

Q-1 Discuss Feature of SQL server 2000.

SQL Server 2000 includes a number of features that support ease of installation, deployment, and use; scalability; data warehousing; and system integration with other server software.

Ease of Installation, Deployment, and Use

SQL Server 2000 includes many tools and features that simplify the process of installing, deploying, managing, and using databases. SQL Server 2000 provides database administrators with all of the tools that are required to fine-tune SQL Server 2000 installations that run production online systems. SQL Server 2000 is also capable of operating efficiently on a small, single-user system with minimal administrative overhead. The installation or upgrade of SQL Server 2000 is driven by a Graphical User Interface (GUI) application that guides users in providing the information that SQL Server 2000 Setup needs. The Setup program itself automatically detects whether an earlier version of SQL Server is present, and after SQL Server 2000 is installed, it asks users whether they want to launch the SQL Server 2000 Upgrade wizard to quickly guide them through the upgrade process. The entire installation or upgrade process is accomplished quickly and with minimal input from the users.

SQL Server 2000 reconfigures itself automatically and dynamically while running. As more users connect to SQL Server 2000, it can dynamically acquire additional resources, such as memory. As the workload falls, SQL Server 2000 frees the resources back to the system. If other applications are started on the server, SQL Server 2000 will detect the additional allocations of virtual memory to those applications and reduce its use of virtual memory in order to reduce paging overhead.

SQL Server 2000 can also increase or decrease the size of a database automatically as data is inserted or deleted. SQL Server 2000 offers database administrators several tools for managing their systems, such as SQL Server Enterprise Manager and SQL Profiler.

Scalability

The SQL Server 2000 database engine is a robust server that can manage terabyte databases being accessed by thousands of users. At the same time, when running at its default settings, SQL Server 2000 has features such as dynamic self-tuning that enable it to work effectively on laptops and desktops without burdening users with administrative tasks.

SQL Server 2000 includes several features that extend the scalability of the system. For example, SQL Server 2000 dynamically adjusts the granularity of locking to the appropriate level for each table referenced by a query and has high-speed optimizations that support Very Large Database (VLDB) environments. In addition, SQL Server 2000 can build parallel execution plans that split the processing of a SQL statement into several parts. Each part can be run on a different Central Processing Unit (CPU), and the complete result set is built more quickly than if the different parts were executed serially. Many of the features that support the extended scalability of SQL Server 2000 are discussed in more detail throughout the training kit.

Data Warehousing

A data warehouse is a database that is specifically structured to enable flexible queries of the data set and decision-making analysis of the result set. A data warehouse typically contains data representing the business history of an organization. A data mart is a subset of the contents of a data warehouse. A data mart tends to contain data that is focused at the department level, or on a specific business area. SQL Server 2000 includes several components that improve the capability to build data warehouses that effectively support decision support processing needs:

- **Data Warehousing Framework.** A set of components and Application Programming Interfaces (APIs) that implement the data warehousing features of SQL Server 2000.
- **Data Transformation Services (DTS).** A set of services that aids in building a data warehouse or data mart.
- **Meta Data Services.** A set of ActiveX interfaces and information models that define the database schema and data transformations implemented by the Data Warehousing Framework. A schema is a method for defining and organizing data, which is also called metadata.
- **Analysis Services.** A set of services that provide OLAP processing capabilities against heterogeneous OLE DB data sources.

English Query. An application development product that enables users to ask questions in English, rather than in a computer language such as SQL.

System Integration

SQL Server 2000 works with other products to form a stable and secure data store for Internet and intranet systems:

SQL Server 2000 works with Windows 2000 Server and Windows NT Server security and encryption facilities to implement secure data storage. SQL Server 2000 forms a high-performance data storage service for Web applications running under Microsoft Internet Information Services. SQL Server 2000 can be used with Site Server to build and maintain large, sophisticated e-commerce Web sites.

The SQL Server 2000 TCP/IP Sockets communications support can be integrated with Microsoft Proxy Server to implement secure Internet and intranet communications. SQL Server 2000 is scalable to levels of performance capable of handling extremely large Internet sites. In addition, the SQL Server 2000 database engine includes native support for XML, and the Web Assistant Wizard helps you to generate Hypertext Markup Language (HTML) pages from SQL Server 2000 data and to post SQL Server 2000 data to Hypertext Transport Protocol (HTTP) and File Transfer Protocol (FTP) locations. SQL Server supports Windows Authentication, which enables Windows NT and Windows 2000 user and domain accounts to be used as SQL Server 2000 login accounts. Users are validated by Windows 2000 when they connect to the network. When a connection is formed with SQL Server, the SQL Server client software requests a trusted connection, which can be granted only if they have been validated by Windows NT or Windows 2000. SQL Server, then, does not have to validate users separately. Users are not required to have separate logins and

passwords for each SQL Server system to which they connect. SQL Server 2000 can send and receive e-mail and pages from Microsoft Exchange or other Message Application Programming Interface (MAPI)-compliant mail servers. This function enables SQL Server 2000 batches, stored procedures, or triggers to send e-mail. SQL Server 2000 events and alerts can be set to send e-mail or pages automatically to the server administrators in case of severe or pending problems.

Q-2 Explain Different Edition of SQL Server 2000 ?

Editions of SQL Server 2000

SQL Server 2000 is available in different editions to accommodate the unique performance, run-time, and price requirements of different organizations and individuals.

SQL Server 2000 Enterprise Edition. This edition is the complete SQL Server offering for any organization. The Enterprise Edition offers the advanced scalability and reliability features that are necessary for mission-critical line-of-business and Internet scenarios, including Distributed Partitioned Views, log shipping, and enhanced failover clustering. This edition also takes full advantage of the highest-end hardware, with support for up to 32 CPUs and 64 GB of RAM. In addition, the SQL Server 2000 Enterprise Edition includes advanced analysis features.

SQL Server 2000 Standard Edition. This edition is the affordable option for small- and medium-sized organizations that do not require advanced scalability and availability features or all of the more advanced analysis features of the SQL Server 2000 Enterprise Edition. You can use the Standard Edition on symmetric multi-processing systems with up to four CPUs and 2 GB of RAM.

SQL Server 2000 Personal Edition. This edition includes a full set of management tools and most of the functionality of the Standard Edition, but it is optimized for personal use. In addition to running on Microsoft's server operating systems, the Personal Edition runs on non-server operating systems, including Windows 2000 Professional, Windows NT Workstation 4.0, and Windows 98. Dual-processor systems are also supported. While this edition supports databases of any size, its performance is optimized for single users and small workgroups and degrades with workloads generated by more than five concurrent users.

SQL Server 2000 Developer Edition. This SQL Server offering enables developers to build any type of application on top of SQL Server. This edition includes all of the functionality of the Enterprise Edition but with a special development and test end-user license agreement (EULA) that prohibits production deployment.

SQL Server 2000 Desktop Engine (MSDE). This edition has the basic database engine features of SQL Server 2000. This edition does not include a user interface, management tools, analysis capabilities, merge replication support, client access licenses, developer libraries, or Books Online. This edition also limits the database size and user workload. Desktop Engine has the smallest footprint of any edition of SQL Server 2000 and is thus an ideal embedded or offline data store.

SQL Server 2000 Windows CE Edition. This edition is the version of SQL Server 2000 for devices and appliances running Windows CE. This edition is programmatically compatible with the other editions of SQL Server 2000, so

developers can leverage their existing skills and applications to extend the power of a relational data store to solutions running on new classes of devices.

Q-3 Components of SQL Server 2000

SQL Server 2000 Relational Database Engine :-

The SQL Server 2000 relational database engine is a modern, highly scalable engine for storing data. The database engine stores data in tables. Each table represents some object class that is of interest to the organization, such as vehicles, employees, or customers. The table has columns that each represent an attribute of the object modeled by the table (such as weight, name, or cost) and rows that each represent a single occurrence of the type of object modeled by the table (such as the car with license plate number ABC-123 or the employee with ID 123456). An application submits a SQL statement to the database engine, which returns the result to the application in the form of a tabular result set. An Internet application submits either a SQL statement or an XPath query to the database engine, which returns the result as an XML document. The relational database engine provides support for the common Microsoft data access interfaces, such as ActiveX Data Objects (ADO), OLE DB, and Open Database Connectivity (ODBC).

The relational database engine is highly scalable. The SQL Server 2000 Enterprise Edition can support groups of database servers that cooperate to form terabyte databases that are accessed by thousands of users at the same time. The database engine also tunes itself, dynamically acquiring resources as more users connect to the database and then freeing the resources as the users log off. In other words, the smaller editions of SQL Server can be used for individuals or small workgroups that do not have dedicated database administrators. Even large Enterprise Edition database servers running in production are easy to administer by using the GUI administration utilities that are part of the product.

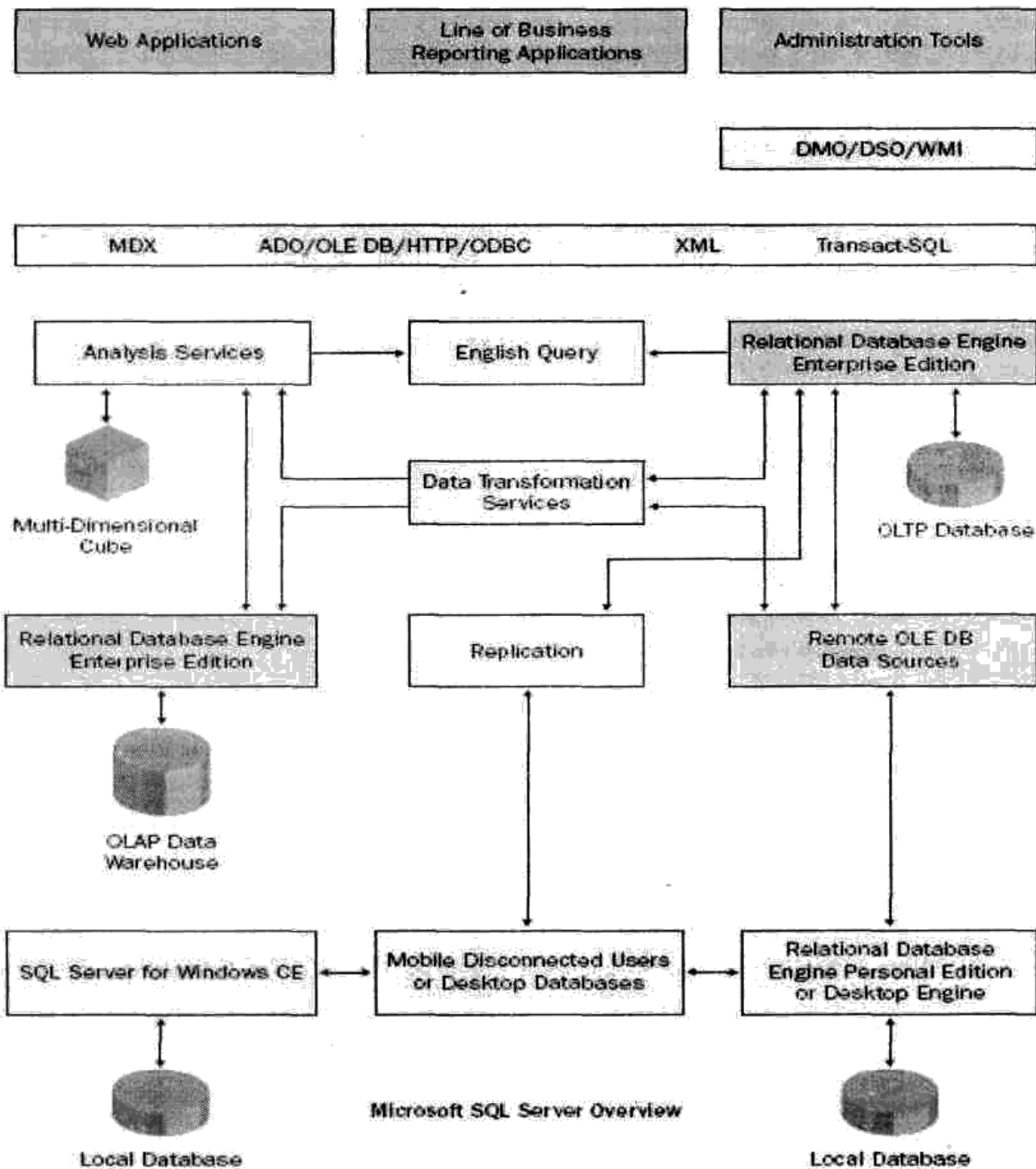


Figure 1.1. The components of SQL Server 2000.

Q-4: Explain Enterprise manager of SQL Server 2000 ?

SQL Server Enterprise Manager

SQL Server Enterprise Manager is the primary administrative tool for SQL Server 2000 and provides a Microsoft Management Console (MMC)-compliant user interface that helps you to perform a variety of administrative tasks:

- Defining groups of servers running SQL Server
- Registering individual servers in a group
- Configuring all SQL Server options for each registered server
- Creating and administering all SQL Server databases, objects, logins, users, and

- permissbns in each registered server
- Defining and executing all SQL Server administrative tasks on each registered server
- Designing and testing SQL statements, batches, and scripts interactively by invoking SQL Query Analyzer
- Invoking the various wizards defined for SQL Server MMC is a tool that presents a common interface for managing different server applications in a Microsoft Windows network. Server applications include a component called a snap-in that presents MMC users with a user interface for managing the server application. SQL Server Enterprise Manager is the Microsoft SQL Server 2000 MMC snap-in.

Q-5 Discuss SQL query analyzer with its all feature.

SQL Query Analyzer is a graphical user interface (GUI) that enables you to design, test, and execute Transact-SQL statements, stored procedures, batches, and scripts interactively. You can run SQL Query Analyzer from inside SQL Enterprise Manager or run it directly from the Start menu. You can also launch SQL Query Analyzer from the command prompt by executing the isqlw utility. (The isqlw utility is discussed in more detail later in this lesson.)

Q-6 Explain Database Architecture of SQL Server 2000 ?

Database Architecture

SQL Server 2000 data is stored in databases. The data in a database is organized into the logical components that are visible to users, while the database itself is 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. Figure 1.2 illustrates the difference between the user view and the physical implementation of a database.

Each instance of SQL Server has four system databases (master, tempdb, msdb, and model) and one or more user databases. Some organizations have only one user database that contains all of their data; some organizations have different databases for each group in their organization. They might also have 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. Some applications use only one database; other applications might access several databases. Figure 1.3 shows the SQL Server system databases and several user databases.

You do not need to run multiple copies of the SQL Server database engine in order for multiple users to access the databases on a server. An instance of the SQL Server Standard Edition or Enterprise Edition is capable of handling thousands of users who are working in multiple databases at the same time. Each instance of SQL Server makes all databases in the instance available to all users who connect to the instance (subject to the defined security permissions)

Database XYZ

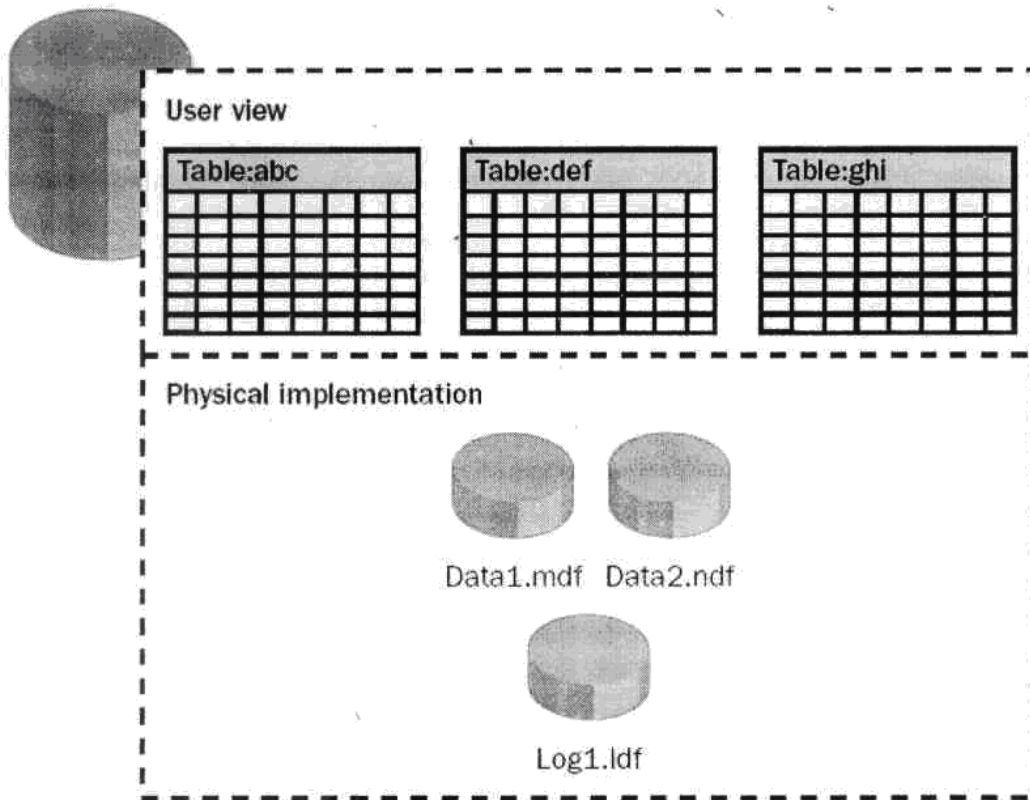


Figure 1.2. User view and physical implementation of a database.

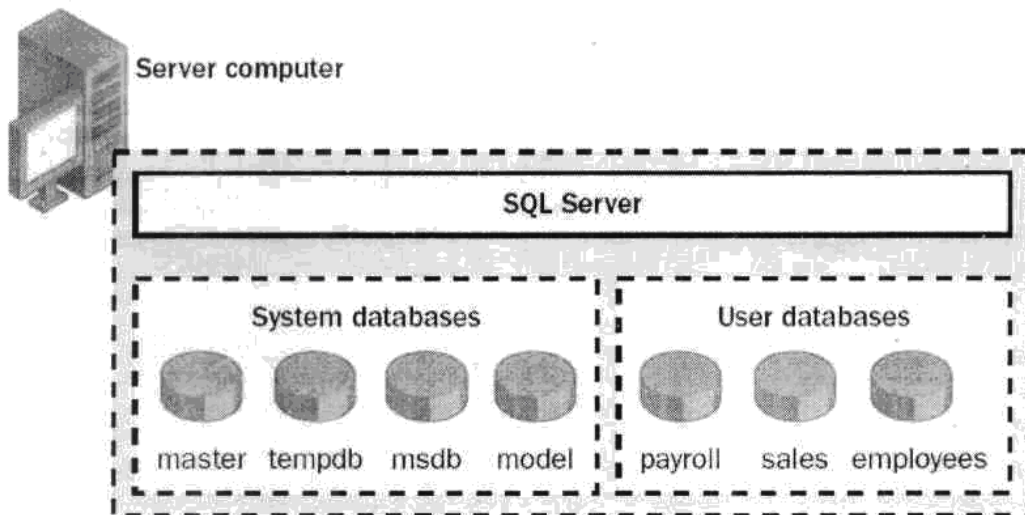


Figure 1.3. System databases and user databases.

If you connect to an instance of SQL Server, your connection is associated with a particular database on the server. This database is called the current database. You are usually connected to a database defined as your default database by the system administrator, although you can use connection options in the database APIs to specify another database. You can switch from one database to another by using either the

Transact-SQL USE <database_name> statement or by using an API function that changes your current database context

SQL Server 2000 enables you to detach a database from an instance of SQL Server, then reattach it to another instance or even attach the database back to the same instance. If you have a SQL Server database file, you can tell SQL Server when you connect to attach that database file using a specific database name.

Q-7 Write a note on Database Objects.

Database Objects

The data in a SQL Server 2000 database is organized into several different objects, which users see when they connect to the database. The following table provides a brief description of the main objects in a database. These objects are discussed in more detail in subsequent chapters.

Object	Description
Table	A two dimensional object consisting of rows and columns that is used to store data a relational database. Each table stores information about one of the types of objects modeled by the database. For example and education database might have one table for teachers a second for students and a third for classes.
Data type	An attribute that specifies what type of information can be stored in a column, parameter or variable SQL server provides system-supplied data types: you can also create user-defined data types.
View	A database object that can be referenced the same way as a table in SQL statements. Views are defined by using SELECT statement and are analogous to an object that contains the result set of this statement.
Stored procedure	A precompiled collection of Transact-SQL statements stored under a name and processed as a unit. SQL Server supplies stored procedures for managing SQL Server and for displaying information about data bases and users. SQL Server applied stored procedures are called System Stored Procedures.
Function	A piece of code that operates as a single logical unit. A function is called by name, accepts optional input parameters, and returns a status and optional output parameters. Many programming languages support functions, including C, Visual Basic and Transact-SQL. Transact-SQL supplies built-in-functions that cannot be modified and supports user-defined functions that users can create and modify.
Index	In a relational database, a database object that provides fast access to data in the rows of a table based on key values. Indexes can also enforce uniqueness on the rows in a table. SQL Server supports clustered and non-clustered indexes. The primary key of a table is automatically indexed. In full-text search, a full text index stores information about significant words and their location within a given column.
Constraint	A property assigned to a table column that prevents certain types of invalid data values from being placed in the column. For example, a

UNIQUE or PRIMARY KEY constraint prevents you from inserting a value that is duplicate of an existing a CHECK constraint prevents you from inserting a value that does not match a search condition and NOT NULL prevents empty values.

- Rule A database object that is bound to columns or use-defined data types and specifies which data values are acceptable in a column. CHECK constraints provide the same function ability and are preferred because they are in the SQL-92 standard.
- Default A data value, option setting, collation, or name assigned automatically by the system if a user does not specify the value, setting, collation, or name also known as an action that is taken automatically at certain events if a user has not specified the action to take.
- Trigger A stored procedure that is executed when data in a specified table is modified. Triggers are often created to enforce referential integrity or consistency among logically related data in different tables.

Q-8 Explain Database Physical Architecture of SQL?

Physical Database Architecture

This section describes the way in which SQL Server 2000 files and databases are organized. The organization of SQL Server 2000 and SQL Server 7.0 is different from the organization of data in SQL Server 6.5 or earlier.

Pages and Extents

The fundamental unit of data storage in SQL Server is the page. In SQL Server 2000, the page size is 8 kilobytes (KB). In other words, SQL Server 2000 databases contain 128 pages per mega byte (MB).

The start of each page is a 96-byte header used to store system information, such as the type of page, the amount of free space on the page, and the object ID of the object owning the page.

Data pages contain all of the data in data rows (except text, ntext, and image data, which are stored in separate pages). Data rows are placed serially on the page (starting immediately after the header). A row offset table starts at the end of the page. The row offset table contains one entry for each row on the page, and each entry records how far the first byte of the row is from the start of the page. The entries in the row offset table are in reverse sequence from the sequence of the rows on the page, as shown in Figure 1.4.

Extents are the basic unit in which space is allocated to tables and indexes. An extent is eight contiguous pages, or 64 KB. In other words, SQL Server 2000 databases have 16 extents per megabyte.

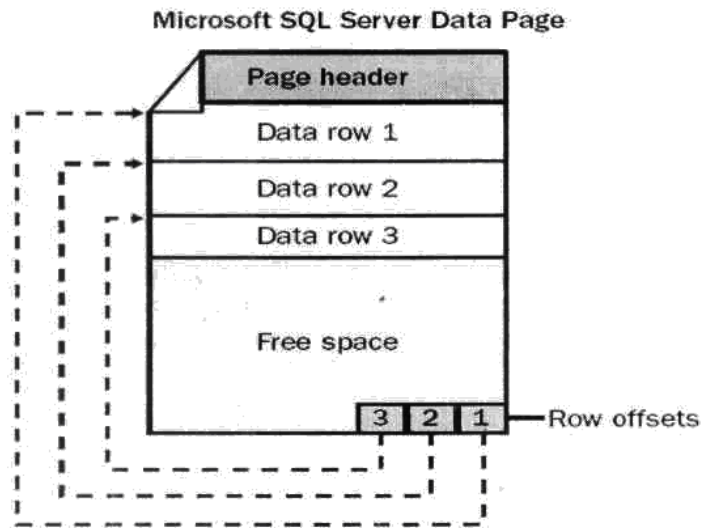


Figure 1.4. Entries in the row offset and rows on the page.

Q-9 Explain Relational database architecture in detail.

The server components of SQL Server 2000 receive SQL statements from clients and process those SQL statements. Figure 1.6 shows the major components involved with processing a SQL statement that is received from a SQL Server client.

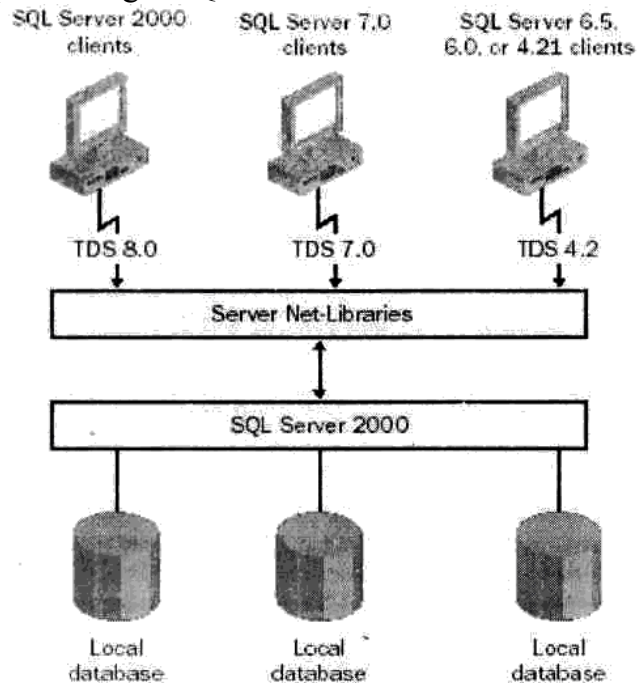


Figure 1.6. Processing a SQL statement that is received from a client.

Tabular Data Stream

SQL statements are sent from clients by using an application-level protocol specific to SQL Server, called Tabular Data Stream (TDS). SQL Server 2000 accepts the following versions of TDS:

TDS 8.0, sent by clients who are running versions of the SQL Server client Components from SQL Server 2000. TDS 8.0 clients support all the features of SQL Server 2000. TDS 7.0, sent by clients who are running versions of the SQL Server client components from SQL Server version 7.0. TDS 7.0 clients do not support features introduced in SQL Server 2000, and the server sometimes has to adjust the data that it sends back to those clients. TDS 4.2, sent by clients who are running SQL Server client components from SQL Server 6.5, 6.0, and 4.21a. TDS 4.2 clients do not support features introduced in either SQL Server 2000 or SQL Server 7.0, and the server sometimes has to adjust the data that it sends back to those clients.

Server Net-Libraries

TDS packets are built by the Microsoft OLE DB Provider for SQL Server, the SQL Server Open Database Connectivity (ODBC) driver, or the DB-Library dynamic link library (DLL). The TDS packets are then passed to a SQL Server client Net-Library, which encapsulates them into network protocol packets. On the server, the network protocol packets are received by a server Net-Library that extracts the TDS packets and passes them to the relational database engine. This process is reversed when results are returned to the client. Each server can be listening simultaneously on several network protocols and will be running one server Net-Library for each protocol on which it is listening.

Relational Database Engine

The database server processes all requests passed to it from the server Net-Libraries. The server then compiles all the SQL statements into execution plans and uses the plans to access the requested data and build the result set that is returned to the client. The relational database engine of SQL Server 2000 has two main parts: the relational engine and the storage engine. One of the most important architectural changes made in SQL Server 7.0 (and carried over to SQL Server 2000) was to strictly separate the relational and storage engine components within the server and to have them use the OLE DB API to communicate with each other, as shown in Figure 1.7.

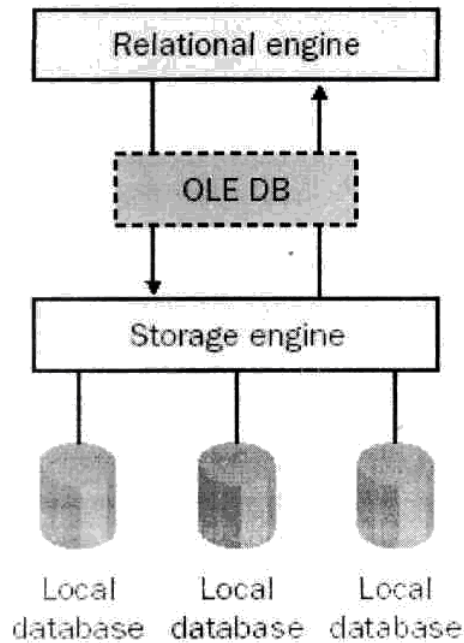


Figure 1.7. Relational engine components.

Q-10 SQL Query Analyze

SQL Query Analyzer is a graphical user interface (GUI) that enables you to design, test, and execute Transact-SQL statements, stored procedures, batches, and scripts interactively. You can run SQL Query Analyzer from inside SQL Enterprise Manager or run it directly from the Start menu. You can also launch SQL Query Analyzer from the command prompt by executing the isqlw utility. (The isqlw utility is discussed in more detail later in this lesson.)

The functionality within SQL Query Analyzer can be described in terms of the interface layout. SQL Query Analyzer includes a number of windows, dialog boxes, and wizards that help you to perform the tasks necessary to manage SQL Server databases and the data stored within those databases. This section discusses many of these interface objects and the functions that can be performed when you access them. For more details about any of the objects that are discussed here or any objects within the interface, refer to SQL Server Books Online.

When you launch SQL Query Analyzer, the Connect To SQL Server dialog box appears. You must specify which instance of SQL Server that you want to access and which type of authenticating to use when connecting to the database. Once you have entered the appropriate information in the Connect To SQL Server dialog box and then clicked OK, SQL Query Analyzer appears and displays the Query window and the Object Browser window, as shown in Figure 2.1.

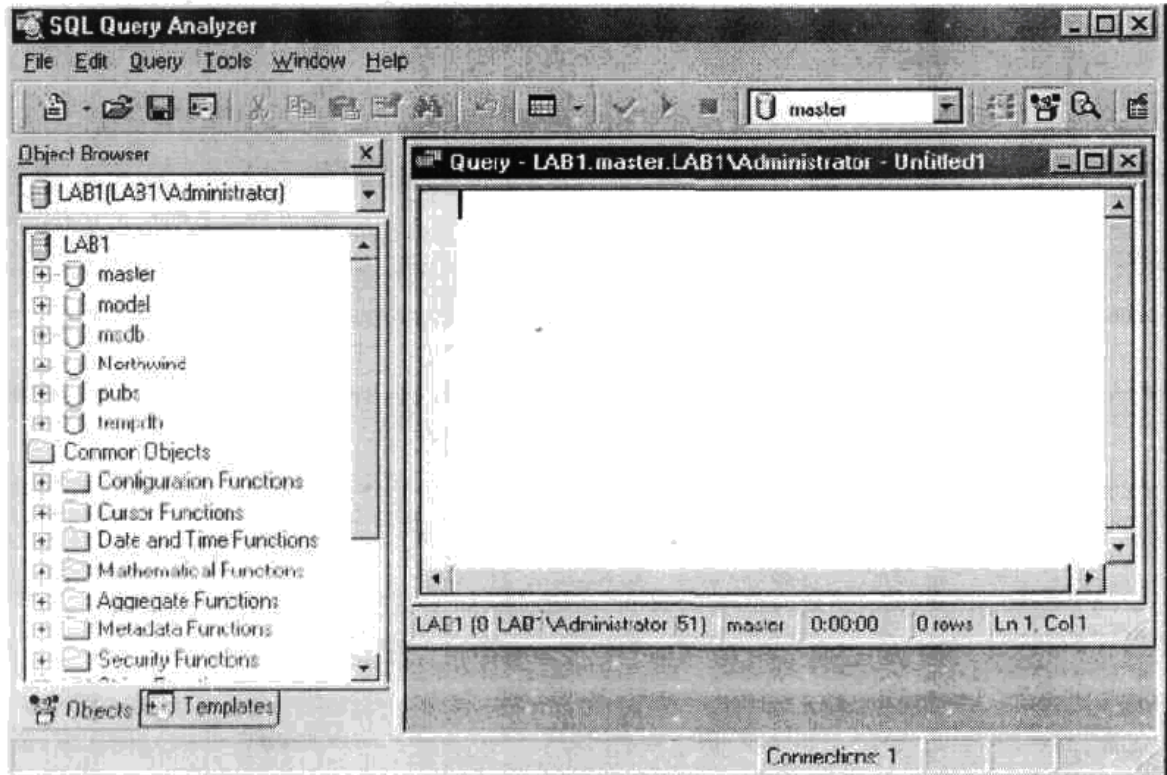


Figure 2.1. SQL Query Analyzer displaying the Query window on the right and the Object Browser window on the left.

Query Window

The Query window is divided into two panes: The Editor pane and the Results pane. When you first open SQL Query Analyzer, only the Editor pane appears, as shown in Figure 2.1. The Results pane appears automatically when you run a Transact-SQL query. You can also open and close the Results pane manually by clicking the Show Results Pane button on the toolbar.

You can customize the window and control the behavior of the Editor pane and the Result pane. The Options dialog box, which you can access from the Tools menu .enables you to control the box and behavior of the Query window. In addition, you can specify which fonts are used for text in the window and you can change the relative size of the Editor pane and the Results pane by dragging the split bar up and down. You can also scroll through the panes (up and down or left and right) as necessary.

Editor Pane :-

The Editor pane is a text-editing window used to enter and execute Transact-SQL statements. You can use one of the following methods to enter code in the Editor pane:

- Type SQL statements directly in the Editor pane.
- Open a saved SQL script. The contents are displayed in the Editor pane .where they can be edited.
- Open a template file. The contents are displayed in frie Editor pane, where they

can be edited.

- Use the scripting features of Object Browser to copy SQL statements for the selected database object into the Editor pane.

The Editor pane in SQL Query Analyzer provides various tools to help you create and edit Transact-SQL statements, including the standard editing commands Undo, Cut, Copy, Paste, and Select All. You can also find and replace text, move the input cursor to a particular line, insert and remove indentation, force case, and insert and remove comment marks.

In addition, you can view Transact-SQL reference topics at SQL Server Books Online and copy the syntax example from the reference into the Editor pane, in order to help create a Transact-SQL statement. You can also save query definitions and other SQL scripts for reuse, and you can create templates (which are boilerplate scripts for creating objects in a database).

Color Coding in Query Analyzer :-

The code entered in the Editor pane is colored by category. The following table lists the default colors and what they indicate:

Color	Category
Red	Character string
Dark red	Stored procedure
Green	System table
Dark green	Comment
Magenta	System function
Blue	Keyword
Gray	Operator

You should use the color coding as a guide to help eliminate errors in your Transact-SQL statements. For example, if you type a keyword and it is not displayed in blue (assuming that you retained the default settings), the keyword might be misspelled or incorrect. Or, if too much of your code is displayed in red, you might have omitted the closing quotation mark for a character string.

Executing Transact-SQL Statements :-

You can either execute a complete script or only selected SQL statements in SQL Query Analyzer:

- Execute a complete script by creating or opening the script in the Editor pane and then pressing F5.
- Execute only selected SQL statements by highlighting the lines of code in the Editor pane and then pressing F5.

When executing a stored procedure in the Editor pane, enter the statement to execute the stored procedure and then press F5. If the statement that executes the procedure is the first in the batch, you can omit the EXECUTE (or EXEC) statement; otherwise, EXECUTE is required.

Results Pane

When you execute a Transact-SQL statement, the query output (result set) is displayed in the Results pane. The Results pane can include a variety of tabs. The options that you select in the interface determine which tabs are displayed. By default, only the Grids tab, which is the active tab, and the Messages tab are displayed.

Grids Tab :-

The Grids tab displays the result set in a grid format, as shown in Figure 2.2. The grid format is displayed much like a table and enables you to select individual cells, columns, or rows from the result set.

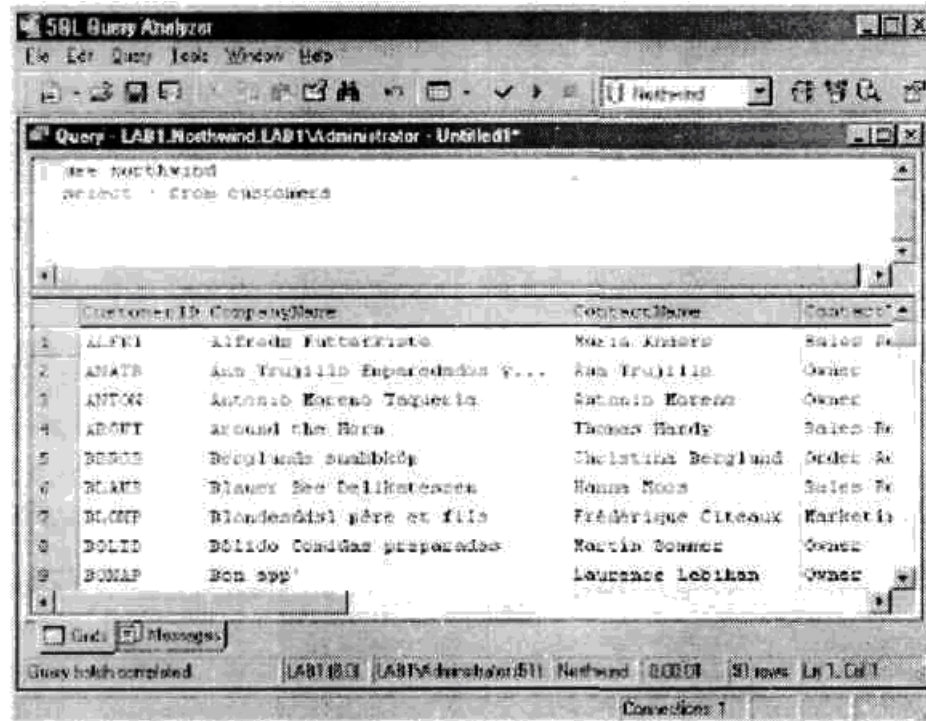


Figure 2.2. Grids tab displaying the result set generated by the executing

Transact-SQL statement.

The Grids tab is always accompanied by the Messages tab, which displays messages relative to the specific query.

Results Tab

The Results tab, like the Grids tab, displays the result set generated by executing a Transact-SQL statement. In the Results tab, however, the result set is displayed as text (refer to Figure 2.3), rather than in a grid format.



Figure 2.4. Execution Plan tab displaying a graphical representation of the executed Transact-SQL statement.

Trace Tab

The Trace tab, like the Execution Plan tab, can assist you with analyzing your queries. The Trace tab displays servertrace information about the event class, subclass, integer data, text data, database D, duration, start time, reads and writes, and Central Processing Unit (CPU) usage, as shown in Figure 2.5. The Trace tab provides information that you can use to determine the server-side impact of a query.

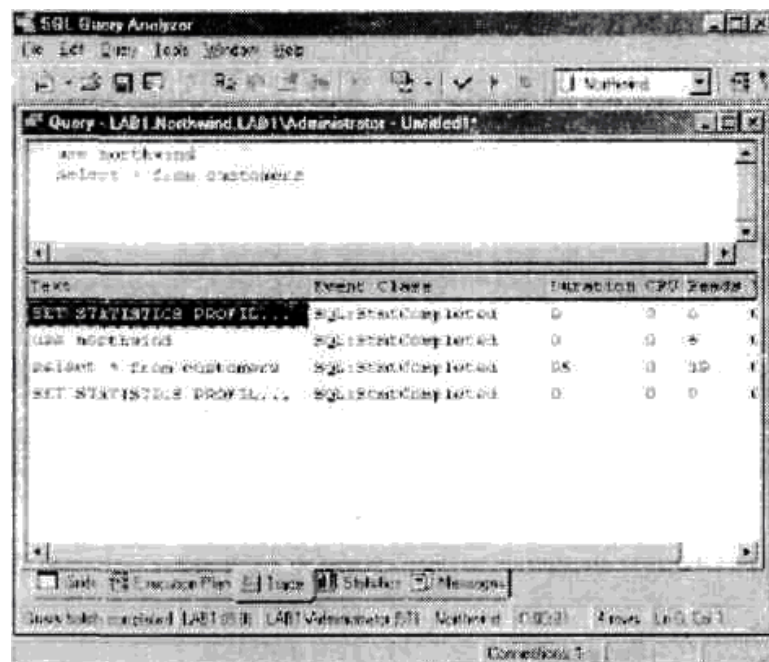


Figure 2.5. Trace tab displaying servertrace information about the executed Transact-SQL statement.

By default, the Trace tab is not displayed in the Results pane. To display the Trace tab, select the Execute Mode button on the toolbar, then select Show Server Trace. The next time you execute a query, the Trace tab will be available, and it will show the servertrace information. The tab will be available until you deselect the Show Server Trace option or until you close SQL Query Analyzer.

Messages Tab

The Messages tab displays messages about the Transact-SQL statement that you executed (or that you tried to execute). If the query ran successfully, the message will include the number of rows returned, as shown in Figure 2.7, or it will state that the command has completed successfully- If the query did not run successfully, the Messages tab will contain an error message identifying why the query attempt was unsuccessful.

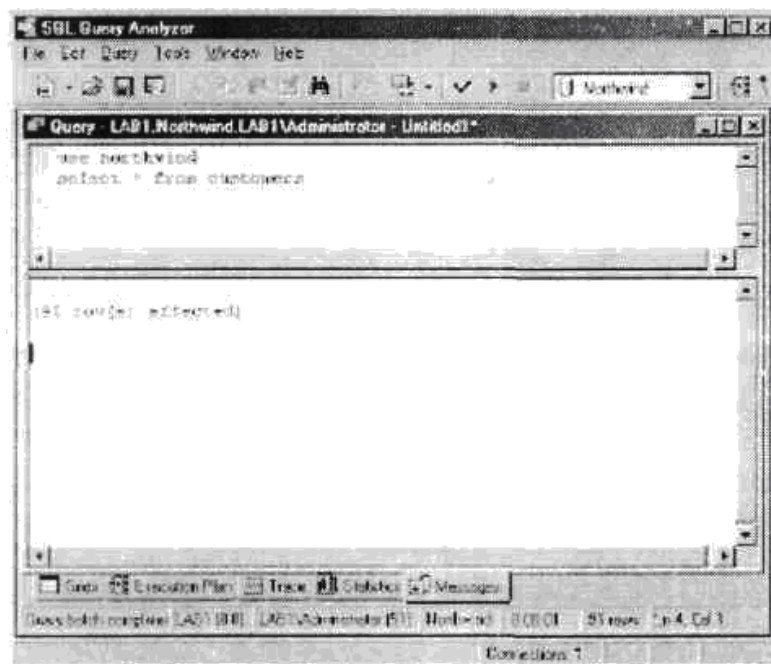


Figure 2.7. The Messages tab displaying a message about the executed Transact-SQL statement.

The Messages tab is available in the Results pane only if the Grids tab is displayed. If the Results tab is displayed, messages appear in that tab.

Q-11 Explain Transact-SQL Statements ?

Transact-SQL Statements

A Transact-SQL statement is a set of code that performs some action on database objects or on data in a database. SQL Server supports three types of Transact-SQL statements: DDL, DCL, and DML.

Data Definition Language

Data definition language, which is usually part of a database management system, is used to define and manage all attributes and properties of a database, including row layouts, column definitions, key columns, file formats, and storage

strategy.

A DDL statement supports the definition or declaration of database objects such as databases, tables, and views. The Transact-SQL DDL used to manage objects is based on SQL-92 DDL statements (with extensions). For each object class, there are usually CREATE, ALTER, and DROP statements (for example, CREATE TABLE, ALTER TABLE, and DROP TABLE).

Most DDL statements take the following form:

```
| CREATE object_name  
| ALTER object_name  
| DROP object_name
```

The following three examples illustrate how to use the Transact-SQL CREATE keyword to create, alter, and drop tables. CREATE is not limited only to table objects, however.

CREATE TABLE

The CREATE TABLE statement creates a table in an existing database. The following statement will create a table named Importers in the Northwind database. The table will include three columns: CompanyID, CompanyName, and Contact.

```
USE Northwind  
CREATE TABLE Importers  
(  
  CompanyID int NOT NULL,  
  CompanyName varchar(40) NOT NULL,  
  Contact varchar(40) NOT NULL  
)
```

ALTER TABLE

The ALTER TABLE statement enables you to modify a table definition by altering, adding, or dropping columns and constraints or by disabling or enabling constraints and triggers. The following statement will alter the Importers table in the Northwind database by adding a column named ContactTitle to the table.

```
USE Northwind  
ALTER TABLE Importers  
ADD ContactTitle varchar(20) NULL
```

DROP TABLE

The DROP TABLE statement removes a table definition and all data, indexes, triggers, constraints, and permission specifications for that table. Any view or stored procedure that references the dropped table must be explicitly dropped by using the DROP VIEW or DROP PROCEDURE statement. The following statement drops the Importers table from the Northwind database.

```
USE Northwind  
DROP TABLE Importers
```

Data Control Language

Data control language is used to control permissions on database objects. Permissions are controlled by using the SQL-92 GRANT and REVOKE statements and the Transact-SQL DENY statement.

GRANT

The GRANT statement creates an entry in the security system that enables a user in the current database to work with data in that database or to execute specific Transact-SQL statements. The following statement grants the Public role SELECT permission on the Customers table in the Northwind database:

```
USE Northwind
GRANT SELECT
ON Customers
TO PUBLIC
```

REVOKE

The REVOKE statement removes a previously granted or denied permission from a user in the current database. The following statement revokes the SELECT permission from the Public role for the Customers table in the Northwind database:

```
USE Northwind
REVOKE SELECT
ON Customers TO
PUBLIC
```

DENY

The DENY statement creates an entry in the security system that denies a permission from a security account in the current database and prevents the security account from inheriting the permission through its group or role memberships.

```
USE Northwind
DENY SELECT
ON Customers
TO PUBLIC
```

Data Manipulation Language

Data manipulation language is used to select, insert, update, and delete data in the objects defined with DDL.

SELECT

The SELECT statement retrieves rows from the database and enables the selection of one or many rows or columns from one or many tables. The following statement retrieves the CustomerID, CompanyName, and ContactName data for companies who have a CustomerID value equal to 1000. The result set is

ordered according to the ContactName value:

```
USE Northwind
SELECT CustomerD, CompanyName, ContactName
FROM Customers
WHERE (CustomerID = 'alfkf OR CustomerD = 'anat/)
ORDER BY ContactName
```

INSERT

An INSERT statement adds a new row to a table or a view. The following statement adds a row to the Territories table in the Northwind database. The TerritoryID value for the new row is 98101; the TerritoryDescription value is Seattle; and the RegionID value is 2.

```
USE Northwind
INSERT INTO Territories VALUES
(98101,'Seattle', 2)
```

Note The INTO keyword is an optional keyword that can be used between INSERT and the target table. Use the INTO keyword for code clarity.

UPDATE

The UPDATE statement changes data in a table. The following statement updates the row in the Territories table (in the Northwind database) whose TerritoryID value is 98101. The Territory Description value will be changed to Downtown Seattle.

```
USE Northwind
UPDATE Territories
SET TerritoryDescription = 'Downtown Seattle'
WHERE TerritoryID = 98101
```

DELETE

The DELETE statement removes rows from a table. The following statement removes the row from the Territories table (from the Northwind database) whose TerritoryID value is 98101.

```
USE Northwind
DELETE FROM Territories
WHERE TerritoryID = 98101
```

Q-12 Discuss different type of function with a reference of sql server 2000.

Functions

A function encapsulates frequently performed logic in a subroutine made up of one or more Transact-SQL statements. Any code that must perform the logic incorporated in a function can call the function rather than having to repeat all of the

function logic. SQL Server2000 supports two types of functions:

Built-in functions. These functions operate as defined in Transact-SQL and cannot be modified. The functions can be referenced only in Transact-SQL statements.

User-defined functions. These functions enable you to define your own Transact-SQL functions by using the CREATE FUNCTION statement.

Built-in Functions

The Transact-SQL programming language contains three types of built-in functions: rowset, aggregate, and scalar.

Rowset Functions

Rowset functions can be used like table references in a Transact-SQL statement. These functions return an object that can be used in place of a table reference in a Transact-SQL statement. For example, the OPENQUERY function is a rowset function that executes the specified pass-through query on the given linked server, which is an OLE DB data source. The OPENQUERY function can be referenced in the FROM clause of a query as though it were a table name. All rowset functions are non-deterministic; that is, they do not return the same result every time they are called with a specific set of input values. Function determinism is discussed in more detail later in this section.

Aggregate Functions

Aggregate functions operate on a collection of values but return a single, summarizing value. For example, the AVG function is an aggregate function that returns the average of the values in a group.

Aggregate functions are allowed as expressions only in the following statement: The select list of a SELECT statement (either a subquery or an outer query) A COMPUTE or COMPUTE BY clause A HAVING clause

With the exception of COUNT, aggregate functions ignore null values. Aggregate functions are often used with the GROUP BY clause of the SELECT statement. All aggregate functions are deterministic; they return the same value any time they are called with a given set of input values.

Scalar Functions

Scalar functions operate on a single value and then return a single value. Scalar functions can be used wherever an expression is valid. Scalar functions are divided into categories, as described in the following table:

Scalar Category	Description
Configuration functions	Return information about the current configuration
Cursor functions	Return information about cursors
Date and time functions	Perform an operation on a date and a time input value and
	return either a string, numeric, or date and time value

Mathematical functions	Perform a calculation based on input values provided as parameters to the function and return a numeric value
Metadata functions	Return information about the database and database objects
Security functions	Return information about users and roles
String functions	Perform an operation on a string (<i>char</i> or <i>varchar</i>) input value and return a string or numeric value
System functions	Perform operations and return information about values, objects, and settings in SQL Server
System statistical Functions	Return statistical information about the system
Text and image functions	Perform an operation on a text or image input value or column and return information about the value

Each category of scalar functions includes its own set of functions. For example, the MONTH function, which is included in the date and time category, is a scalar function that returns an integer representing the month part of a specified date.

Q-14 Discuss processing step of SQL select Statements ?

Processing a SELECT Statement

The steps used to process a single SELECT statement referencing only local base tables (no views or remote tables) illustrate the basic process of executing most Transact-SQL statements. SQL Server uses the following steps to process a single SELECT statement:

1. The parser scans the SELECT statement and breaks it into logical units, such as keywords, expressions, operators, and identifiers.
2. A query tree, sometimes called a sequence tree, is built by describing the logical steps needed to transform the source data into the format needed by the result set.
3. The query optimizer analyzes all of the ways in which the source tables can be accessed and selects the series of steps that will return the result fastest while consuming the fewest resources. The query tree is updated to record this exact series of steps, and the final, optimized version of the query tree is called the execution plan.
4. The relational engine begins executing the execution plan. As steps that need data from the base tables are processed, the relational engine uses OLE DB to request the storage engine to pass up data from the row sets that are requested from the relational engine.
5. The relational engine processes the data returned from the storage engine into the format defined for the result set and returns the result set to the client.

Q-15 Discuss the feature of well designed database

In relational database design theory, normalization rules identify certain attributes that must be present or absent in a well-designed database. These rules can

become quite complicated and go well beyond the scope of this book. There are a few rules that can help you achieve a sound database design, however. A table should have an identifier, it should store data for only a single type of entity, it should avoid nullable columns, and it should not have repeating values or columns

Q-16 Explain Entity Relationship with Suitable Example ?

Entity Relationships

In a relational database, relationships help to prevent redundant data. A relationship works by matching data in key columns—usually columns that have the same name in both tables. In most cases, the relationship matches the primary key from one table, which provides a unique identifier for each row with an entry in the foreign key in the other table. Primary keys and foreign keys are discussed in more detail in Chapter 5, "Implementing Data Integrity." There are three types of relationships between tables: one-to-one, one-to-many, and many-to-many. The type of relationship depends on how the related columns are defined.

One-to-One Relationships

In a one-to-one relationship, a row in table A can have no more than one matching row in table B (and vice versa). A one-to-one relationship is created if both of the related columns are primary keys or have unique constraints. This type of relationship is not common, however, because information related in this way would usually be in one table.

One-to-Many Relationships

A one-to-many relationship is the most common type of relationship. In this type of relationship, a row in table A can have many matching rows in table B, but a row in table B can have only one matching row in table A. For example, the Publishers and Titles tables mentioned previously have a one-to-many relationship. Each publisher produces many titles, but each title comes from only one publisher. A one-to-many relationship is created if only one of the related columns is a primary key or has a unique constraint.

Many-to-Many Relationships

In a many-to-many relationship, a row in table A can have many matching rows in table B (and vice versa). You create such a relationship by defining a third table, called a junction table, whose primary key consists of the foreign keys from both table A and table B. In Figures 3-6 and 3-7, you saw how the author information could be separated into another table. The Books table and the Authors table have a many-to-many relationship. Each of these tables has a one-to-many relationship with the BookAuthor table, which serves as the junction table between the two primary tables

Q-17 What is a file? Discuss different kind of file group. Files and Filegroups

To map a database, SQL Server 2000 uses a set of operating system files. / data

and objects in the database, such as tables, stored procedures, trigger: and views, are stored within the following types of operating system files:

Primary. This file contains the startup information for the database and is used to store data. Every database has one primary data file.

Secondary. These files hold all of the data that does not fit into 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 might be large enough to need multiple secondary data files or to use secondary files on separate disk drives to spread data across multiple disks or to improve database performance.

Transaction Log. These files hold the log information used to recover the database. There must be at least one log file for each database. A simple database can be created with one primary file that contains all data and objects and a log file that contains the transaction log information. Alternatively, a more complex database can be created with one primary file and five secondary files. The data and objects within the database spread across all six files, and four additional bg files contain the transaction bg information. Filegroups group files 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 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. The same performance improvement can be accomplished with a single file created on a redundant array of independent disks (RAID) stripe set. Files and filegroups, however, help to easily add new files to new disks. Additionally, if your database exceeds the maximum size for a single Windows NT file, you can use secondary data files to grow your database further.

Rules for Designing Files and Filegroups When designing files and filegroups, you should adhere to the following rules: A file or filegroup cannot be used by more than one database. For example, the files sales.mdf and sales .ndf, which contain data and objects from the sales database, cannot be used by any other database.

A file can be a member of only one filegroup. Data and transaction bg information cannot be part of the same file or filegroup. Transaction bg files are never part of a filegroup. **Default Filegroups**

A database comprises a primary filegroup and any user-defined file groups. The filegroup that contains the primary file is the primary filegroup. When a database is created, the primary filegroup contains the primary data file and any other files that are not put into another filegroup. All system tables are allocated in the primary filegroup. If the primary filegroup runs out of space, no new catalog information can be added to the system tables. The primary filegroup is filled only if autogrow is turned off or if all of the disks that are holding the files in the primary filegroup run out of space. If this situation happens, either turn autogrow back on or move other files off the disk to free more space. User-defined filegroups are any filegroups that are specifically created by the user when he or she is first creating or later altering the database. If a user-defined filegroup fills up, only the users' tables specifically allocated to that filegroup would

be affected. At any time, exactly one filegroup is designated as the default filegroup. When objects are created in the database without specifying to which filegroup they belong, they are assigned to the default filegroup. The default filegroup must be large enough to hold any objects not allocated to a user-defined filegroup. Initially, the primary filegroup is the default filegroup. The default filegroup can be changed by using the ALTER DATABASE statement. When you change the default filegroup, any objects that do not have a filegroup specified when they are created are allocated to the data files in the new default filegroup. Allocation for the system objects and tables, however, remains within the primary filegroup, not in the new default filegroup.

Changing the default filegroup prevents user objects that are not specifically created on a user-defined filegroup from competing with the system objects and tables for data space.

Recommendations

When implementing a database, you should try to adhere to the following guidelines for using files and filegroups:

- Most databases will work well with a single data file and a single transaction log file.
- If you use multiple files, create a second filegroup for the additional files and make that filegroup the default filegroup. This way, the primary file will contain only system tables and objects.
- To maximize performance, you can create files or filegroups on as many different available local physical disks as possible and place objects that compete heavily for space in different filegroups.
- Use filegroups to enable the placement of objects on specific physical disks.
- Place different tables used in the same join queries in different filegroups. This procedure will improve performance due to parallel disk input/output (I/O) searching for joined data.
- Place heavily accessed tables and the non-clustered indexes belonging to those tables on different filegroups. This procedure will improve performance due to parallel I/O if the files are located on different physical disks.
- Do not place the transaction log file(s) on the same physical disk with the other files and filegroups.

Transaction Logs

A database in SQL Server 2000 has at least one data file and one transaction log file. Data and transaction log information is never mixed on the same file, and individual files are used by only one database. SQL Server uses the transaction log of each database to recover transactions. The transaction log is a serial record of all modifications that have occurred in the database as well as the transactions that performed the modifications. The transaction log records the start of each transaction and records the changes to the data. This log has enough information to undo the modifications (if necessary later) made during each transaction. For some large operations, such as CREATE INDEX, the transaction log instead records the fact that the operation took place. The log grows continuously as logged operations occur in the

database. The transaction log records the allocation and dealcatbn of pages and the commit or rollback of each transaction. This feature enables SQL Server to either apply (roll forward) or back out (roll back) each transaction in the following ways:

A transaction is rolled forward when you apply a transaction bg. SQL Server copies the afterimage of every modification to the database or reruns statements such as CREATE INDEX. These actions are applied in the same sequence in which they originally occurred. At the end of this process, the database is in the same state that it was in atthe time the transaction log was backed up.

A transaction is rolled back when you back out an incomplete transactbn. SQL Server copies the before images of all modifications to the database since the BEGIN TRANSACTION. If it encounters transaction log records indicating that a CREATE INDEX was performed, it performs operations that logically reverse the statement. These before images and CREATE INDEX reversals are applied in the reverse of their original sequence. At a checkpoint, SQL Server ensures that all transaction log records and database pages that were modified are written to disk During each database's recovery process, which occurs when SQL Server is restarted, a transaction needs to be rolled forward only if it is not known whether all the transaction's data modifications were actually written from the SQL Server buffer cache to disk. Because a checkpoint forces all modified pages to disk, it represents the point at which the startup recovery must start rolling forward transactors. Because all pages modified before the checkpoint are guaranteed to be on disk, there is no need to roll forward anything done before the checkpoint. Transaction log backups enable you to recover the database to a specific point in time (for example, prior to entering unwanted data) or to the point of failure. Transaction log backups should be a consideration in your media-recovery strategy.

Q-18 What is Data Type ? Explain System Define, User Define, Column level Data Type ?

A data type is an attribute that specifies what type of data can be stored in a column, parameter, or variable. SQL Server provides a set of system-supplied data types. In addition, you can create user-defined data types that are based on the system supplied data types. This lesson describes system-supplied data types and user-defined data types, and it explains how to identify which data type you should use when defining a column.

System-Supplied Data Types

In SQL Server, each column has a related data type, which is an attribute that specifies the type of data (integer, character, monetary, and so on) that the object can hold. Certain objects other than columns also have an associated data type. The following objects have data types:

- | Columns in tables and views
- | Parameters in stored procedures
- | Variables
- | Transact-SQL functions that return one or more data values of a specific data type

| Stored procedures that have a return code (which always has an integer data type)

Assigning a data type to each column is one of the first steps to take toward designing a table. SQL Server supplies a set of system data types that define all of the types of data that you can use with SQL Server. You can use data types to enforce data integrity, because the data that is entered or changed must conform to the type specified in the original CREATE TABLE statement. For example, you cannot store someone's last name in a column defined with the datetime data type because a date time column accepts only valid dates. Assigning a data type to an object defines four attributes of the object

The kind of data contained by the object. For example, the data might be character, integer, or binary.

The length of the stored value or its size. The lengths of image, binary, and varbinary data types are defined in bytes. The length of any of the numeric date types is the number of bytes required to hold the number of digits allowed for that data type. The lengths of character string and Unicode data types are defined in characters.

The precision of the number (numeric data types only). The precision is the number of digits that the number can contain. For example, a smallint object can hold a maximum of five digits; therefore, it has a precision of five.

The scale of the number (numeric data types only). The scale is the number of digits that can be stored to the right of the decimal point. For example, an int object cannot accept a decimal point and has a scale of zero. A money object can have a maximum of four digits to the right of the decimal point and has a scale of four. The following table provides descriptions of the categories of data types that SQL Server supports and descriptions of the base data types that each category contains:

Category	Description	Base Data Type	Description
Binary	Binary data stores strings of bits. The data consists of hexa-decimal numbers. For example, the decimal number 245 is hexadecimal F5.	Binary	Data must have the same fixed length (up to 8 KB).
		varbinary	Data can vary in the number of hexa-decimal digits (up to 8 KB).
		image	Data can be variable length and exceed 8KB.
Character	Character data consists of any combination of letters, symbols, and numeric characters. For example, valid character data includes the "Johu928" and •(0*&(%B99nhjkJ" combinations.	char	Data must have same fixed length (up to 8 KB).

		varchar	Data can vary in the number of characters, but the length cannot exceed 8 KB,
		Text	Data can be ASCII characters that exceed 8 KB,
Date and time	Date and time data consists of valid date and time combinations. There are no separate time and date data types for storing only times or only dates	Datetime	Date data should range from January 1, 1753 through December 31, 9999 (requires 8 bytes per value).

Category	Description	Base Data Type	Description
		smalldatetime	Date data should range from January 1, 1900 through June 6, 2079 (requires 4 bytes per value).
Decimal	Decimal data consists of data that is stored to the least significant digit.	decimal	Data can be a maximum of 38 digits all of which can be to the right of the decimal point. The data type stores an exact representation of the number, there is no approximation of the stored value.
		numeric	In SQL Server, the numeric data type is equivalent to the decimal data type.
Floating point	Approximate numeric (floating-point) data consists of data preserved as accurately as the binary numbering system can offer.	float	Data is a floating-point number from $-1.79E + 308$ through $1.79E - 308$.
		real	Data is a floating-point number from $-3.40E + 38$ through $3.40E - 38$.
Integer	Integer data consists of negative and positive whole numbers. such as -15, 0, 5, and 2,509	bigint	Data is a number in the range from -2^{63} through $2^{63}-1$ (9223372036854775807). Storage size is 8 bytes.

		Int	Data is a number in the range from -2.147.483.648 through 2.147.483.647 only (requires 4 bytes of storage per value).
		smallint	Data is a number in the range from -32.768 through 32.767 only (requires 2 bytes of storage per value).
		tinyint	Data is a number in the range from zero through 255 only (requires 1 byte of storage per value).
Monetary	Monetary data represents positive or negative amounts of money.	money	Data is a monetary value in the range from -922,337,203,685.4 through +922,337,203,685,477.5807 (requires 8 bytes to store value).
		smallmoney	Data is a monetary value in the range of -214,748.3648 through 214,748.3647 (requires 4 bytes to store a value).

Category	Description	Base Data Type	Description
Special	Special data consists of data that does not fit in any of the other categories of data.	Bit	Data consists of either a 1 or a 0. Use the bit data type when representing TRUE or FALSE or YES or NO.
		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.
		timestamp	This data type is used to indicate the sequence of SQL Server activity on a row and is represented as an increasing number in a binary format.

		uniqueidentifier	Data consists of a 16-byte hexadecimal number indicating a globally unique identifier (GUID) The GUID is useful when a row must be unique among many other rows.
		SQL_variant	This data type stores values of various SQL Server—supported data types except text, ntext, timestamp, image, and sql variant.
		Table	This data type is used to store a result set for later processing. The table data type can be used only to define local variables of type table or the return value of a user-defined function.
Unicode	Using Unicode data types, a column can store any character defined by the Unicode Standard, which includes all of the characters defined in the various character sets. Unicode data types take twice as much storage space as non-Unicode data types.	nchar	Data must have the same fixed length (up to 4000 Unicode characters)
		Nvarchar	Data can vary in the number of Unicode characters (up to 4000).
		Ntext	Data can exceed 4000 Unicode characters

All data stored in SQL Server must be compatible with one of these base data types. The cursor data type is the only base data type that cannot be assigned to a table column. You can use this type only for variables and stored procedure parameters

Several base data types have synonyms (for example, rowversion is a synonym for timestamp, and national character varying is a synonym for nvarchar).

User-Defined Data Types

User-defined data types are based on the system data types in SQL Server 2000. User-defined data types can be used when several tables must store the same type of data in a column and you must ensure that these columns have exactly the same data type, length, and nullability. For example, a user-defined data type called postalcode could be created based on the char data type. When you create a user-defined data

type, you must supply the following parameters:

| Name

| System data type upon which the new data type is based

| Nullability (whether the data type allows null values) When nullability is not explicitly defined, it will be assigned based on the ANSI null default setting for the database or connection.

Note : If a user-defined data type is created in the Model database, it exists in all new user-defined databases. If the data type is created in a user-defined database, however, the data type exists only in that user-defined database. You can create a user-defined data type by using the `sp_addtype` system stored procedure or by using SQL Server Enterprise Manager.

Column Data Types

In this exercise, you will identify the data types that you should use in your column definitions when you create the tables for the database that you created in Exercise

1. The tables and columns will be based on the objects and data constraints that you identified when you developed your database design. You will use system-supplied base data types for your database, rather than user-defined data types. Each column must have a data type. To perform this exercise, you will need paper and a pencil to write down the data type for each column.

To review existing tables and columns and their data types

1. Open SQL Server Enterprise Manager.
2. Expand the console tree until you can view the list of objects in the Northwind database.
3. Click the Tables node listed beneath the Northwind node. A list of tables in the Northwind database appears in the right pane.
4. Right-click the Employees table, then click Properties. The Table Properties -Employees dialog box appears.
5. Review the list of columns and their data types. Notice that the size of each column is listed to the right of the data type.
6. Close the Table Properties - Employees dialog box.
7. Right-click the Orders table, then click Properties. The Table Properties -Orders dialog box appears.
8. Review the list of columns and their data types. Close the Table Properties -Orders dialog box.
9. Open the properties for several other tables, and review the columns and data types.

To identify the data types for the Authors table

1. Make a list of each column in the Authors table.
2. Refer to the data constraints that you identified for the Authors table when you developed your database design. Which data constraints apply to the AuthorID column of the Authors table? At this point, you are concerned only with identifying the data type for the AuthorID column and determining what type of data that column will contain. In this case, you want SQL Server to generate this ID automatically, which means that when you define this column, you will need to include the IDENTITY property in the definition. The IDENTITY property can be used only with an integer or decimal data type. You will learn more about defining this type of column in the next lesson. You decide to use an integer data type rather than decimal, because decimal is unnecessary as an ID. You also decide that the smallint data type is adequate to use to identify authors. The smallint data type supports an ID of up to 32,767—many more authors than you anticipate the database ever needing to store.
3. Write down smallint next to the AuthorID column.
4. Review the database design and the data constraints for the FirstName and

LastName columns. What type of data will you store in this column? Because a name can vary in length but will not likely exceed 30 characters, you decide to use the varchar(30) data type for each column.

5. Review the database design and the data constraints for the YearBorn and YearDied columns. You can assume that each column will contain only four characters. Because date and time data types do not include a year-only data type, you decide to use a character data type. Which data type should you use for the YearBorn and YearDied columns?
6. Review the database design and the data constraints for the Description column. What type of data will you store in this column? Because the description can vary in length but will not likely exceed 200 characters, you decide to use the varchar(200) data type for each column.
7. Be sure to write down the name of the correct data type next to the name of each column in the Authors table.

To identify the column data types for tables in the BookShopDB database

1. Write down the name of each table in your database design.
2. Review the database design and the data constraints for each column in the tables.
3. Identify the data type for each column. What is the data type for each column in the BookShopDB tables?
4. Be certain to write down the data type next to the name of each column (or at least record this information in some way). You will need this information for later exercises.

Q-19 Enforce a data integrity:

(A) Entity integrity

Entity integrity defines a row as a unique instance of an entity for a particular table. Entity integrity enforces the integrity of the identifier column or the primary key of a table (through indexes, UNIQUE constraints, PRIMARY KEY constraints, or IDENTITY properties).

(B) Domain integrity

Domain integrity is the validity of entries for a given column. You can enforce domain integrity by restricting the type (through data types), the format (through CHECK constraints and rules), or the range of possible values (through FOREIGN KEY constraints, CHECK constraints, DEFAULT definitions, NOT NULL definitions, and rules).

(C) Referential integrity: -

Referential integrity preserves the defined relationships between tables when records are entered or deleted. In SQL Server, 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 non-existent 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 doing any of the following:

- Adding records to a related table if there is no associated record in the primary table
- Changing values in a primary table that result in orphaned records in a related table
- Deleting records from a primary table if there are related records in the foreign table

For example, with the Sales and Titles tables in the Pubs database, referential integrity is based on the relationship between the foreign key (title_id) in the Sales table and the primary key (title_id) in the Titles table. Figure given below.

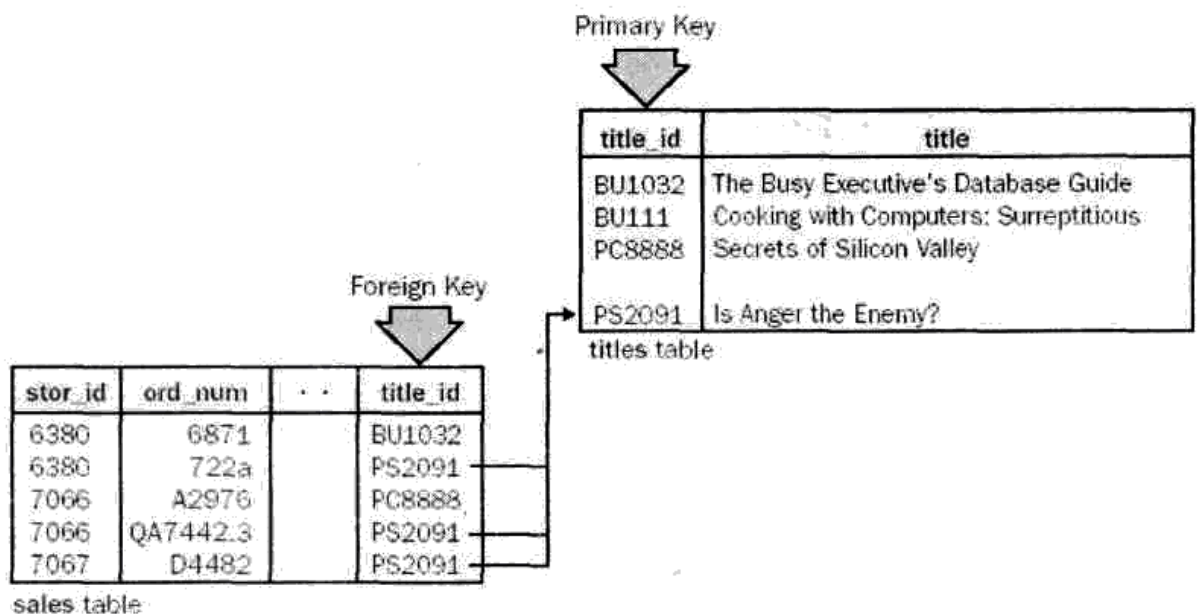


Figure 5.1. Referential integrity between the Sales table and the Titles table.

User-Defined Integrity

User-defined integrity enables you to define specific business rules that do not fall into one of the other integrity categories. All of the integrity categories support user-defined integrity (all column-level and table-level constraints in the CREATE TABLE statement, stored procedures, and triggers).

Q-20 Explain Integrity Constant ? Introduction to Integrity Constraints

Constraints enable you to define the way SQL Server 2000 automatically enforces the integrity of a database. Constraints define rules regarding the values allowed in columns and are the standard mechanisms for enforcing integrity. Using constraints is preferred to using triggers, rules, or defaults. The query optimizer also uses constraint definitions to build high-performance query execution plans.

Constraints can be column constraints or table constraints:

| A column constraint is specified as part of a column definition and applies only to that column.

| A table constraint is declared independently from a column definition and can apply to more than one column in a table. Table constraints must be used when more than one column is included in a constraint. For example, if a table has two or more columns in the primary key, you must use a table constraint to include both columns in the primary key. Consider a table that records events happening in a computer in a factory. Assume that events of several types can happen at the same time, but no two events happening at the same time can be of the same type. This rule can be enforced in the table by including both the type and time columns in a two-column primary key, as shown in the following

CREATE TABLE statement:

```
CREATE TABLE FactoryProcess
```

```
(
```

```
  EventType INT,
```

```
  EventTime DATETIME,
```

```
  EventSite CHAR(50),
```

```
  EventDesc CHAR(1024),
```

```
  CONSTRAINT event_key PRIMARY KEY (EventType, EventTime)
```

```
)
```

SQL Server supports four main classes of constraints: PRIMARY KEY constraints,

UNIQUE constraints, FOREIGN KEY constraints, and CHECK constraints.

PRIMARY KEY Constraints

A table usually has a column (or combination of columns) whose values uniquely identify each row in the table. This column (or columns) is called the primary key of the table and enforces the entity integrity of the table. You can create a primary key by defining a PRIMARY KEY constraint when you create or alter a table. A table can have only one PRIMARY KEY constraint, and a column that participates in the PRIMARY KEY constraint cannot accept null values. Because PRIMARY KEY constraints ensure unique data, they are often defined for identity columns. When you specify a PRIMARY KEY constraint for a table, SQL Server 2000 enforces data uniqueness by creating a unique index for the primary key columns. This index also permits fast access to data when the primary key is used in queries. If a PRIMARY KEY constraint is defined for more than one column, values can be duplicated within one column—but each combination of values from all of the columns in the PRIMARY KEY constraint definition must be unique. Figure 5.2 illustrates how the `au_id` and `title_id` columns in the `TitleAuthor` table form a composite PRIMARY KEY constraint, which ensures that the combination of `aujd` and `titlejd` is unique.

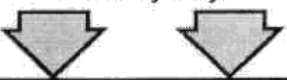
Creating PRIMARY KEY Constraints

You can create a PRIMARY KEY constraint by using one of the following methods:

| Creating the constraint when the table is created (as part of the table definition)
 | Adding the constraint to an existing table, provided that no other PRIMARY KEY constraint already exists You can modify or delete a PRIMARY CONSTRAINT once it has been created. For example, you might want the

PRIMARY KEY constraint of the table to

Primary Key



au_id	title_id	au_ord	royaltyper
172-32-1176	PS3333	1	100
213-46-8915	BU1032	2	40
213-46-8915	BU2075	1	100
238-95-7766	PC1035	1	100
267-41-2394	BU1111	2	40

Figure 5.2. The primary key of the TitleAuthor table in the Pubs database.

reference other columns, or you might want to change the column order, index name, clustered option, or fill factor of the PRIMARY KEY constraint. You cannot change the length of a column defined with a PRIMARY KEY constraint.

Note: To modify a PRIMARY KEY constraint by using Transact-SQL, you must first delete the existing PRIMARY KEY constraint and then re-create it with the new definition.

The following CREATE TABLE statement creates the Table1 table and defines the

CoM column as the primary key:

```
CREATE TABLE Table 1
(
  Coll INT PRIMARY KEY,
  Col2 VARCHAR(30)
)
```

You can also define the same constraint by using a table-level PRIMARY KEY constraint: CREATE TABLE Table1

```
(
  Coll INT,
  Col2 VARCHAR(30),
  CONSTRAINT table_pk PRIMARY KEY (Coll)
)
```

You can use the ALTER TABLE statement to add a PRIMARY KEY constraint to an existing table:

```
ALTER TABLE Table 1
ADD CONSTRAINT table_pk PRIMARY KEY (Coll)
```

When a PRIMARY KEY constraint is added to an existing column (or columns) in the table, SQL Server 2000 checks the existing data in the columns to ensure that it follows the rules for primary keys

- | No null values
- | No duplicate values

If a PRIMARY KEY constraint is added to a column that has duplicate or null values, SQL Server returns an error and does not add the constraint. You cannot add a PRIMARY KEY constraint that violates these rules.

SQL Server automatically creates a unique index to enforce the uniqueness requirement of the PRIMARY KEY constraint. If a clustered index does not already exist in the table (or a non-clustered index is not explicitly specified), a unique, clustered index is created to enforce the PRIMARY KEY constraint.

Important A PRIMARY KEY constraint cannot be deleted if it is referenced by a FOREIGN KEY constraint in another table. The FOREIGN KEY constraint must be deleted first. FOREIGN KEY constraints are discussed later in this lesson.

UNIQUE Constraints

You can use UNIQUE constraints to ensure that no duplicate values are entered in specific columns that do not participate in a primary key. Although both a UNIQUE constraint and a PRIMARY KEY constraint enforce uniqueness, you should use a UNIQUE constraint instead of a PRIMARY KEY constraint in the following situations:

| If a column (or combination of columns) is not the primary key. Multiple UNIQUE constraints can be defined on a table, whereas only one PRIMARY KEY constraint can be defined on a table.

| If a column allows null values. UNIQUE constraints can be defined for columns that allow null values, whereas PRIMARY KEY constraints can be defined only on columns that do not allow null values.

A UNIQUE constraint can also be referenced by a FOREIGN KEY constraint.

Creating UNIQUE Constraints

You can create a UNIQUE constraint in the same way that you create a PRIMARY KEY constraint:

- | By creating the constraint when the table is created (as part of the table definition)
- | By adding the constraint to an existing table, provided that the column or combination of columns comprising the UNIQUE constraint contains only unique or NULL values. A table can contain multiple UNIQUE constraints. You can use the same Transact-SQL statements to create a UNIQUE constraint that you used to create a PRIMARY KEY constraint. Simply replace the words PRIMARY KEY with the word UNIQUE. As with PRIMARY KEY constraints, a UNIQUE constraint can be modified or deleted once it has been created.

When a UNIQUE constraint is added to an existing column (or columns) in the table, SQL Server 2000 (by default) checks the existing data in the columns to ensure that all values, except null, are unique. If a UNIQUE constraint is added to a column that has duplicated values, SQL Server returns an error and does not add the

constraint.

SQL Server automatically creates a UNIQUE index to enforce the uniqueness requirement of the UNIQUE constraint. Therefore, if an attempt is made to insert a duplicate row, SQL Server returns an error message saying that the UNIQUE constraint has been violated and does not add the row to the table. Unless a clustered index is explicitly specified, a unique, non-clustered index is created by default to enforce the UNIQUE constraint.

FOREIGN KEY Constraints

A foreign key is a column or combination of columns used to establish and enforce a link between the data in two tables. Create a link between two tables by adding a column (or columns) to one of the tables and defining those columns with a FOREIGN KEY constraint. The columns will hold the primary key values from the second table. A table can contain multiple FOREIGN KEY constraints.

For example, the Titles table in the Pubs database has a link to the Publishers table because there is a logical relationship between books and publishers. The pub_id column in the Titles table matches the primary key column in the Publishers table, as shown in Figure 5.3. The pub_id column in the Titles table is the foreign key to the Publishers table.

You can create a foreign key by defining a FOREIGN KEY constraint when you create or alter a table. In addition to a PRIMARY KEY constraint, a FOREIGN KEY constraint can reference the columns of a UNIQUE constraint in another table.

A FOREIGN KEY constraint can contain null values; however, if any column of a composite FOREIGN KEY constraint contains null values, then verification of the FOREIGN KEY constraint will be skipped.

Note: A FOREIGN KEY constraint can reference columns in tables in the same database or within the same table (self-referencing tables).

Although the primary purpose of a FOREIGN KEY constraint is to control the data that can be stored in the foreign key table, it also controls changes to data in the primary key table. For example, if the row for a publisher is deleted from the Publishers table and the publisher's ID is used for books in the Titles table, the relational integrity between the two tables is broken. The deleted publisher's books are orphaned in the titles table without a link to the data in the Publishers table. A FOREIGN KEY constraint prevents this situation. The constraint enforces referential integrity by ensuring that changes cannot be made to data in the primary key table if those changes invalidate the link to data in the foreign key table. If an attempt is made to delete the row in a primary key table or to change a primary key value, the action will fail if the deleted or changed primary key value corresponds to a value in the FOREIGN KEY constraint of another table.

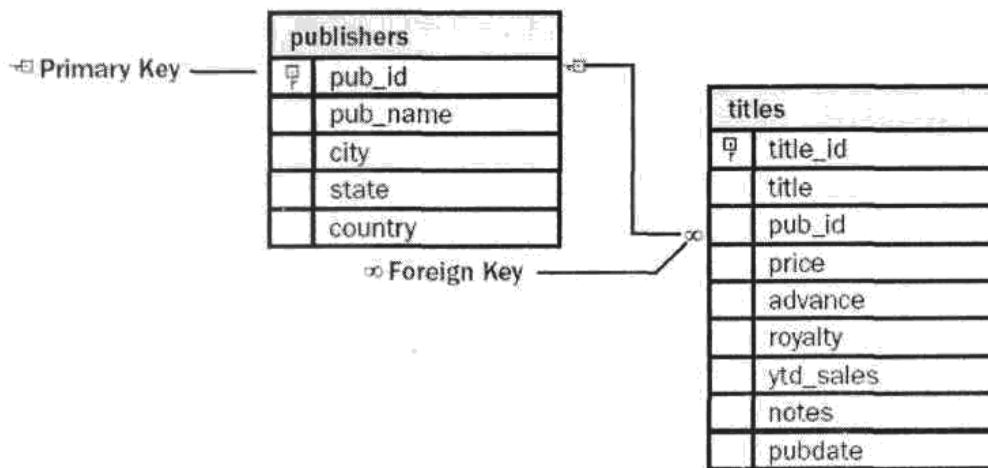


Figure 5.3. A FOREIGN KEY constraint defined in the Titles table of the Pubs database.

To successfully change or delete a row in a FOREIGN KEY constraint, you must first either delete the foreign key data in the foreign key table or change the foreign key data in the foreign key table—thereby linking the foreign key to different primary key data.

Creating FOREIGN KEY Constraints

You can create a FOREIGN KEY constraint by using one of the following methods:

- | Creating the constraint when the table is created (as part of the table definition)
 - | Adding the constraint to an existing table, provided that the FOREIGN KEY constraint is linked to an existing PRIMARY KEY constraint or a UNIQUE constraint in another (or the same) table
- You can modify or delete a FOREIGN KEY constraint once it has been created. For example, you might want the table's FOREIGN KEY constraint to reference other columns. You cannot change the length of a column defined with a FOREIGN KEY constraint.

Note: To modify a FOREIGN KEY constraint by using Transact-SQL, you must first delete the existing FOREIGN KEY constraint and then re-create it with the new definition.

The following CREATE TABLE statement creates the Table1 table and defines the Col2 column with a FOREIGN KEY constraint that references the EmployeeID column, which is the primary key in the Employees table:

```
CREATE TABLE Table 1
(
  Col1 INT PRIMARY KEY,
  Col2 INT REFERENCES Employees(EmployeeID)
)
```

You can also define the same constraint by using a table-level FOREIGN KEY constraint: CREATE TABLE Table1


```
(  
Col1 INT PRIMARY KEY,  
Col2 INT,  
CONSTRAINT col2_fk FOREIGN KEY(Col2)  
REFERENCES Employees (EmployeeID)  
)
```

You can use the ALTER TABLE statement to add a FOREIGN KEY constraint to An existing table:

```
ALTER TABLE Table 1  
ADD CONSTRAINT col2Jk FOREIGN KEY (Col2)  
REFERENCES Employees (EmployeeID)
```

When a FOREIGN KEY constraint is added to an existing column (or columns) in the table, SQL Server 2000 (by default) checks the existing data in the columns to ensure that all values, except null values, exist in the columns of the referenced PRIMARY KEY or UNIQUE constraint. You can prevent SQL Server from checking the data in the column against the new constraint, however, and force it to add the new constraint regardless of the data in the column. This option is useful when the existing data already meets the new FOREIGN KEY constraint or when a business rule requires the constraint to be enforced only from this point forward. You should be careful when adding a constraint without checking existing data, however, because this action bypasses the controls in SQL Server that enforce the data integrity of the table.

Disabling FOREIGN KEY Constraints

You can disable existing FOREIGN KEY constraints when performing the following actions:

- | Executing INSERT and UPDATE statements. Disable a FOREIGN KEY constraint during INSERT and UPDATE statements if new data will violate the constraint or if the constraint should apply only to the data already in the database. Disabling the constraint enables data in the table to be modified without being validated by the constraints

- | Implementing replication processing. Disable a FOREIGN KEY constraint during replication if the constraint is specific to the source database. When a table is replicated, the table definition and data are copied from the source database to a destination database. These two databases are usually (but not necessarily) on separate servers. If the FOREIGN KEY constraints are specific to the source database but are not disabled during replication, they might unnecessarily prevent new data from being entered in the destination database.

CHECK Constraints

CHECK constraints enforce domain integrity by limiting the values that are accepted by a column. They are similar to FOREIGN KEY constraints in that they control the values that are placed in a column. The difference is in how they determine when values are valid. FOREIGN KEY constraints get the list of valid values from

another table, and CHECK constraints determine the valid values from a logical expression that is not based on data in another column. For example, it is possible to limit the range of values for a salary column by creating a CHECK constraint that allows only data ranging from \$15,000 through \$100,000. This feature prevents the entering of salaries from outside the normal salary range. You can create a CHECK constraint with any logical (Boolean) expression that returns TRUE or FALSE based on the logical operators. To allow only data that ranges from \$15,000 through \$100,000, the logical expression is as follows:

salary >= 15000 AND salary <= 100000

You can apply multiple CHECK constraints to a single column. The constraints are evaluated in the order in which they are created. In addition, you can apply a single CHECK constraint to multiple columns by creating it at the table level. For example, a multiple-column CHECK constraint can be used to confirm that any row with a country column value of USA also has a two-character value in the state column. This feature enables multiple conditions to be checked in one place.

Creating CHECK Constraints

You can create a CHECK constraint by using one of the following methods:

| Creating the constraint when the table is created (as part of the table definition)

| Adding the constraint to an existing table

You can modify or delete CHECK constraints once they have been created. For example, you can modify the expression used by the CHECK constraint on a column in the table.

Note : To modify a CHECK constraint using Transact-SQL, you must first delete the existing CHECK constraint and then re-create it with the new definition

The following CREATE TABLE statement creates the Table1 table and defines the Col2 column with a CHECK constraint that limits the column-entered values to a range between 0 and 1000:

```
CREATE TABLE Table1
(
  Col1 INT PRIMARY KEY,
  Col2 INT
  CONSTRAINT limit_amount CHECK (Col2 BETWEEN 0 AND 1000),
  Col3 VARCHAR(30)
)
```

You can also define the same constraint by using a table-level CHECK constraint: CREATE TABLE Table1

```
(
  Col1 INT PRIMARY KEY,
  Col2 INT,
  Col3 VARCHAR(30),
  CONSTRAINT limit_amount CHECK (Col2 BETWEEN 0 AND 1000)
```

)

You can use the ALTER TABLE statement to add a CHECK constraint to an existing table:

ALTER TABLE Table1 ADD CONSTRAINT !limit_amount CHECK (Col2 BETWEEN 0 AND 1000) When a CHECK constraint is added to an existing table, the CHECK constraint can apply either to new data only or to existing data as well. By default, the CHECK constraint applies to existing data as well as to any new data. The option of applying the constraint to new data only is useful when the existing data already meets the new CHECK constraint or when a business rule requires the constraint to be enforced only from this point forward.

For example, an old constraint might require postal codes to be limited to five digits, but a new constraint might require nine-digit postal codes. Old data with fivedigit postal codes is still valid and will coexist with new data that contains ninedigit postal codes. Therefore, only new data should be checked against the new constraint.

You should be careful when adding a constraint without checking existing data, however, because this action bypasses the controls in SQL Server 2000 that enforce the integrity rules for the table.

Disabling CHECK Constraints

You can disable existing CHECK constraints when performing the following actions

| Executing INSERT and UPDATE statements. Disable a CHECK constraint during INSERT and UPDATE statements if new data will violate the constraint or if the constraint should apply only to the data already in the database. Disabling the constraint allows data in the table to be modified without being validated by the constraints.

| Implementing replication processing. Disable a CHECK constraint during replication if the constraint is specific to the source database. When a table is replicated, the table definition and data are copied from the source database to a destination database. These two databases are usually (but not necessarily) on separate servers. If the CHECK constraints specific to the source database are not disabled, they might unnecessarily prevent new data from being entered into the destination database.

Q-21 Explain Select clause The SELECT Clause

The SELECT clause includes the SELECT keyword and the select list. The select list is a series of expressions separated by commas. Each expression defines a column in the result set. The columns in the result set are in the same order as the sequence of expressions in the select list. Using Keywords

the Select List

The select list can also contain keywords that control the final format of the result set.

The DISTINCT Keyword

The DISTINCT keyword eliminates duplicate rows from a result set. For example, the Orders table in the Northwind database contains duplicate values in the ShipCity column. To get a list of the ShipCity values with duplicates removed, enter the following code; SELECT DISTINCT ShipCity, ShipRegion FROM Orders ORDER BY ShipCity

The TOP n Keyword

The TOP n keyword specifies that the first n rows of the result set are to be returned. If ORDER BY is specified, the rows are selected after the result set is ordered. The n placeholder is the number of rows to return (unless the PERCENT keyword is specified). PERCENT specifies that n is the percentage of rows in the result set that are returned. For example, the following SELECT statement returns the first 10 cities in alphabetic sequence from the Orders table: SELECT DISTINCT TOP 10 ShipCity, ShipRegion FROM Orders ORDER BY ShipCity

The AS Keyword

You can improve the readability of a SELECT statement by giving a table an alias (also known as a correlation name or range variable). A table alias can be assigned either with or without the AS keyword:

```
| table_name AS table_alias
```

```
| table_name table_alias
```

In the following example, the alias p is assigned to the Publishers table:

```
USE pubs
```

```
SELECT p.pub_id, p.pub_name
```

```
FROM publishers AS p
```

Important: If an alias is assigned to a table, all explicit references to the table in the Transact-SQL statement must use the alias, not the table name.

Types of Information in the Select List

A select list can include many types of information, such as a simple expression or a scalar subquery. The following example shows many of the items that you can include in a select list:

```
SELECT FirstName + ' ' + LastName AS "Employee Name",IDENT ID COL AS "Employee ID",HomePhone,Region FROM Northwind.dbo.Employees ORDER BY LastName, FirstName ASC
```

In this statement, the employees' first and last names are combined into one column. A space is added between the first and last names. The name of the column that will contain the employee names is Employee Name. The result set will also include the identity column, which will be named Employee ID in the result set; the HomePhone column; and the Region column. The result set is ordered first by last name and then by first name.

Q-22 Explain different type of joining with suitable example.

Joins are used to retrieve data from multiple tables. In SQL Server we have three types of joins :-

- 1) Inner Join
- 2) Outer Join
- 3) Cross Join

(1) Inner Join- is the default type of join. It produces resultset which contains only matched rows.

Example: Display EmpNo, EmpName, DeptName & Location.

```
Select EmpNo, EmpName, DName, DLoc
From Employees Inner Join Dept
On Employees.DeptNo =
Dept.DeptNo
```

Note: Tablename.Columnname It refers specific column in the table.

Tablename.* It refers all the columns in the table

(2) Outer Join- produces result set which contains matched rows as well as unmatched rows. We have three types of outer joins:-

- 1) Left Outer Join
- 2) Right Outer Join
- 3) Full Outer Join

Left outer join- Resultset contains all the rows from left table & only matched rows from right table.

Ex: Display all the departments' information & corresponding employees' information.

```
Select Dept.*, EmpName, Sal
From Dept Left outerjoin Employees
On
Dept.DeptNo = Employees.DeptNo
```

Right outer join - Resultset contains all the rows from right table & only matched rows from left table.

Ex: Display EmpNo, EmpName & all Departments names.

```
Select EmpNo, EmpName, DName
From Employees right outer join Dept
On Employees.DeptNo =
Dept.DeptNo
```

Full outer join - Resultset contains all the rows from left table & all the rows from right table.

(4) Cross Join - A join without any condition every row in first table joined with every row in second table.

Ex:

```
Select * from
Employees cross join dept
```

Cross join resultset takes more memory. It is also called as cartesian join. Self Join - Joining a table with itself is called self join. To work with self join we use alias

tables.

Alias tables & Alias columns -

- 1) Alias tables are used to refer the same table multiple times in a query.
Alias names can be accessible as long as query executed.

Ex:

```
Select columnist  
From tablename aliasname
```

- 2) Alias columns are used to change the columns heading in the output. Syntax:
Select columnname as [alias name]
From tablename

Ex:

```
Select d1.DeptNo as [DeptNumber] From Deptdi
```

Q-23 Explain bulk insert statement using BCP utility.

The BCP command prompt utility copies SQL Server data to or from a data file. You will use this utility most frequently to transfer large volumes of data into a SQL Server table from another program often from another database management system (DBMS).

When the bcp utility is used, the data is first exported from the source program to a data file and is then imported from the data file into a SQL Server table. Alternatively, bcp can be used to transfer data from a SQL Server table to a data file for use in other programs such as Microsoft Excel. Data can also be transferred into a SQL Server table from a data file by using the BULK INSERT statement. The BULK INSERT statement cannot bulk copy data from an instance of SQL Server to a data file, however. With the BULK INSERT statement, you can bulk copy data to an instance of SQL Server by using the functionality of the bcp utility in a Transact-SQL statement (rather than from the command prompt).

If you are importing data, the destination table must already exist. If you are exporting to a file, bcp will create the file. The number of fields in the data file does not have to match the number of columns in the table or be in the same order.

The data in the data file must be in character format or in a format that the bcp utility generated previously, such as native format. Each column in the table must be compatible with the field in the data file being copied. For example, it is not possible to copy an int field to a datetime column using native format bcp.

Relevant permissions to bulk copy data are required for source and destination files and tables;- To bulk copy data from a data file into a table, you must have INSERT and SELECT permissions for the table. To bulk copy a table or view to a data file, you must have SELECT permission for the table or view being bulk copied.

The following bcp command copies data from the Publishers table in the Pubs database and into the Publishers.txt file:

```
BCP pubs..publishers out publishers.txt -c -T
```

Using Data Formats

The bcp utility can create or read data files in the default data formats by specifying a switch at the command prompt. The following table includes a description

of the four default data formats:

Data Format	Bcp Switch	BULK INSERT Clause	Description
Native	-n	BULK INSERT – native	Use the native data types. Storing information in native format is useful when information must be copied from one instance of SQL Server to another. Using native format saves time and space, preventing unnecessary conversion of data types in and from character format. A data file in native format cannot be read by any program other than bcp, however.
Character	-c	DATAFILETYPE 'char'	– Uses the character (<i>char</i>) data format for all columns, providing tabs between fields and a new-line character at the <i>end</i> of each row as default terminators. Storing information in character format is useful when the data is used with another program, such as a spreadsheet, or when the data needs to be copied into an instance of SQL Server from another database. Character format tends to be used when copying data in other programs that have the functionality to export and import data in plain-text format.
		character 'widechar'	the DATAFILETYPE clause of the <i>BULK INSERT</i> statement) uses the Unicode character data format for all columns, providing (as default terminators) tabs between fields and a new-line character at the end of each row. This format allows data to be copied from <i>n</i> server (that is using a code page different from the code page used by the client running bcp) another server that uses the same or different code page as the original server. This format prevents the loss of any character data, the source and destination are not Unicode data types. In addition, only a minimum

Unicode native	-N	QATAFILETYPE 'widenative'	number of extended characters are lost if the source and destination are not Unicode data types. Uses native database data types for all non-character data and uses Unicode character data format for all character {char, m'htir. vorchar. iivn/ihar, text, and next)data
----------------	----	------------------------------	--

Q-24 Explain Transact sql server cursor with its suitable example.

Transact-SQL Server Cursors

Transact-SQL Server cursors are based on the DECLARE CURSOR statement and are used mainly in Transact-SQL scripts, stored procedures, and triggers. Transact-SQL cursors are implemented on the server and are managed by Transact-SQL statements sent from the client to the server. They are also contained in batches, stored procedures, or triggers. When working with Transact-SQL cursors, you use a set of Transact-SQL statements to declare, populate, and retrieve data (as outlined in the following steps):

1. Use a DECLARE CURSOR statement to declare the cursor. When you declare the cursor, you should specify the SELECT statement that will produce the cursor's result set
2. Use an OPEN statement to populate the cursor. This statement executes the SELECT statement embedded in the DECLARE CURSOR statement.
3. Use a FETCH statement to retrieve individual rows from the result set. Typically, a FETCH statement is executed many times (at least once for each row in the result set).
4. If appropriate, use an UPDATE or DELETE statement to modify the row. This step is optional.
5. Use a CLOSE statement to close the cursor. This process ends the active cursor operation and frees some resources (such as the cursor's result set and its lock on the current row). The cursor is still declared, so you can use an OPEN statement to reopen it.
6. Use a DEALLOCATE statement to remove the cursor reference from the current session. This process completely frees all resources allocated to the cursor (including the cursor name). After a cursor is deallocated, you must issue a DECLARE statement to rebuild the cursor.

The following set of Transact-SQL statements illustrates how to declare a cursor, populate that cursor, retrieve data from the result set, update that data, close the cursor, and deallocate the cursor:

```
/* Declares the AuthorsCursor cursor and associates the cursor with a SELECT statement. */
USE Pubs
DECLARE AuthorsCursor CURSOR FOR SELECT * FROM Authors ORDER BY Aujname
/* Populates the AuthorsCursor cursor with the result set from the SELECT statement.
```

```

*/OPEN AuthorsCursor
/* Retrieves the first row from the result set. */ FETCH NEXT FROM AuthorsCursor
/* Updates the phone number within the retrieved row. */ UPDATE Authors SET
Phone = '415 658-9932' WHERE CURRENT OF AuthorsCursor /* Closes the
AuthorsCursorcursor. */ CLOSE AuthorsCursor /* Deallocates the
AuthorsCursorcursor. */ DEALLOCATE AuthorsCursor

```

Q-25 Write a step to create a stored procedure-using wizard.

The Create Stored Procedure wizard walks you through the steps necessary to create a new stored procedure. You can access the wizard by selecting Wizards from the Tools menu. In the Select Wizard window, expand the Database option, then select the Create Stored Procedure Wizard and click OK. From there, you complete the steps in the wizard. Figure 8.2 shows the options on the Welcome to the Create Stored Procedure wizard screen that you specify when you run the Create Stored Procedure wizard.

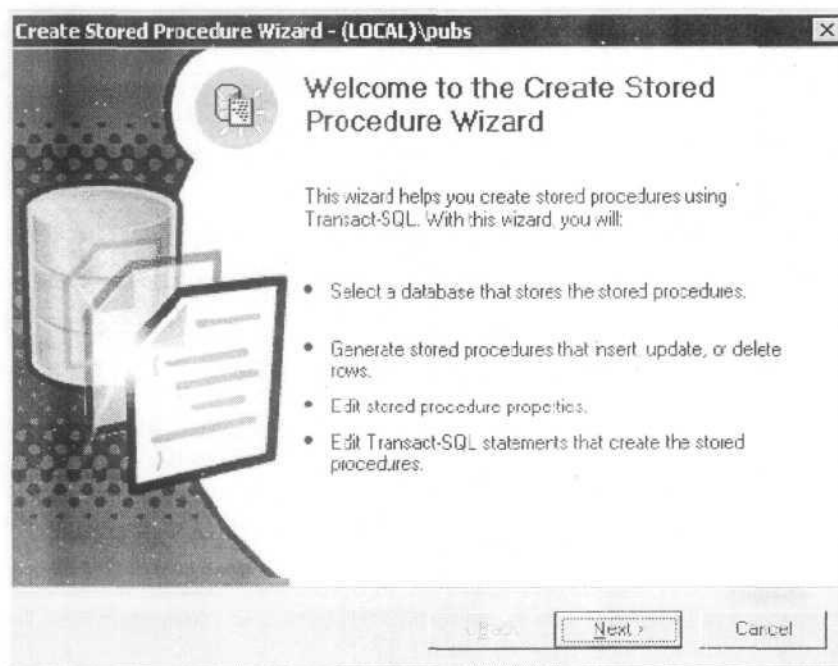


Figure 8.2. The Welcome to the Create Stored Procedure Wizard screen.

Creating and Adding Extended Stored Procedures

After creating an extended stored procedure, you must register it with SQL Server. Only users who have the sysadmin role can register an extended stored procedure with SQL Server. To register the extended stored procedure, you can use the sp_addextendedproc system stored procedure in Query Analyzer or use Enterprise Manager. In Enterprise Manager, expand the Master database, right-click the Extended Stored Procedures node, and then click New Extended Stored Procedure. Extended stored procedures can be added only to the Master database.

Deferred Name Resolution

When a stored procedure is created, SQL Server does not check for the

existence of any objects that are referenced in it. This feature exists because it's possible that an object, such as a table referenced in the stored procedure, does not exist when the stored procedure is created. This feature is called deferred name resolution. Object verification occurs when the stored procedure is executed. When referring to an object (such as a table) in a stored procedure, make sure to specify the owner of the object. By default, SQL Server assumes that the creator of the stored procedure is also the owner of the object referenced in the procedure. To avoid confusion, consider specifying dbo as the owner when creating all objects (both stored procedures and objects referenced in the stored procedures).

Executing a Stored Procedure

As you have seen in previous lessons, you can run a stored procedure in Query Analyzer simply by typing its name and any required parameter values. For example, you viewed the contents of a stored procedure by typing sp_helptext and the name of the stored procedure to be viewed. The name of the stored procedure to be viewed is the parameter value.

If the stored procedure isn't the first statement in a batch, in order to run it you must precede the name of the stored procedure with the EXECUTE keyword or with the shortened version of the keyword, EXEC.

Calling a Stored Procedure for Execution

When you specify the procedure name, the name can be fully qualified, such as [database_name].[owner].[procedure_name]. Or, if you make the database containing the stored procedure the current database (USE database_name), then you can execute the procedure by specifying [owner].[procedure_name]. If the procedure name is unique in the active database, you can simply specify [procedure_name].

owner other than dbo. SQL Server does not automatically search the Master database for extended stored procedures. Therefore, either fully qualify the name of an extended stored procedure or change the active database to the location of the extended stored procedure.

Specifying Parameters and Their Values

If a stored procedure requires parameter values, you must specify them when executing the procedure. When input and output parameters are defined, they are preceded by the "at" sign (@), followed by the parameter name and the data type designation. When they are called for execution, you must include a value for the parameter (and optionally, the parameter name). The next two examples run the au_info stored procedure in the Pubs database with two parameters:

@lastname and @firstname:

--call the stored procedure with the parameter values.

USE Pubs

GO

EXECUTE aujinfo Green, Marjorie --call the stored procedure with parameter names and values.

```
USE Pubs
GO
EXECUTE aunjfo
@lastname = 'Green', @firstname = 'Marjorie'
```

Q-27 How to Create Trigger Using Transact SQL Statement ?

Creating Triggers Using Transact-SQL

You can use the CREATE TRIGGER statement to create a trigger by using Query Analyzer or a command-prompt tool such as osql. When using CREATE TRIGGER, you must specify the trigger name, the table or view upon which the trigger is applied, the class of trigger (INSTEAD OF or AFTER), the event or events that fire the trigger, and the task that you wish the trigger to perform. Optionally, you can specify whether the trigger should be replicated or encrypted. The WITH APPEND clause remains for backward compatibility but shouldn't be used to create triggers for a SQL Server 2000 database.

The main clauses in a CREATE TRIGGER statement can be summarized as follows:

```
CREATE TRIGGER trigger_name
ON tablejname or viewname
FOR trigger_class and triggerjype(s)
AS Transact-SQL statements
```

This section discusses the CREATE TRIGGER, ON, and FOR /AFTER/INSTEAD OF clauses in detail and provides examples of how they are used in a trigger statement.

Lesson 3 discusses the Transact-SQL statements appearing after the AS clause. For more details about trigger clauses not shown here, refer to SQL Server Books Online.

The CREATE TRIGGER Clause

Trigger creation begins with the CREATE TRIGGER clause followed by a trigger name. Triggers do not allow specifying the database name as a prefix to the object name. Therefore, select the database with the USE database name clause and the GO keyword before creating a trigger. GO is specified because CREATE TRIGGER must be the first statement in a query batch.

Permission to create triggers defaults to the table owner. For consistency, consider creating tables, triggers, and other database objects so that dbo is the owner. For example, to create a trigger named Alerter in the BookShopDB database, you can use the following Transact-SQL code:

```
USE BookShopDB
GO
CREATE TRIGGER dbo.alerter
```

Trigger names must follow the rules for identifiers. For example, if you decide to create a trigger named Alerter for the Employees Table, you must enclose the name in brackets as shown:

```
CREATE TRIGGER dbo.[alerter for employees table] Administering the trigger
```


object, such as deleting it, also requires that you follow the rules for identifiers.

The ON Clause

Triggers must be assigned to a table or view. Use the ON clause to instruct the trigger on to what table or view it should be applied. When a trigger is applied, the table or view is referred to as the trigger table or the trigger view. For consistency, specify the table or view owner after the ON clause. For example, to apply a trigger to the Employees table named Alerter (where both objects—the table and the trigger— are owned by dbo), you can use the following Transact-SQL code:

CREATE TRIGGER dbo.alerter

ON dbo .employees

A trigger is applied only to a single table or view. If you need to apply the same trigger task to another table in the database, create a trigger of a different name that contains the same business logic. Then, apply the new trigger to the other table. The default trigger class, AFTER, can be applied only to a table. The new trigger class, INSTEAD OF, can be applied to either a table or a view.

The FOR, AFTER, and INSTEAD OF Clauses

A trigger event type must be specified when the trigger is created. Valid event types include INSERT, UPDATE, and DELETE. A single trigger can be fired because of one, two, or all three of the events occurring. If you want a trigger to fire on all events, follow the FOR, AFTER, or INSTEAD OF clause with INSERT, UPDATE, and DELETE. The event types can be listed in any order. For example, to make the trigger named Alerter fire on all events, you can use the following

Transact-SQL code: CREATE

TRIGGER dbo.alerterON

dbo.employees

FOR INSERT, UPDATE, DELETE The FOR clause is synonymous with the AFTER clause. Therefore, the previous code example creates an AFTER trigger. To create Alerter as an INSTEAD OF trigger, you can use the following Transact-SQL code:

CREATE TRIGGER dbo.alerter

ON d bo .employees

INSTEAD OF INSERT, UPDATE, DELETE Notice that the FOR clause is replaced with INSTEAD OF.

The AS Clause

The AS clause and the Transact-SQL language following it designates the task that the trigger will perform. The following example shows how to create an Alerter trigger that sends an e-mail to a user named BarryT when an INSERT, UPDATE, orDELETE occurs on the employees table:

USE BookShopDB

GO

CREATE TRIGGER dbo.alerter

```

ON dbo.employees
AFTER INSERT, UPDATE, DELETE
AS EXEC master..xp_sendmail
'BarryT,
'A record was just inserted, updated or deleted in the Employees table.'
GO

```

This example is kept simple so that you can see clearly how a task is created in a trigger. There are a number of ways to make the task more useful. For example, you could write the task so that the e-mail message details the exact change that occurred. Lesson 3 explores more complex trigger tasks.

Q-28 Explain Pseudo table with trigger for insert & Delete statement.

When an INSERT, UPDATE, or DELETE trigger fires, the event creates one or more pseudo tables (also known as logical tables). These logical tables can be thought of as the transaction logs of the event. There are two types of logical tables: the Inserted table and the Deleted table. An insert or update event creates an Inserted logical table. The Inserted logical table contains the record set that has been added or changed. The UPDATE trigger also creates a Deleted logical table. The Deleted logical table contains the original record set as it appeared before the update. The following example creates a trigger that displays the contents of the Inserted and Deleted tables following an update event to the Authors table:

```

CREATE TRIGGER dbo.updatetables
ON d bo .authors
AFTER UPDATE
AS
SELECT "Description" = The Inserted table:'
SELECT * FROM inserted
SELECT "Description" = The Deleted table:'
SELECT* FROM deleted

```

Following a simple UPDATE statement that changes an author's name from Dean to Denby, the trigger displays the following results:

The Inserted table:

Straight Denby Oakland CA 94609

The Deleted table:

Straight Dean Oakland CA 94609

The Authors table (trigger table) contains the updated record after the update trigger runs. When the trigger fires, the update to the Authors table can be rolled back by programming bgic into the trigger. This transaction rollback capability also applies to INSERT and DELETE triggers.

Q-29 explain view and its function Overview of Views:

A view acts as a filter on the underlying tables referenced in the view. The query that defines the view can be from one or more tables or from other views in the

current database or other databases. You can also use dBTributed queries to define views that use data from multiple heterogeneous sources. This functionality is useful, for example, if you want to combine similarly structured data from different servers, each of which stores data for a different region of your organization.

A view can be thought of as either a virtual table or a stored query. The data accessible through a standard 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 in the same way a table is referenced.

Figure 10.1 shows a view based on a SELECT statement that retrieves data from the Titles table and the Publishers table in the Pubs database.

There are no restrictions on querying through views and few restrictions on modifying data through them. In addition, a view can reference another view. You can use a view to perform any or a II of the following functions:

- | Restricting a user to specific rows in a table
- | Restricting a user to specific columns
- | Joining columns from multiple tables so that they boklike a single table
- | Aggregating information instead of supplying details

Views can be used to partition data across multiple databases or instances of SQL Server 2000. Partitioned views enable you to distribute database processing across a group of serve is.

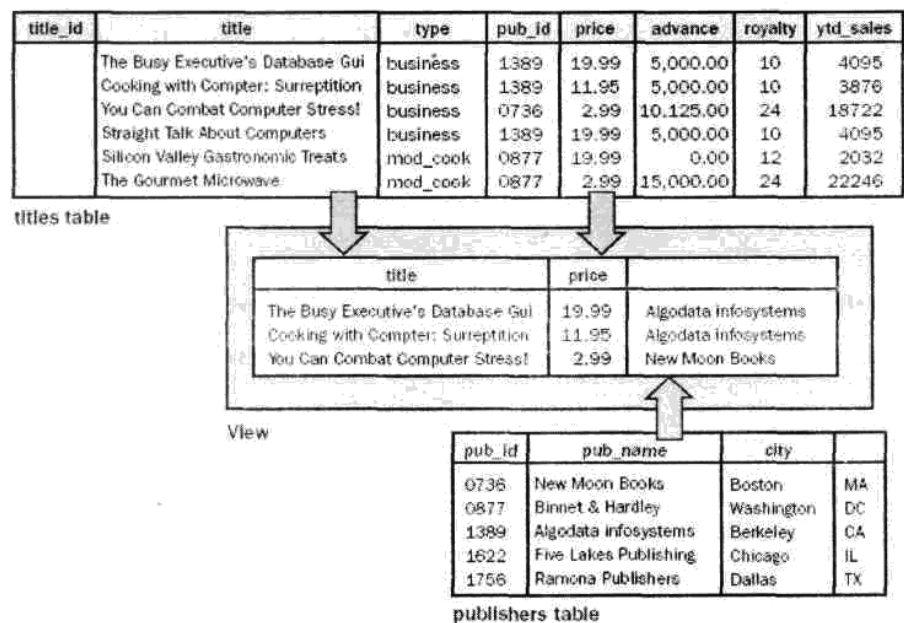


Figure 10.1. A view based on data from two tables.

SQL Server 2000 also supports indexed views, which greatly improve the performance of complex views such as those usually found in data warehouses or other decision support systems. With a standard view, the result set is not saved in the database. Rather, it is dynamically incorporated into the logic of the statement and is

built dynamically at run time.

Complex queries, however, such as those in decision support systems, can reference large numbers of rows in base tables and aggregate large amount of information into relatively concise aggregates (such as sums or averages). SQLServer

2000 supports creating a clustered index on a view that implements such a complex query. When the CREATE INDEX statement is executed, the result set of the view SELECT is stored permanently in the database. Future SQL statements that reference the view will have substantially better response times. Modifications to the base data are automatically reflected in the view.

Q-31 Give a example of clustered index view

Creating a clustered index on a view stores the data as it exists at the time the index is created. An indexed view also automatically reflects modifications made to the data in the base tables after the index is created, the same way as an index created on a base table. As modifications are made to the data in the base tables, they are also reflected in the data stored in the indexed view. Because the view's clustered index must be unique, SQL Server can more efficiently find the index rows affected by any data modification.

Indexed views can be more complex to maintain than indexes on base tables. You should create an index on a view only if the improved speed in retrieving the result outweighs the increased overhead of making modifications. This improvement usually zoccurs for views that are mapped over relatively static data, process many rows, and are referenced by many queries.

A view must meet the following requirements before you can create a clustered index on it:

- | The ANSI_NULLS and QUOTED_IDENTIFIER options must have been set to ON when the CREATE VIEW statement was executed. The OBJECTPROPERTY function reports this setting for views through the ExeclsAnsiNullsOn or ExeclsQuotedIdentOn properties.
- | The ANSIJMULLS option must have been set to ON for the execution of all CREATE TABLE statements that create tables referenced by the view.
- | The view must not reference any other views, only base tables.
- | All base tables referenced by the view must be in the same database as the view and must have the same owner as the view.
- | The view must be created with the SCHEMABNDING optbn. SCHEMABINDING binds the view to the schema of the underlying base tables.
- | User-defined functions referenced in the view must have been created with the SCHEMABINDING option.
- | Tables and user-defined functions must be referenced by two-part names. Onepart, three-part, and four-part names are not allowed.
- | All functions referenced by expressions in the view must be deterministic. The IsDeterministic property of the OBJECTPROPERTY function reports whether a user-defined function is deterministic.
- | If GROUP BY is not specified, the view select list cannot contain aggregate expressions.

- | If GROUP BY is specified, the view select list must contain a COUNT_BIG(*) expression, and the view definition cannot specify HAVING, CUBE, or ROLLUP.
- | A column resulting from an expression that either evaluates to a float value or uses float expressions for its evaluation cannot be a key of an index in an indexed view or table.

In addition to the previous restrictions, the SELECT statement in the view cannot contain the following Transact-SQL syntax elements:

- | The asterisk (*) syntax to specify columns in a select list
- | A table column name used as a simple expression that is specified in more than one view column
- | A derived table
- | Rowset functions
- | A UNION operator
- | Subqueries
- | Outer or self joins
- | The TOP clause
- | The ORDER BY clause
- | The DISTINCT keyword
- | COUNT(*) (COUNT_BIG(*) is allowed)
- | The AVG, MAX, MIN, STDEV, STDEVP, VAR, or VARP aggregate functions
- | A SUM function that references a nullable expression
- | The full-text predicates CONTAINS or FREETEXT
- | The COMPUTE or COMPUTE BY clauses

Q-30 Write a note on scenario for using a view. Scenarios for Using Views

You can use views in a variety of ways to return data.

To Focus on Specific Data

Views enable users to focus on specific data that interests them and on the specific tasks for which they are responsible. You can leave out unnecessary data in the view. This action also increases the security of the data, because users can see only the data that is defined in the view and not the data in the underlying table.

To Simplify Data Manipulation

Views can simplify how users manipulate data. You can define frequently used joins, UNION queries, and SELECT queries as views so that users do not have to specify all of the conditions and qualifications each time an additional operation is performed on that data. For example, a complex query that is used for reporting purposes and that performs subqueries, outer joins, and aggregation to retrieve data from a group of tables can be created as a view. The view simplifies access to the data because the underlying query does not have to be written or submitted each time the report is generated. The view is queried instead. You can also create inline user-defined functions that operate logically as parameterized views (views that have

parameters in WHERE-clause search conditions).

To Customize Data

Views enable different users to see data in different ways, even when they are using the same data concurrently. This feature is particularly advantageous when users who have many different interests and skill levels share the same database. For example, a view can be created that retrieves only the data for the customers with whom an account manager deals. The view can determine which data to retrieve based on the login ID of the account manager who uses the view.

To Export and Import Data

You can use views to export data to other applications. For example, you might want to use the Stores and Sales tables in the Pubs database to analyze sales data in Microsoft Excel. To perform this task, you can create a view based on the Stores and Sales tables. You can then use the bicep utility to export the data defined by the view. Data can also be imported into certain views from data files by using the bops utility or the BULK INSERT statement, providing that rows can be inserted into the view by using the INSERT statement

To Combine Partitioned Data

The Transact-SQL UNION set operator can be used within a view to combine the result of two or more queries from separate tables into a single result set. This display appears to the user as a single table (called a partitioned view). For example, if one table contains sales data for Washington and another table contains sales data for California, a view could be created from the UNION of those tables. The view represents the sales data for both regions. To use partitioned views, create several identical tables, specifying a constraint to determine the range of data that can be added to each table. The view is then created by using these base tables. When the view is queried, SQL Server automatically determines which tables are affected by the query and references only those tables. For example, if a query specifies that only sales data for the state of Washington is required, SQL Server reads only the table containing the Washington sales data, no other tables are accessed. Partitioned views can be based on data from multiple heterogeneous sources (such as remote servers), not just from tables in the same database. For example, to combine data from different remote servers (each of stores data for a different region of your organization), you can create distributed queries that retrieve data from each data source, and you can then create a view based on those distributed queries. Any queries read only data from the tables on the remote servers that contain the data requested by the query. The other servers referenced by the distributed queries in the view are not accessed. When you partition data across multiple tables or multiple servers, queries accessing only a fraction of the data can run faster because there is less data to scan. If the tables are located on different servers or on a computer that has multiple processors, each table involved in the query can also be scanned in parallel, thereby improving query performance. Additionally, maintenance tasks (such as rebuilding indexes or backing up a table) can be executed more quickly.

By using a partitioned view, the data still appears as a single table and can be

queried as such without having to reference the correct underlying table manually. Partitioned views are updateable if either of these conditions is met: An INSTEAD OF trigger is defined on the view with logic to support INSERT, UPDATE, and DELETE statements, j Both the view and the INSERT, UPDATE, and DELETE statements follow the rules defined for updateable, partitioned views.

Q-31: Explain Types of Index ?

Index Types

There are two types of indexes: clustered and non clustered. Both types of indexes are structured as B-trees. A clustered index contains table records in the leaf level of the B-tree. A non clustered index contains a bookmark to the table records in the leaf level. If a clustered index exists on a table, a non clustered index uses it to facilitate data lookup. In most cases, you will create a clustered index on a table before you create non clustered indexes.

Clustered Indexes

There can be only one clustered index on a table or view, because the clustered index key physically sorts the table or view. This type of index is particularly efficient for queries, because data records—also known as data pages—are stored in the leaf level of the B-tree. The sort order and storage location of a clustered index is analogous to a dictionary in that the words in a dictionary are sorted alphabetically and definitions appear next to the words.

When you create a primary key constraint in a table that does not contain a clustered index, SQL Server will use the primary key column for the clustered index key. If a clustered index already exists in a table, a nonclustered index is created on the column defined with a primary key constraint. A column defined as the PRIMARY key is a useful index because the column values are guaranteed to be unique. Unique values create smaller B-trees than redundant values and thus make more efficient lookup structures.

Note: A column defined with a unique constraint creates a nonclustered index automatically.

To force the type of index to be created for a column or columns, you can specify the CLUSTERED or NONCLUSTERED clause in the CREATE TABLE, ALTER TABLE, or CREATE INDEX statements. Suppose that you create a Persons table containing the following columns: PersonID, FirstName, LastName, and SocialSecurityNumber. The PersonID column is defined as a primary key constraint, and the SocialSecurityNumber column is defined as a unique constraint. To make the SocialSecurityNumber column a clustered index and the PersonID column a nonclustered index, create the table by using the following syntax:

```
CREATE TABLE dbo.Persons
(
person id smallint PRIMARY KEY NONCLUSTERED,
firstname varchar(30),
lastname varchar(40), socialsecuritynumber char(11)
```

UNIQUE CLUSTERED

)

Indexes are not limited to constraints. You create indexes on any column or combination of columns in a table or view. Clustered indexes enforce uniqueness internally. Therefore, if you create a non unique, clustered index on a column that contains redundant values, SQL Server creates a unique value on the redundant columns to serve as a secondary sort key. To avoid the additional work required to maintain unique values on redundant rows, favor clustered indexes for columns defined with primary key constraints.

Non clustered Indexes

On a table or view, you can create 250 non clustered indexes or 249 non clustered indexes and one clustered index. You must first create a unique clustered index on a view before you can create non clustered indexes. This restriction does not apply to tables, however. A non clustered index is analogous to an index in the back of a book. You can use a book's index to locate pages that match an index entry. The database uses a nonclustered index to locate matching records in the database. If a clustered index does not exist on a table, the table is unsorted and is called a heap. A nonclustered index created on a heap contains pointers to table rows. Each entry in an index page contains a row ID (RID). The RID is a pointer to a table row in a heap, and it consists of a page number, a file number, and a slot number. If a clustered index exists on a table, the index pages of a nonclustered index contain clustered index keys rather than RIDs. An index pointer, whether it is a RID or an index key, is called a bookmark.

Q-32 Explain Index Characteristics ?

Index Characteristics

A number of characteristics (aside from the index type, which is clustered or nonclustered) can be applied to an index. An index can be defined as follows:

- | Unique duplicate records are not allowed
- | A composite of columns—an index key made up of multiple columns
- | With a fill factor to allow index pages to grow when necessary
- | With a pad index to change the space allocated to intermediate levels of the B-tree
- | With a sort order to specify ascending or descending index keys

Note: Additional characteristics, such as file groups for index storage, can be applied to an index. Refer to CREATE INDEX in SQL Server Books Online and to Lesson 2 for more information.

Indexes are applied to one or more columns in a table or view. With some limitations, you can specify indexes on computed columns.

Q-33 Write Note on Transect Log Architecture ?

Transaction Log Architecture

Every SQL Server database has a transaction log that records all transactions and the database modifications made by each transaction. This record of transactions and their modifications supports three operations:

- | Recovery of individual transactions. If an application issues a ROLLBACK statement or if SQL Server detects an error (such as the loss of communication with a client), the log records are used to roll back any modifications made during an incomplete transaction.
- | Recovery of all incomplete transactions when SQL Server is started. If a server running SQL Server fails, the databases might be left in a state where some modifications were never written from the buffer cache to the data files, and there might be some modifications from incomplete transactions in the data files. When a copy of SQL Server is started, it runs a recovery of each database. Every modification

recorded in the log that was not written to the data files is rolled forward. Every incomplete transaction found in the transaction log is then rolled back to ensure that the integrity of the database is preserved.

| Rolling a restored database forward to the point of failure. After the loss of a database, as is possible if a hard drive fails on a server that does not have a Redundant Array of Independent Disks (RAID), you can restore the database to the point of failure. You first restore the last full or differential database backup and then restore the sequence of transaction log backups to the point of failure. As you restore each log backup, SQL Server reapplies all of the modifications recorded in the log to roll forward all of the transactions. When the last log backup is restored, SQL Server then uses the log information to roll back all transactions that were not complete at that point.

The transaction log is not implemented as a table but as a separate file or set of files in the database. The log cache is managed separately from the buffer cache for data pages, resulting in simple, fast, and robust code within the database engine. The format of log records and pages is not constrained to follow the format of data pages. You can implement the transaction log on several files. You can also define the files to autogrow as required, which reduces the potential of running out of space in the transaction log and reduces administrative overhead. The mechanism for truncating unused parts of the log is quick and has a minimal effect on transaction throughput.

Q-34 explain types of transaction

SQL Server supports three types of transactions: explicit, autocommit, and implicit.

Explicit Transactions

An explicit transaction is one in which you explicitly define both the start and the end of the transaction. Explicit transactions were also called user-defined or userspecified transactions in earlier versions of SQL Server. DB-Library applications and Transact-SQL scripts use the BEGIN TRANSACTION, COMMIT TRANSACTION, COMMIT WORK, ROLLBACK TRANSACTION, or ROLLBACK WORK Transact-SQL statements to define explicit transactions: BEGIN TRANSACTION. Marks the starting point of an explicit transaction for a connection. COMMIT TRANSACTION or COMMIT WORK. Used to end a transaction successfully if no errors were encountered. All data modifications made in the transaction become a permanent part of the database. Resources held by the transaction are freed. ROLLBACK TRANSACTION or ROLLBACK WORK. Used to erase a transaction in which errors are encountered. All data modified by the transaction is returned to the state it was in at the start of the transaction. Resources held by the transaction are freed. In the following transaction, the ROLLBACK TRANSACTION statement rolls back any changes made by the UPDATE statement:

```
BEGIN TRANSACTION GO USE Northwind
GO
UPDATE Cus tome is
SET ContactName = 'Hanna Moos'
```

```
WHERE CustomerID = BLAUS'  
GO  
ROLLBACK TRANSACTION  
GO
```

If a COMMIT TRANSACTION statement had been used in this example, rather than a ROLLBACK TRANSACTION statement, the update would have been made to the database. You can also use explicit transactions in the OLE DB, ADO, and ODBC APIs. For more information about using explicit transactions with these APIs, refer to SQL Server Books Online. Explicit transaction mode lasts only for the duration of the transaction. When the transaction ends, the connection returns to the transaction mode that it was in before the explicit transaction was started (either implicit or auto commit mode).

Autocommit Transactions

Autocommit mode is the default transaction management mode of SQL Server. Every Transact-SQL statement is committed or rolled back when it is completed. If a statement completes successfully, it is committed; if it encounters any error, it is rolled back. A SQL Server connection operates in auto commit mode whenever this default mode has not been overridden by either explicit or implicit transactions. Auto commit mode is also the default mode for ADO, OLE DB, ODBC, and DBLibrary. A SQL Server connection operates in autocommit mode until a BEGIN TRANSACTION statement starts an explicit transaction or until implicit transaction mode is set to ON. When the explicit transaction is committed or rolled back or when implicit transaction mode is turned off, SQL Server returns to autocommit mode.

Implicit Transactions

When a connection is operating in implicit transaction mode, SQL Server automatically starts a new transaction after the current transaction is committed or rolled back. You do nothing to delineate the start of a transaction; you only commit or roll back each transaction. Implicit transaction mode generates a continuous chain of transactions. After implicit transaction mode has been set to ON for a connection, SQL Server automatically starts a transaction when it first executes any of the following statements: The transaction remains in effect until you issue a COMMIT or ROLLBACK statement. After the first transaction is committed or rolled back, SQL Server automatically starts a new transaction the next time any of these statements is executed by the connection. SQL Server keeps generating a chain of implicit transactions until implicit transaction mode is turned off. Implicit transaction mode is set either by using the Transact-SQL SET statement or by using database API functions and methods.

```
ALTER TABLE INSERT  
CREATE OPEN  
DELETE REVOKE  
DROP SELECT  
FETCH TRUNCATE TABLE  
GRANT UPDATE
```


Transact-SQL Implicit Transactions

DB-Library applications and Transact-SQL scripts can use the Transact-SQL SET IMPLICIT_TRANSACTIONS ON statement to start implicit transaction mode. You should use the SET IMPLICIT_TRANSACTIONS OFF statement at the end of the batch to turn implicit transaction mode off. Use the COMMIT TRANSACTION, COMMIT WORK, ROLLBACK TRANSACTION, or ROLLBACK WORK statements to end each transaction. The following statement first creates the ImplicitTran table, then starts implicit transaction mode, then runs two transactions, and then turns off implicit transaction mode:

```
USE Pubs
GO
CREATE TABLE ImplicitTran
(
  Cola INT PRIMARY KEY,
  Colb CHAR(3) NOT NULL
)
GO
SET IMPLICIT_TRANSACTIONS ON
GO
/* First implicit transaction started
by an INSERT statement */
INSERT INTO ImplicitTran
VALUES (1, 'aaa')
GO
INSERT INTO ImplicitTran
VALUES (2, 'bbb')
GO
/* Commit first transaction */
COMMIT TRANSACTION
GO
/* Second implicit transaction started
by an INSERT statement */
INSERT INTO ImplicitTran VALUES (3, 'ccc')
GO
SELECT *
FROM ImplicitTran
GO
/* Commit second transaction */
COMMIT TRANSACTION
GO
SET IMPLICIT_TRANSACTIONS OFF
GO
```

API Implicit Transactions

You can use the ODBC and OLE DB APIs to set implicit transactions. Refer to

SQL Server Books Online for more information. ADO does not support implicit transactions. ADO applications use either autocommit mode or explicit transactions.

Distributed Transactions

Distributed transactions span two or more servers known as resource managers. The management of the transaction must be coordinated among the resource managers by a server component called a transaction manager. SQL Server can operate as a resource manager in distributed transactions coordinated by transaction managers such as the Microsoft Distributed Transaction Coordinator (MS DTC), or by other transaction managers that support the X/Open XA specification for Distributed Transaction Processing. A transaction within a single SQL Server that spans two or more databases is actually a distributed transaction. SQL Server, however, manages the distributed transaction internally. To the user, it operates as a local transaction. At the application, a distributed transaction is managed in much the same way as a local transaction. At the end of the transaction, the application requests the transaction to be either committed or rolled back. A distributed commit must be managed differently by the transaction manager to minimize the risk that a network failure might result in some resource managers successfully committing while others are rolling back the transaction. You can achieve this goal by managing the commit process in two phases: Prepare phase. When the transaction manager receives a commit request, it sends a prepare command to all of the resource managers involved in the transaction. Each resource manager then does everything required to make the transaction durable, and all buffers holding bg images for the transaction are flushed to disk. As each resource manager completes the prepare phase, it returns success or failure of the prepare phase to the transaction manager. Commit phase. If the transaction manager receives successful prepares from all of the resource managers, it sends commit commands to each resource manager. The resource managers can then complete the commit. If all of the resource managers report a successful commit the transaction manager then sends a success notification to the application. If any resource manager reports a failure to prepare, the transaction manager sends a ROLLBACK command to each resource manager and indicates the failure of the commit to the application. SQL Server applications can manage distributed transactions either through Transact-SQL or through the database API.

Transact-SQL Distributed Transactions

The distributed transactions started in Transact-SQL have a relatively simple structure:

1. A Transact-SQL script or application connection executes a Transact-SQL statement that starts a distributed transaction.
2. The SQL Server instance executing the statement becomes the controlling server in the transaction.
3. The script or application then executes either distributed queries against linked servers or remote stored procedures against remote servers.
4. As distributed queries and remote procedure calls are made, the controlling server automatically calls MS DTC to enlist the linked and remote servers in the distributed transaction.
5. When the script or application issues either a COMMIT or ROLLBACK statement, the controlling SQL Server calls MS DTC to manage the two-phase commit process or to notify the linked

and remote servers to roll back their

MS DTC Distributed Transactions

Applications written by using OLE DB, ODBC, ADO, or DB-Library can use Transact-SQL distributed transactions by issuing Transact-SQL statements to start and stop Transact-SQL distributed transactions. OLE DB and ODBC, however, also contain support at the API level for managing distributed transactions. OLE DB and ODBC applications can use these API functions to manage distributed transactions that include other COM resource managers that support MS DTC transactions other than SQL Server. They can also use the API functions to gain more control over the boundaries of a distributed transaction that includes several SQL Servers. The distributed transactions started in Transact-SQL have a relatively simple structure. The Transact-SQL statements controlling the distributed transactions are few because SQL Server and MS DTC do most of the work internally.

Q-35 What is deadlock ? How to minimize it using SQL Server?

Minimizing Deadlocks

Although deadlocks cannot be avoided completely, the number of deadlocks can be minimized. Minimizing deadlocks can increase transaction throughput and reduce system overhead because fewer transactions are rolled back, undoing all of the work performed by the transaction. In addition, fewer transactions are resubmitted by applications because they were rolled back when they were deadlocked.

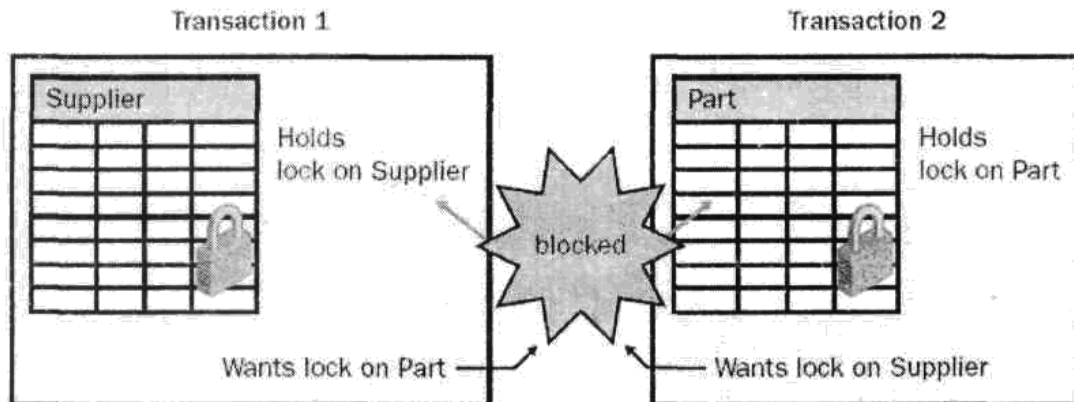


Figure 12.1. A deadlock on two transactions accessing the Supplier table and the Part table.

You should adhere to the following guidelines to help minimize deadlocks:

- | Access objects in the same order.
- | Avoid user interaction during transactions.
- | Keep transactions short and in one batch.
- | Use a low isolation level.
- | Use bound connections.

Q-36 Explain requirement for database security plan.

Requirements

In Chapter 3, "Designing a SQL Server Database," you learned that a security plan identifies database users, the data objects that they can access, and activities that they can perform in the database. Gather this information by extracting security requirements from system requirements and by determining any other security requirements that might not be part of the system requirements. For example, system requirements might not include security administrative activities such as auditing the execution of a stored procedure or running a database backup. After listing the data objects that need protection and the restricted activities that users perform in the database, create a list of unique users or classes of users and groups that access the database. Armed with this information, create a User-to-Activity Map. The map is a table that cross-references the two lists to identify which users can access protected data objects and which restricted activities they can perform in the database.

Suppose you have an employee database that is accessed by 100 users. The database contains an employee information table (Employees), a salary grade table (Salaries), and an employee location table (Locations). All employees access the Employees table and the office location information in the Locations table. A small group of employees can access the Salaries table and the home address information in the Locations table. The same small group of employees can add, delete, and modify data in all of these tables. Another user is responsible for performing nightly backups, and another group of users can create and delete data objects in the database. The security requirements for this simple database example are as follows:

- | All employees run SELECT statements against the Employees table.
- | All employees run SELECT statements on the office location information in the Locations table.
- | A small group of employees runs INSERT, DELETE, and UPDATE statements against all columns in all three tables.
- | A user runs nightly database backups and performs general database administration and requires full server access.
- | A group of users runs CREATE and DROP statements in the database. The list of unique users, classes of users, and groups that access this database is as follows:
- | All employees are a class of users covered by the Public database role.
- | Members of the Human Resources Windows 2000 group require restricted access to the database.
- | User account JDoe is a database administrator.
- | Company database developers create and delete objects in SQL Server. The following User-to-Activity Map is constructed from the information provided:

Q-37 Explain steps for index tuning using wizard ?

Running the Index Tuning Wizard

You can start the Index Tuning wizard from Enterprise Manager, Query Analyzer, or SQL Profiler. In Enterprise Manager, the Index Tuning wizard is a listed wizard in the Select wizard window. In Query Analyzer, the Index Tuning wizard is an option in the Query menu, and in SQL Profiler it's an option in the Tools menu.

When you start the Index Tuning wizard, an introductory screen appears, as Shown in Figure 14.3.



Figure 14.3. The introductory screen of the Index Tuning wizard.

After connecting to the server, the Index Tuning wizard requires that you select a database and specify whether you want to keep the existing indexes, whether you want to create indexed views, and how thorough of an analysis should be performed. The wizard does not recommend that any indexes be dropped if the Keep All Existing Indexes checkbox is selected. Recommendations will include only new indexes. If you are running SQL Server 2000 Enterprise Edition or Developer Edition, the Index Tuning wizard can create indexes on views if the Add Indexed Views checkbox is selected. The more thorough the analysis, the more significant will be the CPU consumption while analysis is being performed. If CPU consumption is a problem, take anyof the following measures:

- | Lower the level of analysis by selecting the Fast or Medium tuning modes. However, a thorough analysis can result in a greater overall improvement in performance.
- | Analyze a smaller workload and fewer tables.
- | Run the analysis against a test version of the production server. Save the results to a script and then run the script against the production server.
- | Run the wizard on a client computer, not the SQL Server.

After you select the Index Tuning wizard configuration, you must select a workload. Workload data comes from a trace file or trace table or a selection in the Query Analyzer.

The Query Analyzer selection option is available only if you start the Index

Tuning wizard from the Query Analyzer. Do not include index or query hints in the workload. If you do, the Query Optimizer formulates an execution plan based on the index hints, which might prevent the selection of an ideal execution plan.

After you select the workload, you can change the default index tuning parameters, select the tables for the wizard to analyze, and then run the analysis. Following the analysis, the wizard might not make index suggestions if there isn't enough data in the tables being sampled or if recommended indexes do not offer enough projected improvement in query performance over existing indexes.

To configure and run the Index Tuning wizard

1. Open Query Analyzer, and connect to your local server.
2. In the Editor pane of the Query window, type any character or press the space bar.
You must take this action for the Index Tuning wizard to be an available option in the Query pull-down menu.
3. Click the Query menu and then click Index Tuning wizard. The Welcome to the Index Tuning wizard screen appears.
4. Click Next. The Select Server and Database screen appears.
5. Select BookShopDB from the Database drop-down list box and click Next. The Specify Workload screen appears. Notice that the Query Analyzer radio button is selected. This option is available only when you start the Index Tuning wizard from Query Analyzer.
6. Click the My Workload File radio button. An Open window appears and TraceOLtrc is listed in the folder and file pane.
7. Double-click TraceO1.trc. The path and file name of the trace file appears in the Specify Workload screen.
8. Press the Advanced Options button to review the index tuning parameters and then click Cancel.
9. Press Next on the Specify Workload screen. The Select Tables to Tune screen appears.
10. Scroll down in the list and select the [dbo].[table01] checkbox.
11. Click Next. The analysis runs as the Index Tuning wizard determines the type of indexes to recommend. When the analysis completes, the Index Recommendations screen appears and two indexes are recommended. Below the index recommendations a bullet shows the estimated query performance improvement based on the sampled workload. The Index Tuning wizard recommends a clustered index named Table1 with a key on the UniqueID column and a non clustered index named Table012 with a key on the Col03 and LongCol02 columns. Later in the Index Tuning wizard screens, you can choose to save a script to create these indexes. You can customize the script before executing it. For example, you might want to name the indexes differently.
12. Click the Analysis button and review the various reports available from the Reports drop-down list box, then click Close.
13. In the Index Recommendations screen, click Next. The Schedule Index Update Job screen appears.

14. Click the Apply Changes checkbox and then click Next. The Completing The IndexTuning wizard screen appears.
15. Click Finish. The database b updated with the recommended changes and then a message box appears, indicating that the Index Tuning wizard has successfully completed.
16. Click OK to close the IndexTuning wizard.
17. Leave Query Analyzer open to complete the next practice.

@ @ @ @ @ @