursors
- Keyset-driven cursors

Static cursors detect few or no changes but consume relatively few resources while scrolling, although they store the entire cursor in **tempdb**. Dynamic cursors detect all changes but consume more resources while scrolling, although they make the lightest use of **tempdb**. Keyset-driven cursors lie in between, detecting most changes but at less expense than dynamic cursors.

### 32.@@FETCH\_STATUS:

Returns the status of the last cursor FETCH statement issued against any cursor currently opened by the connection.

Return value	Description
0	FETCH statement was successful.
-1	FETCH statement failed or the row was beyond the result set.
-2	Row fetched is missing.

### 33. Stored Procedure:

A Stored Procedure is a group of Transact-SQL statements compiled into a single execution plan.

Stored procedures assist in achieving a consistent implementation of logic across applications. The SQL statements and logic needed to perform a commonly performed task can be designed, coded, and tested once in a stored procedure. Each application needing to perform that task can then simply execute the stored procedure. Coding business logic into a single stored procedure also offers a single point of control for ensuring that business rules are correctly enforced.

## Parameters

Parameters are used to exchange data between stored procedures and the application or tool that called the stored procedure:

- Input parameters allow the caller to pass a data value to the stored procedure.
- Output parameters allow the stored procedure to pass a data value or a cursor variable back to the caller.
- Every stored procedure returns an integer return code to the caller. If the stored procedure does not explicitly set a value for the return code, the return code is 0.

34. **32**

35. Using RAISERROR.

It returns a user-defined error message and sets a system flag to record that an error has occurred. Using RAISERROR, the client can either retrieve an entry from the **sysmessages** table or build a message dynamically with user-specified severity and state information. After the message is defined it is sent back to the client as a server error message.

36. A trigger is a complex integrity constraint that can be applied on a table. Trigger will be fired automatically when the user performs any insertion, deletion or updation operations on the table.

37. Instead of Triggers are used to do updations on views, which is not possible previously. Three types of Instead of Triggers are available. *Instead of Insert, Instead of Delete and Instead of Update*. For a table we can have only one of each category.

38. Both Stored Procedures and Triggers contain pre-compiled set of executable statements. The difference between them is:

- A procedure contains parameters, where as a trigger does not allow any parameters.
- Procedures should be called explicitly from another program where as a Trigger will be fired automatically when the user performs any insertion, deletion or updation operations on the table.

39. An UDF is also a database object contains pre-compiled set of executable statements as a unit. They can return any type of value.

40. The difference between an UDF and Stored Procedure is:



- Stored procedure allows *input* and *output* parameters. Whereas, an UDF allows only *input* parameters. Instead of output parameters they support RETURN values.
- Stored Procedures also provide RETURN keyword but it specifies only the success or failure by returning an integer value. Whereas, an UDF's RETURN value can return any type of data.
- A Stored Procedure cannot be used inside a SELECT statement. Whereas, an UDF can be used as part of the SELECT statement.

41.

A transaction is a logical unit of work in which, all the steps must be performed or none. ACID stands for Atomicity, Consistency, Isolation, Durability. These are the properties of a transaction. For more information and explanation of these properties, see SQL Server books online or any RDBMS fundamentals text book.

### **Atomicity**

A transaction must be an atomic unit of work; either all of its data modifications are performed, or none of them is performed.

### **Consistency**

When completed, a transaction must leave all data in a consistent state. In a relational database, all rules must be applied to the transaction's modifications to maintain all data integrity. All internal data structures, such as B-tree indexes or doubly-linked lists, must be correct at the end of the transaction.

### **Isolation**

Modifications made by concurrent transactions must be isolated from the modifications made by any other concurrent transactions. A transaction either sees data in the state it was in before another concurrent transaction modified it, or it sees the data after the second transaction has completed, but it does not see an intermediate state. This is referred to as serializability because it results in the ability to reload the starting data and replay a series of transactions to end up with the data in the same state it was in after the original transactions were performed.

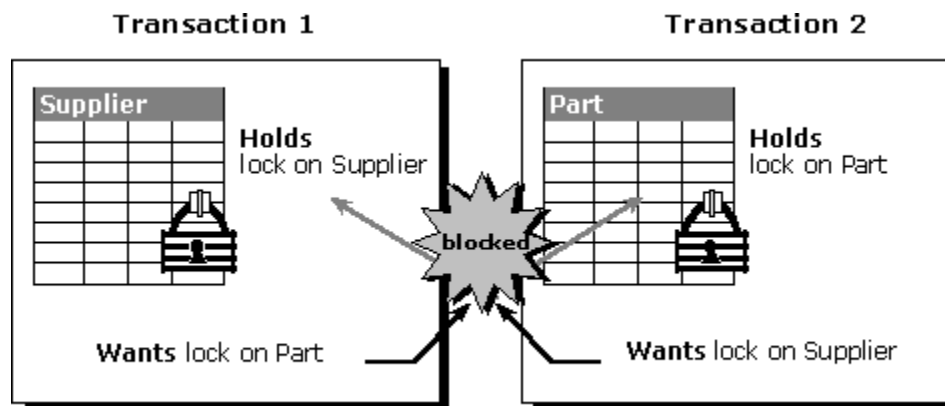
## Durability

After a transaction has completed, its effects are permanently in place in the system. The modifications persist even in the event of a system failure.

42. Microsoft® SQL Server™ 2000 uses locking to ensure transactional integrity and database consistency. Locking prevents users from reading data being changed by other users, and prevents multiple users from changing the same data at the same time. If locking is not used, data within the database may become logically incorrect, and queries executed against that data may produce unexpected results

## 43. DeadLocking:

A deadlock occurs when there is a cyclic dependency between two or more threads for some set of resources.



In this illustration, thread T1 has a dependency on thread T2 for the **Part** table lock resource. Similarly, thread T2 has a dependency on thread T1 for the **Supplier** table lock resource. Because these dependencies form a cycle, there is a deadlock between threads T1 and T2.

44. `select a.empid,a.ename,a.sal from emp a where 3=(select count(distinct(b.sal) from emp b where a.sal<b.sal)`

45. Deleting Duplicate rows in SQL Server:

As there is no pseudo-column like RowID in SQL Server so we cannot make a single line query in SQL Server.

See following example:

```
create table tName (firstname varchar(20));
```

–sample data

```
insert into tName
```

```
select 'Jas' union all
```

```
select 'Raj' union all
```

```
select 'Arsh' union all
```

```
select 'Jas' union all
```

```
select 'Aks'
```

– add temporarily identity column.

```
alter table tName add tid int identity(1,1)
```

–query to delete duplicate rows from table

```
delete from tName where tid not in
```

```
(select min(tid) from tName a where a.firstname = tName.firstname)
```

– drop temporarily added identity column

```
alter table tName drop column tid
```

46. Database Consistency Checker commands are used to check the consistency of database objects in the database.

47. Two types of Authentications are available.

- Windows Authentication
- SQL Server Authentication

48. Two types of Authentication modes are available:

- Windows NT mode
- Mixed mode

49. This is a very open ended question and there could be a lot of reasons behind the poor performance of a query. But some general issues that you could talk about would be: No indexes, table scans, missing or out of date statistics, blocking, excess recompilations of stored procedures, procedures and triggers without SET NOCOUNT ON, poorly written query with unnecessarily complicated joins, too much normalization, excess usage of cursors and temporary tables.