Client/Server Architecture

MCAD
Amir Ahani

The terms *client*,*server,* and *client/server* can be used to refer to very general concepts or specific items of hardware or software. At the most general level, a *client* is any component of a system that requests services or resources from other system components. A *server* is any system component that provides services or resources to other system components.

For example, when you print a document from your workstation on a network, the workstation is the client and the print spooling machine is the server.

Any client/server data-based system consists of the following things:

- **The server.** A collection of data items and supporting objects organized and presented to facilitate services, such as searching, sorting, recombining, and retrieving, updating, and analyzing data. The database consists of the physical storage of data and the database services. All data access occurs through the server; the physical data is never directly accessed.
- **The client.**A software program that might be used interactively by a person or that could be an automated process. This includes all software that interacts with the server, either requesting data from the database or sending data to the database. Examples are: management utilities—those that are part of the SQL Server product and those bought separately, ad-hoc query and reporting software, custom applications, off-the-shelf applications, and Web server-based applications.
- **The communication between the client and the server.** The communication between the client and the server is largely dependent on how the client and server are implemented. Both physical and logical levels of communication can be identified.

When you communicate with someone using the telephone, the telephone system is the physical layer and a spoken natural language is the logical layer of communication. For a data-based system, the physical communication can be a network, if the server and the client are on different computers. It can be via inter-process communication if the server and the client are on the same computer. The logical communication may be low-level operating system calls, a proprietary data access language, or the open Structured Query Language (SQL).

All implementations of data-based systems fall into one of three categories:

- **File-based systems.** Commonly found on personal computers, these systems use an application that directly accesses data files on a local hard drive or on a network file server. These systems implement the database services and the logical communication as part of the client application, only the physical communication and the physical storage of data are external to the client

application. In this implementation, the client application fulfills the role of client and the role of server, as shown in Figure 1.

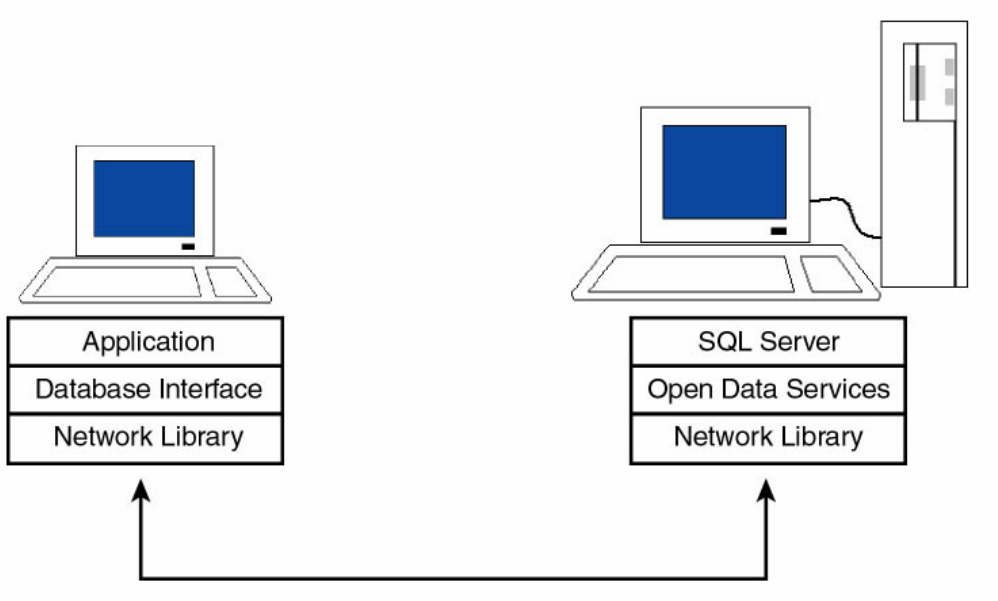


Figure 1 *File-based system*

- **Host based systems.** Typically used in legacy mainframe and mini-computer environments, these systems implement all or most of the database services and client functionality on a large central computer. The user views and interacts with the client application remotely using a terminal. The communication between the client and the database is done on the host computer. In this implementation, the host computer fulfills the role of client and the role of server, as shown in Figure 2.

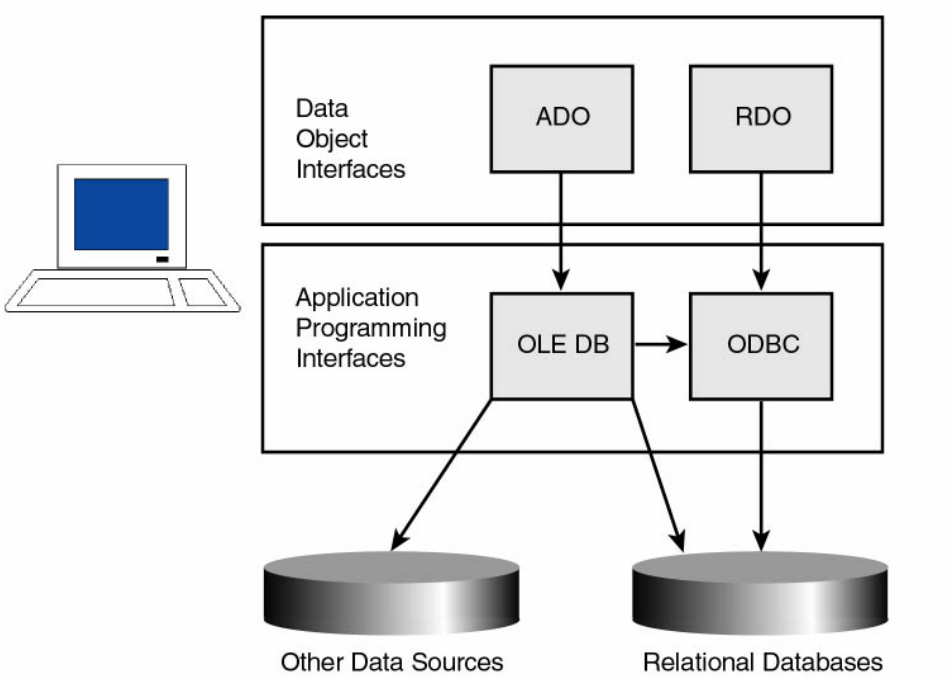


Figure 2 *Host-based systems.*

- **Client/server systems.** These systems are designed to separate database services from the client, allowing the communication between them to be more flexible and open. Database services are implemented on a powerful computer, allowing centralized management, security, and shared resources—therefore, the server in client/server is the database and its services. Client applications are implemented on a variety of platforms, using a variety of tools. This process allows greater flexibility and high quality user applications—this is the client in client/server. Figure 3 shows the client application and the database server in the client/server implementation.

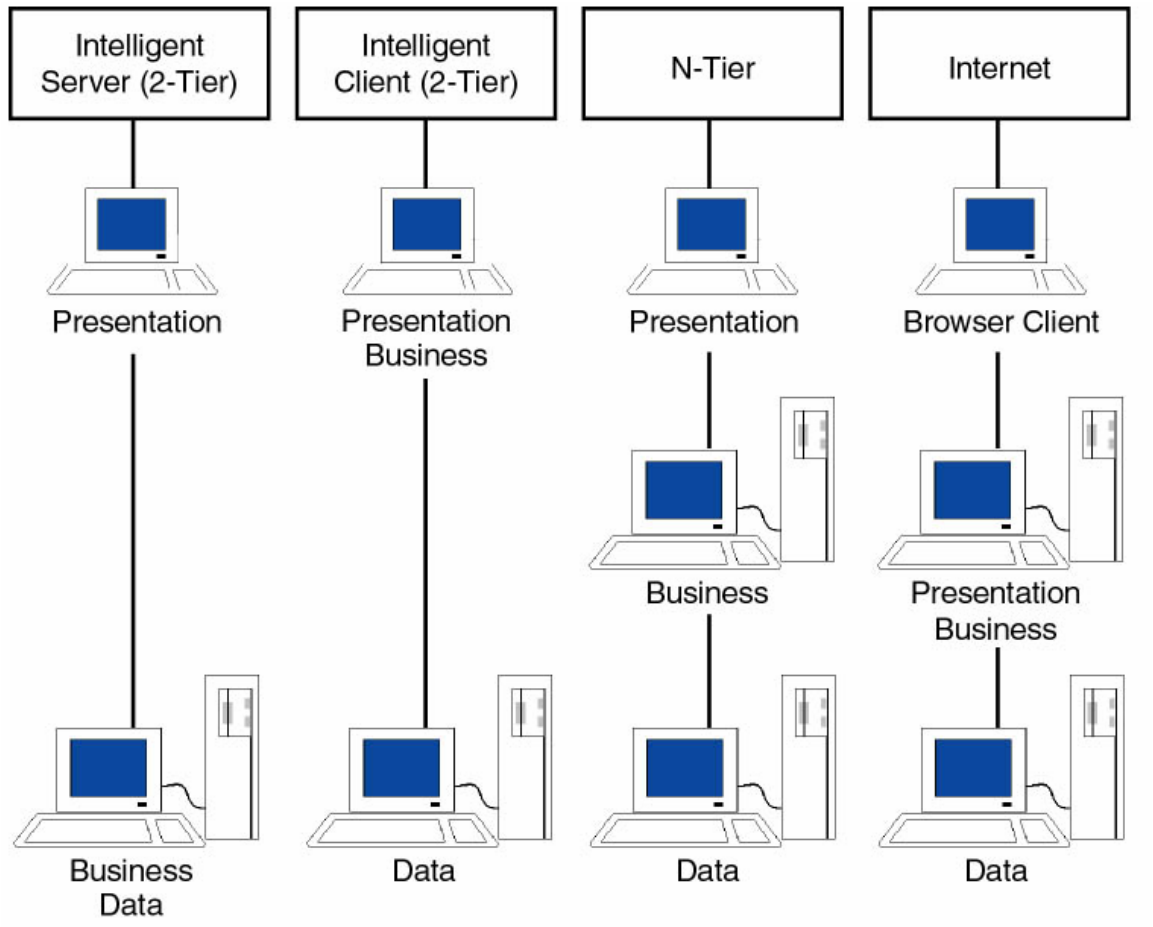


Figure 3 *Client/Server systems.*

The following table compares some of the advantages and disadvantages of file-based, host-based, and client/server systems. Many organizations now use a mix of these systems. For example, data capture may be performed on a host-based system with thousands of terminals. Data is then queried, manipulated, and analyzed by a client/server system, either directly on the host, or after transferring the data to another database.

File-based	Host based	Client/Server
Low cost	High initial cost	Variable cost
Low security	High security	Medium to high security
Low reliability	High reliability	Medium to high reliability
Application development possible with few skills	Application development requires skilled staff	Application development requires skilled staff
Well-suited to small	Not appropriate for	Can be used for small

databases and end-user databases	small or end-user databases	databases. Not appropriate for end-user-databases
Scalable to medium databases (~ 50 MB)	Scalable to very large databases (1000s of GB)	Scalable to very large databases (1000s of GB)
Minimal centralized management	Excellent centralized management	Excellent centralized management
Highly flexible end -user interface	Inflexible end-user interface	Flexible end-user interface
Low to medium vendor lock-in	High vendor lock-in	Medium vendor lock-in
Uses network inefficiently	Uses network efficiently	Can use network efficiently

Hundreds of commercial data based systems are available. These systems range from those comprised of a single application running on a single personal computer to those comprised of hundreds of applications running on complex networks of mainframe, mini, and personal computers. All commercial data-based systems have these three basic components. Clearly, therefore, these basic components could all occur on a single computer or may be distributed across many computers. Try to identify these components whenever you encounter a data-based system. In a large system, each component may be further layered into many parts, but you should always be able to distinguish the database, the client, and the communication between the two.

Relational Database Management System (RDBMS)

A *relational database* is a collection of data organized in two-dimensional tables consisting of named columns and (usually unique) rows. Each table represents the mathematical concept of a relation as defined in set theory. In set theory, columns are known as *attributes* and rows are known as *tuples*. The operations that may be performed on tables are similarly based on manipulation of relations to produce new relations, usually referred to as *queries* or *views*.

Relational databases differ from non-relational databases in that the database user is not aware of system dependencies that may be stored within the data. No knowledge of the underlying database is required; data can be queried and updated using standard languages (these languages together make up SQL) that produce a consistent result. SQL Server databases are relational.

An RDBMS is responsible for:

- Storing and making data available in tables.

- Maintaining the relationships between tables in the database.
- Ensuring the integrity of data, by making sure that rules governing the data values and defining the relationships between tables are not violated.
- Recovering all data to a point of known consistency in case of a system failure.

Transact-SQL

SQL Server uses Transact-SQL, a version of the Structured Query Language (SQL), as its database query and programming language. *SQL* is a set of commands that allows you to specify the information that you want to retrieve or modify. With Transact-SQL, you can access data and query, update, and manage relational database systems.

The American National Standards Institute (ANSI) and the International Standards Organization (ISO) have defined standards for SQL. Transact-SQL supports the latest ANSI SQL standard published in 1992, called ANSI SQL-92, plus many extensions to provide increased functionality.

Microsoft Index Server

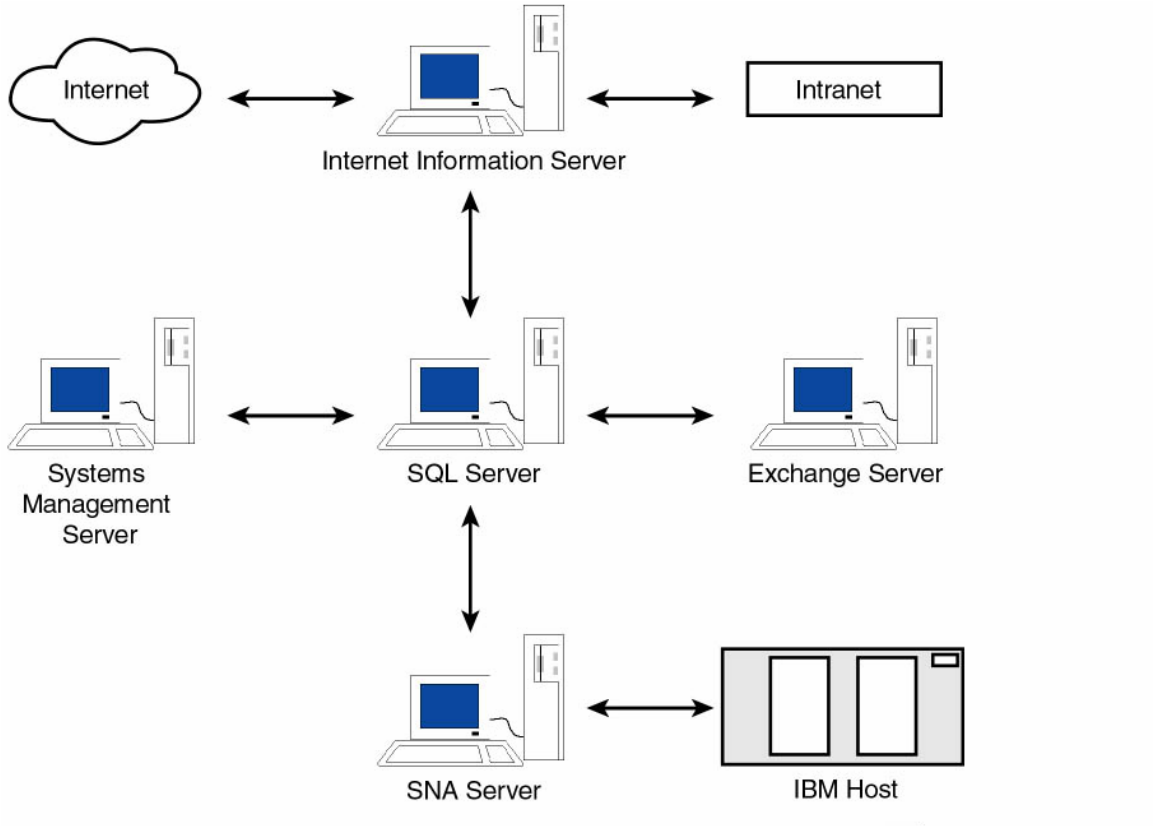
Microsoft Index Server, a full-text indexing and search engine supported by various Microsoft BackOffice products.

Microsoft Cluster Server

Microsoft Cluster Server (MSCS), a feature of Microsoft Windows NT Server Enterprise Edition, supports the connection of two servers, or nodes, into a cluster for higher availability and better manageability of data and applications. SQL Server works in conjunction with MSCS to switch automatically to the secondary node if the primary node fails.

SQL Server Integration with Microsoft BackOffice

SQL Server integrates well with other Microsoft BackOffice products. *BackOffice* is a group of server applications that work together to help you build business



The following table describes some commonly used BackOffice applications that work with or use SQL Server:

BackOffice application	Description
Microsoft Internet Information Server (IIS)	Allows Internet browser clients access to data in SQL Server.
Microsoft Exchange Server	SQL Server can send e-mail messages using Microsoft Exchange Server or other Messaging Application Programming Interface (MAPI)-compliant providers. SQL Server can send messages when an error occurs or when a scheduled task (such as a database backup) succeeds or fails.
Microsoft SNA Server	Links IBM environments running the Systems Network Architecture (SNA) protocol with PC-based networks. You can integrate SQL Server with IBM mainframe or AS/400 applications and data using SNA Server.

Microsoft Systems Management Server (SMS)	Manages computer software, hardware, and inventory. SMS requires SQL Server to store its databases.

SQL Server provides database, client, and communication components.

The database component of SQL Server is implemented as a group of 32-bit Windows applications or Windows NT services. Tools for installing, configuring, and managing these applications and services are also supplied.

The client component includes Windows 32-bit-based and command prompt-based server and database management and Transact-SQL query utilities. Many application programmers interface and object model components are supplied to enable custom application development.

The client-to-database communication for SQL Server is accomplished by two pieces of software, the database interface and a Net-Library. These are installed on the client and are included as part of the server installation. A utility for configuring the installed Net-Library client software is provided.

SQL Server Services

The SQL Server services are MSSQLServer, SQL Server Agent, and Microsoft Distributed Transaction Coordinator (MS DTC).

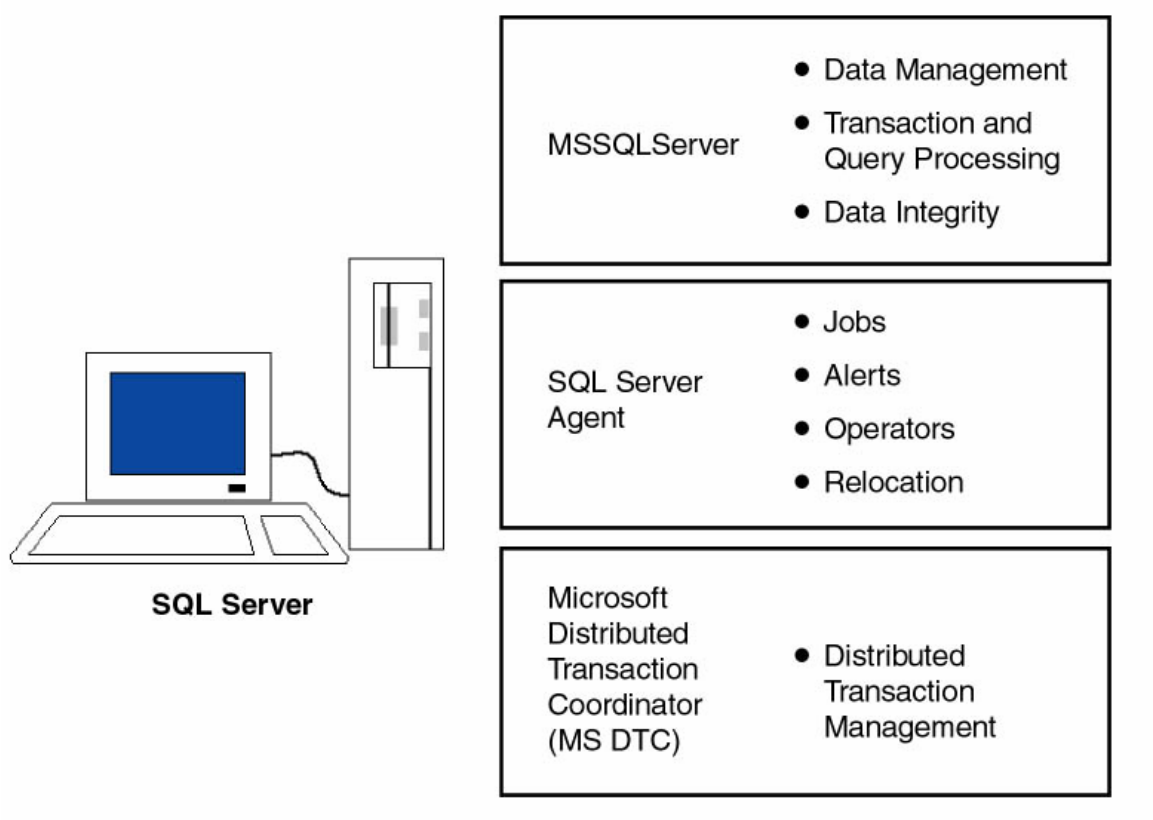


Figure 4 *SQL Server services.*

Optionally, you can also install the Microsoft Search Service. All SQL Server services run as services on Windows NT or as applications on Windows 32-bit platforms.

MSSQLServer Service

The MSSQLServer service is the RDBMS. The service processes all Transact-SQL statements and manages all files that comprise the databases on the server. All the other SQL Server services are dependent on the MSSQLServer service and exist to extend or complement the functionality of the MSSQLServer service. The MSSQLServer service:

- Allocates computer resources among multiple concurrent users.
- Prevents logical problems, such as timing requests from users who want to update the same data at the same time.
- Enforces data consistency and integrity.
- Enforces security.

SQL Server Agent Service

SQL Server Agent is a service that works in conjunction with SQL Server to create and manage local or multi-server jobs, alerts, and operators.

Microsoft Distributed Transaction Coordinator (MS DTC) Service

Microsoft Distributed Transaction Coordinator (MS DTC), also a component of Microsoft Transaction Server, is a transaction manager that allows clients to include several different sources of data in one transaction. MS DTC coordinates the proper completion of distributed transactions to ensure that all updates on all servers are permanent—or, in the case of errors, that all modifications are cancelled. This is achieved using a process called *two-phased commit*. MS DTC is an X/Open XA-compliant transaction manager.

Microsoft Search Service

The Microsoft Search Service provides full-text search capabilities for text column data. This optional service can be installed during the standard SQL Server installation or later.

SQL Server Client Software

SQL Server includes a variety of client software for designing and creating databases, querying data, and administering the server.

SQL Server Enterprise Manager Snap-in for Microsoft Management Console

SQL Server Enterprise Manager is a server administration and database management client. It is a Microsoft Management Console (MMC) snap-in. MMC is a shared user interface for BackOffice server management that provides a convenient, consistent environment for administration tools.

SQL Server Query Analyzer

The *SQL Server Query Analyzer* is a Transact-SQL query tool used to send individual or batched Transact-SQL statements to SQL Server. It also provides query analysis, statistics information, and the ability to manage multiple queries in different windows simultaneously.

NOTE

SQL Server Query Analyzer replaces ISQL/w found in previous versions of SQL Server.

SQL Server Administration Tools and Wizards

SQL Server provides many administrative tools and wizards that assist with particular aspects of SQL Server. The following table describes SQL Server tools and wizards:

Graphical tool	Purpose
SQL Server Client Configuration	Utility used to manage the communication components on SQL Server clients.
SQL Server Performance Monitor	Windows NT Performance Monitor settings file, provides a preconfigured view of some of the common SQL Server counters.
SQL Server Profiler (previous called SQL Trace)	Utility used to capture a continuous record of server activity and provide auditing capability.
SQL Server Service Manager	Graphical utility used for starting, stopping, and pausing SQL Server services.
SQL Server Setup	Application used to install and reconfigure SQL Server and SQL Server clients.
SQL Server wizards	Collection of tools that guide users through complex tasks, such as importing data, creating a database maintenance plan or configuring replication.
Data Transformation Services	A set of components that allows you to import, export, and transform data between multiple heterogeneous sources using an OLE DB-based architecture.

SQL Server Command Prompt Management Tools

SQL Server command prompt management tools allow you to enter Transact-SQL statements and execute script files. The following table describes the most frequently used command prompt utilities that are provided with SQL Server. Each file is an executable program.

Utility	Description
osql	Utility that uses Open Database Connectivity (ODBC) to communicate with SQL Server—primarily used to execute batch files containing one or more SQL statements
bcp	Batch utility used to export and import data to and from SQL Server and non SQL Server databases. This utility copies data to or from a standard text or binary data file.

NOTE

The osql utility replaces isql found in previous versions of SQL Server. The isql utility, which uses DB-Library to communicate with SQL Server is available for backward compatibility.

SQL Server Help and SQL Server Books Online

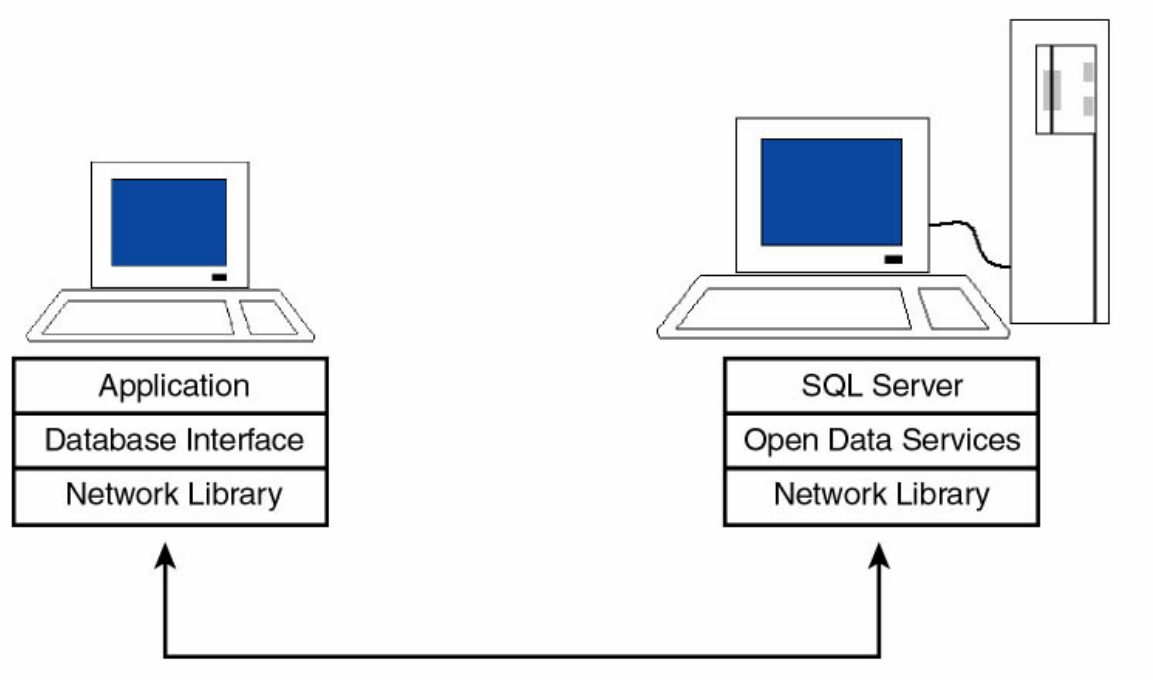
SQL Server offers extensive documentation and different types of Help to assist you. The following table describes each type of Help provided by SQL Server:

Type of Help	Description
Application Help	Several SQL Server tools—including SQL Server Enterprise Manager, SQL Server Profiler, and SQL Server Query Analyzer—provide this context-sensitive help on the application interface. Clicking the Help button, selecting a command on the Help menu, or pressing F1are all ways to open Application Help.
Transact-SQL Help	When you use SQL Server Query Analyzer, highlight a statement name and then press SHIFT+F1.
SQL Server Books Online	SQL Server Books Online provides online, indexed, searchable access to all the SQL Server documentation.

SQL Server provides many structured architectures that hide underlying technical details, simplifying the development, maintenance, and management of your database applications.

Communication Architecture

SQL Server uses layered communication architecture to isolate applications from the underlying network and protocols. This architecture allows you to deploy the same application in different network environments.



The components in the communication architecture include the following:

Application

An *application* is developed using a database application programming interface (API) or object model. The application has no knowledge of the underlying network protocols used to communicate with SQL Server. Examples of client applications include SQL Server Enterprise Manager, SQL Server Query Analyzer, osql, ad -hoc query and report writers, web server-based applications, and business applications.

Database Interface

The *database interface* is used by the application to send requests to SQL Server. The interface receives results returned by SQL Server and passes them to the application. The interface may process the results before passing them to the application. The database interfaces for SQL Server are OLE DB, ODBC, and DB-Library.

Network Library

A *network library*is a communication software component that packages the database requests and results for transmission by the appropriate network protocol. A network library, also known as a Net- Library, must be installed on both the client and server.

Clients and servers can use more than one Net-Library concurrently, but they must use a common network library in order to communicate successfully. SQL Server has Net-

Libraries that support network protocols such as TCP/IP, Named Pipes, Novell IPX/SPX, Banyan VINES/IP, and AppleTalk ADSP.

Some Net-Libraries support only one network protocol; for example, the TCP/IP Sockets Net-Library supports only the TCP/IP network protocol. Other Net-Libraries, such as the Multi-protocol Net-Library, support multiple network protocols. The entire SQL Server database interfaces work on any of the Net-Libraries.

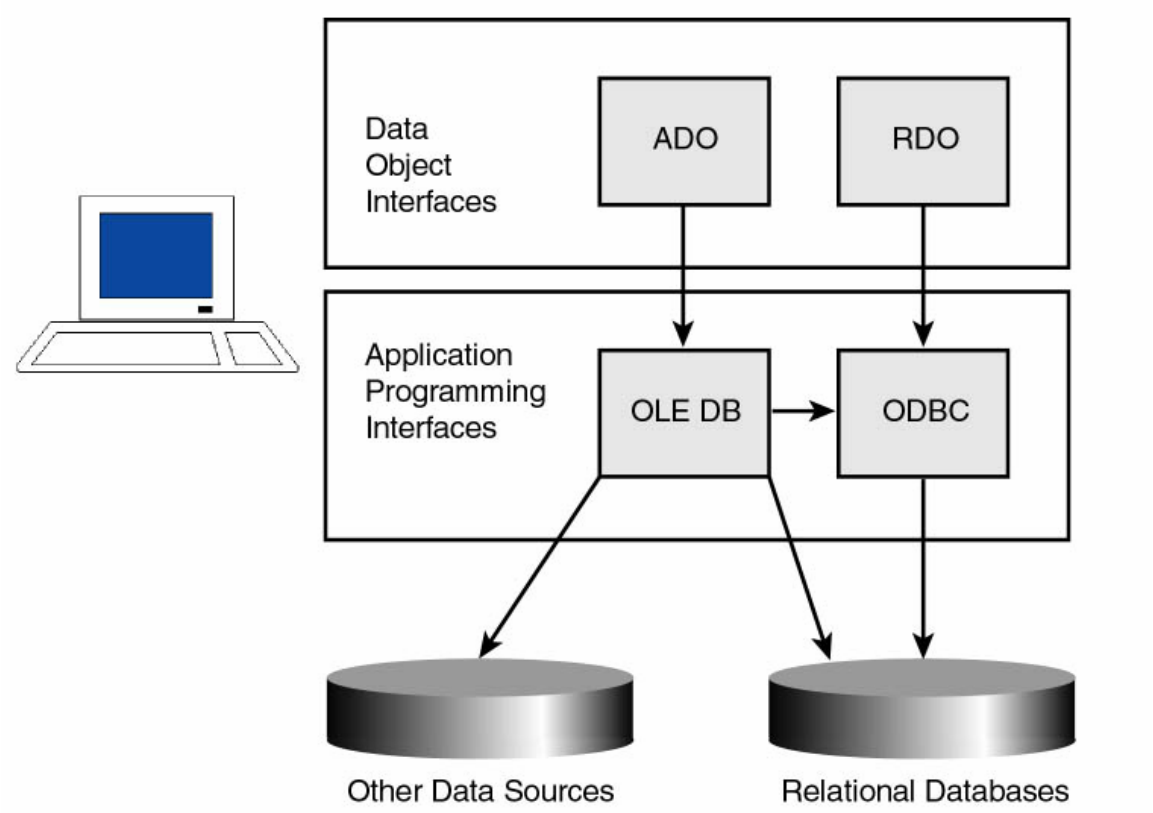
Consider two main criteria when choosing a Net-Library. First, review the capabilities of the Net-Libraries and match them to your needs; for example, the Multi-protocol Net-Library supports encrypting data sent across the network but it does not support server name enumeration. Second, match the Net-Library to your network infrastructure and client network software.

Open Data Services

Open Data Services function as the interface between server Net-Libraries and server-based applications. SQL Server itself and extended stored procedure DLLs are examples of server-based applications. This component handles network connections, passing client requests to SQL Server for processing and returning any results and replies to SQL Server clients. Open Data Services automatically listens on all server Net-Libraries that are installed on the server.

Data Access Architecture

Users use SQL Server databases through an application that uses a data object interface or an API to gain access to SQL Server. .



SQL Server supports commonly used and emerging database interfaces. It supports low level native APIs, as well as easy-to-use data object interfaces.

Application Programming Interfaces (APIs)

A database *application programming interface (API)* defines how to write an application to connect to a database and pass commands to the database. SQL Server provides native support for two main classes of database APIs, which in turn determine the data object interface that you can use. Use the database APIs to have more control over application behavior and to gain better performance.

OLE DB

OLE DB is a Component Object Model (COM)-based data access interface. It supports applications written to use OLE DB or data object interfaces that use OLE DB. OLE DB is designed to work with relational databases (such as those in SQL Server) as well as with non-relational data sources (such as a full-text index or an e-mail message store).

OLE DB uses a provider to gain access to a particular data source. Providers for SQL Server, Oracle, Jet (Microsoft Access databases), and ODBC are supplied with SQL Server. Using the OLE DB provider for ODBC, OLE DB can be used to gain access to any ODBC data source.

ODBC

ODBC is a call-level interface. It communicates directly with SQL Server and supports applications or components that are written to use ODBC or data object interfaces that use ODBC. ODBC is designed to work with relational databases (such as that in SQL Server) only, although there are limited ODBC drivers available for some non-relational data sources.

ODBC uses a driver to gain access to a particular data source.

Data Object Interfaces

In general, *data object interfaces* are easier to use than database APIs but may not expose as much functionality as an API.

ActiveX Data Objects (ADO)

ActiveX Data Objects (ADO) encapsulates the OLE DB API in a simplified object model that reduces application development and maintenance costs. ADO can be used from many development environments, such as Microsoft Visual Basic, Microsoft Visual C++, Visual Basic for Applications, Active Server Pages (ASP), and the Microsoft Internet Explorer scripting object model. These characteristics make ADO the primary database interface for developing client/server business applications.

Remote Data Objects (RDO)

Remote Data Objects (RDO) maps over and encapsulates the ODBC API. RDO can be used from Microsoft Visual Basic and Visual Basic for Applications.

Administration Architecture

SQL Server provides a variety of management tools that minimize and automate routine administrative tasks.

SQL Server Administration

You can administer SQL Server using:

- Batch utilities provided with SQL Server, such as osql and bcp.
- Graphical administration tools provided with SQL Server (SQL Server Enterprise Manager, SQL Server Query Analyzer, and SQL Profiler).
- COM-compatible applications, written in languages such as Visual Basic.

All these tools use Transact-SQL statements to initiate actions on SQL Server. In some cases, you will specify the Transact-SQL statements explicitly; in others, the tool will generate the Transact-SQL statement(s) for you.

SQL Distributed Management Objects (SQL-DMO)

SQL Distributed Management Objects (SQL-DMO) is a COM-based object model provided by SQL Server. SQL-DMO hides the details of the Transact-SQL statements; it is suitable for writing administration applications and scripts for SQL Server. The graphical administration tools provided with SQL Server are written using SQL-DMO. SQL-DMO is not a data interface model; it should not be considered for writing standard database applications.

SQL Server Agent

SQL Server Agent is a service that works in conjunction with SQL Server to enable the following administrative capabilities:

Alert Management

Alerts provide information about the status of a process, such as when a job is complete or when an error occurs. SQL Server Agent monitors the Windows NT application event log for events. If SQL Server Agent has been configured to respond to the event, an alert is generated when a new event appears in the log.

Notification

SQL Server Agent can send e-mail messages, page an operator, or start another application when an alert occurs. For example, you can set an alert to occur when a database or transaction log is almost full and then have the alert generate a notification to inform someone that the alert has occurred.

Job Execution

SQL Server Agent includes a job creation and scheduling engine. Jobs can be simple (for example, single-step operations), or they can be complex (for example, multistep tasks that require scheduling). You can create job steps with Transact-SQL, scripting languages, or operating system commands.

Replication Management

Replication is the process of copying data or transactions from one SQL Server to another. SQL Server Agent is responsible for synchronizing data between servers, monitoring the data for changes, and replicating the information to other servers.

Application Architecture

Before you design an application for SQL Server, it is important to spend time designing a database to model the business accurately. A well-designed database requires fewer

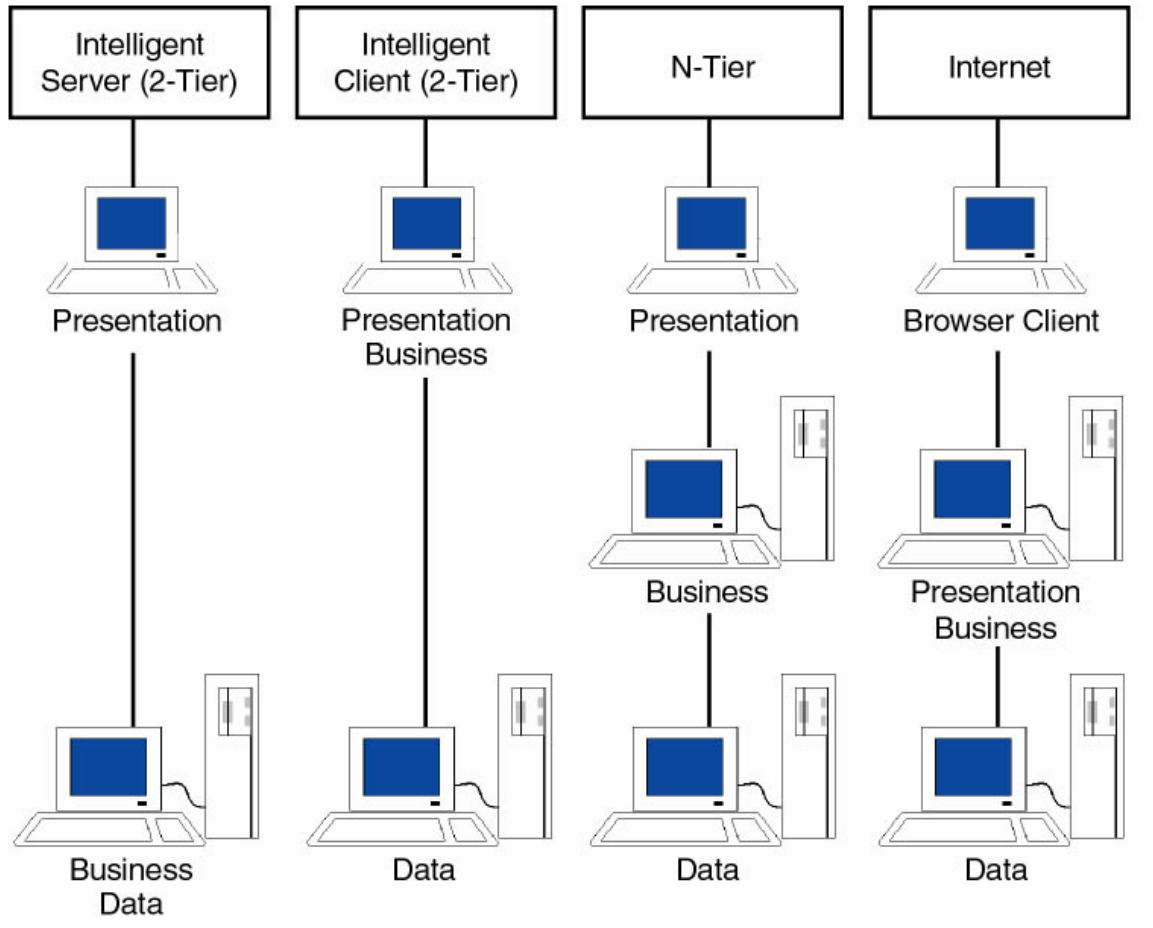
changes and generally performs more efficiently. Planning a database design requires knowledge of the business functions that you want to model and the database concepts and features that you use to represent those business functions.

You can use many application architectures with SQL Server. The application architecture that you select depends on your database design and your business needs; it also affects how you develop, deploy, and manage your software application. This lesson presents some of the most common architectural designs and deployment options.

Application Layering

Selecting a layered application approach affords flexibility and a choice of management options. You can divide software applications into three logical layers. Each layer, which can physically reside on one or more servers, provides clearly defined services.

- Presentation Services. Format and present information to users and provide an interface for data entry.
- Business Services. Implement business rules such as checking limits, validating data, and providing calculated or summarized data.
- Data Services. Provide storage, retrieval, security, and integrity for data.



After these layers have been defined, they can be implemented in various ways:

Intelligent Server (2-Tier)

Most processing occurs on the server, with presentation services handled on the client. In many instances, most of the business services logic is implemented in the database. This design is useful when clients do not have sufficient resources to process the business logic. However, the server can become a bottleneck because database and business services compete for the same hardware resources. Network use can be inefficient because the client has to send all data to the server for verification, sometimes resending the same data many times until the server approves it.

Corporate applications designed from a database-centric point of view are an example of this design.

Intelligent Client (2-Tier)

Most processing occurs on the client, with data services handled on the server. This design is the traditional client/server environment that is widely used. However, network traffic can be heavy and transactions longer (that is, clients may lock data for lengthy periods), which can in turn affect performance.

Applications developed for small organizations with products such as Microsoft Access are an example of this design.

N-Tier

Processing is divided among a database server, an application server, and clients. This approach separates logic from data services, and you can easily add more application servers or database servers as needed. However, the potential for complexity increases, and this approach may be slower for small applications.

Multi-tiered enterprise applications and applications developed with transaction processing monitors are examples of the N-Tier design.

Internet

Processing is divided into three layers, with the business and presentation services residing on the Web server and the clients using simple browsers. Any client that has a browser is supported, and software does not need to be maintained on the client.

A Website that uses several Web servers to manage connections to clients and a single SQL Server database that services requests for data is an example of this architecture.

SQL Server Security Overview

SQL Server validates users at two levels. *Login authentication* ensures that the user is a known, valid user of the SQL Server. *Permissions validation* checks that the user is authorized to use a particular statement or object in a database.

Login Authentication

A user must have a login account to connect to SQL Server. SQL Server recognizes two login authentication mechanisms—SQL Server authentication and Windows NT authentication (Figure 5)—each of which has a different type of login account.

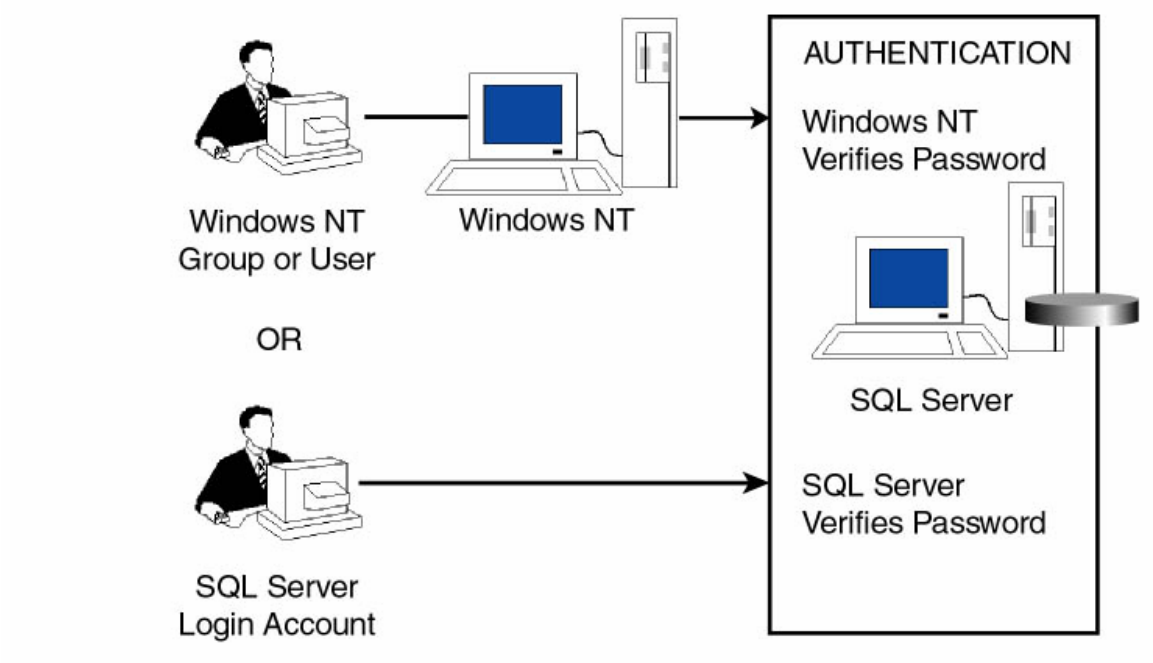


Figure 5 *Login authentication.*

SQL Server Authentication

When using SQL Server authentication, a SQL Server system administrator defines a SQL Server login account and password. Users must supply both the SQL Server login and password when they connect to SQL Server.

Windows NT Authentication

When using Windows NT authentication, a Windows NT user or group account controls user access to SQL Server—a user does not provide a SQL Server login account when connecting. A SQL Server system administrator must grant access to SQL Server to either the Windows NT user account or the Windows NT group account.

Authentication Mode

When SQL Server is running on Windows NT, a system administrator can specify that it run in one of two authentication modes:

Windows NT Authentication Mode

When the SQL Server is using Windows NT Authentication Mode, only Windows NT authentication is allowed. Users cannot specify a SQL Server login account. A SQL Server installed on Windows 95/98 does not support this mode.

Mixed Mode

When the SQL Server is using Mixed Mode, users can connect to SQL Server with Windows NT authentication or SQL Server authentication.

CAUTION

Clients connecting to a server running in Windows NT Authentication Mode must use a Windows NT authenticated connection. Clients connecting to a server running in Mixed Mode choose an authentication mechanism.

Permission Validation

SQL Server accepts Transact-SQL statements after a user's login has been successfully authenticated. Each time a user sends a statement, SQL Server checks that the user has permission to carry out the action requested by the statement. If the user has permission, the action is carried out; if not, an error is returned to the user (Figure 6).

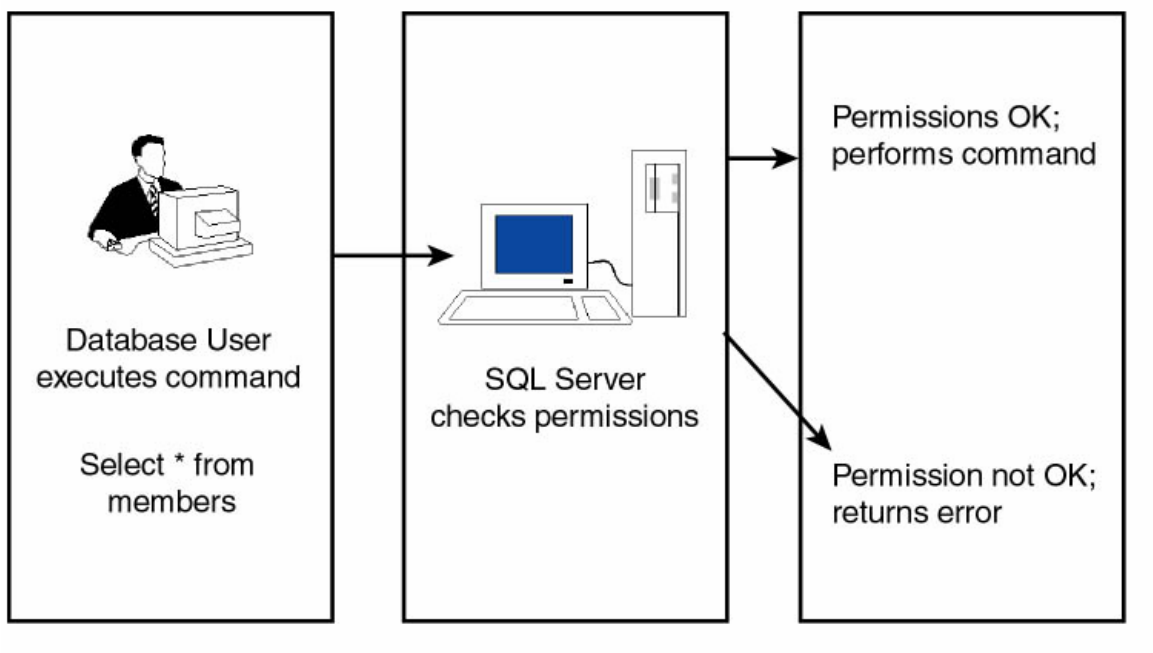


Figure 6 *Permission validations.*

NOTE

In most cases, the user will be interacting with an application user interface, unaware of the Transact-SQL statements that their actions are generating.

Database User Accounts and Roles

User accounts and roles, which identify a successfully logged-in user within a database, are used to control ownership of objects. Permissions to execute statements and use objects in a database are granted to users and roles. Each user account is mapped to a SQL Server login, as shown in Figure 7.

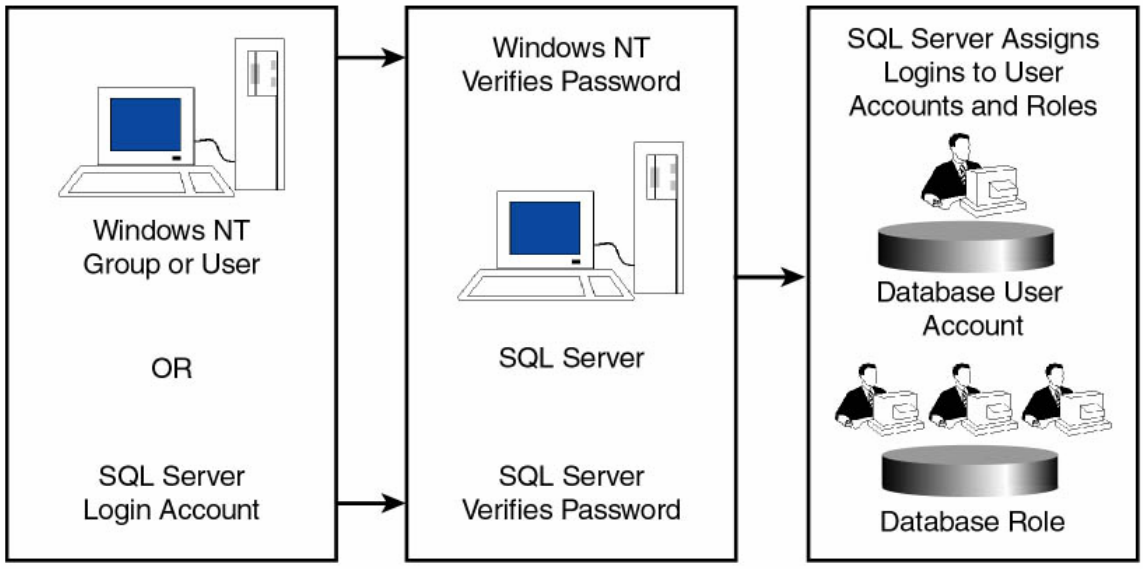


Figure 7 *Users and roles.*

Database User Accounts

The *user accounts* used to apply security permissions are mapped to Windows NT user or group accounts or to SQL Server login accounts. User accounts are specific to a database and cannot be used in other databases.

Database Roles

Roles enable you to assemble users into a single unit to which you can apply permissions. Both server-level and database-level roles exist. A user can be made a member of more than one role.

SQL Server provides predefined server and database roles for common administrative functions so that you can easily assign a selection of administrative permissions to a particular user simply by making them a member of the appropriate roles. You also can create your own user-defined database roles.

NOTE

Roles replace the SQL Server version 6.5 concepts of aliases and groups.

Fixed Server Roles

Fixed server roles provide groupings of administrative privileges at the server level, independently of databases. Examples of fixed server roles are roles for system administrators, database creators, and security administrators.

Fixed Database Roles

Fixed database roles provide groupings of administrative privileges at the database level. Examples of fixed database roles are roles for backing up and restoring a database, security administrators, reading data, and modifying data.

User-defined Roles

You also can create your own database roles to represent work performed by a group of employees in your organization. You do not have to grant, revoke, or deny permissions from each person. If the function of a role changes, you can easily change the permissions for the role and the changes apply automatically to all members of the role.

Elements of Transact-SQL

As you write and execute Transact-SQL statements, you will use:

- Data Control Language (DCL) statements, which are used to control permissions on database objects.
- Data Definition Language (DDL) statements, which are used to create objects in the database.
- Data Manipulation Language (DML) statements, which are used to query and modify data.
- Additional language elements, such as variables, operators, functions, control-of-flow language, and comments.

Data Control Language Statements

DCL statements are used to change the permissions associated with a database user or role. The following table describes the DCL statements:

Statement	Description
GRANT	Creates an entry in the security system that allows a user to work with data or execute certain Transact-SQL statements.
DENY	Creates an entry in the security system denying permission from a security account and prevents the user, group, or role from inheriting the permission through its group and role memberships. By default, only members of the sysadmin, dbcreator, db_owner, or db_securityadmin roles can execute DCL statements.

Example

This example grants the public role permission to query the Products table.

```
USE Northwind

GRANT SELECT ON Products TO public
```

NOTE

DCL statements are covered in detail in *System Administration for Microsoft SQL Server 7.0*

Data Definition Language Statements

DDL statements define database structure by creating and managing databases and database objects, such as tables and stored procedures. Most DDL statements take the form shown here:

- CREATE object_name
- ALTER object_name
- DROP object_name

By default, only members of the sysadmin, dbcreator, db_owner, or db_ddladmin roles can execute DDL statements. In general, it is recommended that no other accounts be used to create database objects. If different users create their own objects in a database, then each object owner is required to grant the proper permissions to each user of those objects. This requirement causes an administrative burden so you should avoid having different users create objects. Restricting statement permissions to these roles also avoids problems with object ownership that can occur when an object owner has been dropped from a database or when the owner of a stored procedure or view does not own the underlying tables.

If multiple user accounts have been used to create objects, the sysadmin and db_owner roles can use the sp_changeobjectowner system stored procedure to consolidate object ownership.

Example

The following script creates a table called *Customer* in the Northwind database. It includes CustID, Company, Contact, and Phone columns.

```
USE Northwind
CREATE TABLE Customer
(
CustID int,
Company varchar(40),
Contact varchar(30),
Phone char(12)
```

)

NOTE

In previous versions of SQL Server the SETUSER function could be used to impersonate another user. This function is included in SQL Server 7.0 for backward compatibility and can be used by the sysadmin and db_owner roles. You should remove this function from your databases and applications as SQL Server may not support it in future versions.

SQL Server Object Names

SQL Server provides a series of standard naming rules for object identifiers and a method of using delimiters for identifiers that are not standard. It is recommended that you name objects using the standard identifier characters if possible.

Standard Identifiers

Standard identifiers can be from 1 to 128 characters in length and can include letters, symbols (@, #, _, or $), and numbers. No embedded spaces are allowed in standard identifiers. Rules for using identifiers include:

- The first character must be an alphabetic character of a–z or A–Z or one of the symbols @, #, or –.
- After the first character, identifiers can include letters, numerals, or the @, #,–, or $ symbols.
- Identifier names starting with a symbol have special uses:

  An identifier beginning with @ denotes a local variable or parameter.

  Many SQL Server function identifiers begin with @@, so you should not create identifiers that start with @@.

  An identifier beginning with # denotes a local temporary table or procedure.

  An identifier beginning with ## denotes a global temporary object.

TIP

Names for temporary objects should not exceed 29 characters, including the # or ## symbol, because SQL Server gives them an internal numeric suffix.

Delimited Identifiers

If an identifier complies with all of the rules for the format of identifiers, it can be used with or without delimiters. If an identifier does not comply with one or more of the rules for the format of identifiers, it must always be delimited.

Delimited identifiers can be used in the following situations:

- When names contain embedded spaces or other characters not allowed in standard identifiers
- When reserved words are used for object names or portions of object names
- Delimited identifiers are enclosed in brackets or double quotation marks when they are used in Transact-SQL statements. Forexample:
- Bracketed identifiers are delimited by square brackets ([ ]):

```
SELECT * FROM [Blanks In Table Name]
```

- Quoted identifiers are delimited by double quotation marks (""):

```
SELECT * FROM "Blanks in Table Name"
```

NOTE

Quoted identifiers can only be used if the SET QUOTED_IDENTIFIER option is set ON. Bracketed delimiters can always be used, regardless of the status of the SET QUOTED_IDENTIFIER option.

Naming Guidelines

When naming database objects, you should:

- Keep names short.
- Use meaningful names where possible.
- Use a clear and simple naming convention. Decide what works best for your situation and be consistent. Try not to make naming conventions too complex because they can become difficult to track or understand. For example, you can remove vowels if an object name must resemble a keyword (such as a backup stored procedure named *bckup*).
- Use an identifier that distinguishes the type of object, especially for views and stored procedures. System administrators often mistake views for tables, an oversight that can cause unexpected problems. For example, prefix all view names with the letters *vw*.
- Keep object names and user names unique. For example, avoid creating a *sales* table and a *sales*role within the same database.
- Names are not case sensitive but case is maintained. Transact-SQL statements are not case sensitive but are generally entered in uppercase.

Referring to SQL Server Objects

You can refer to SQL Server objects in several ways. You can specify the full name of the object (its fully qualified name), or specify only part of the object's name and have SQL Server determine the rest of the name from the context in which you are working.

Fully Qualified Names

The complete name of a SQL Server object includes four identifiers: server name, database name, owner name, and object name in the following format:

```
server.database.owner.object
```

A name that specifies all four parts is known as a *fully qualified name*Each object created in SQL Server must have a unique fully qualified name. For example,two tables can be named *orders* in the same database as long as they belong to different owners. In addition, column names must be unique within a table or view.

Partially Specified Names

When referencing an object, you do not always have to specify the server, database, and owner. You can omit leading identifiers. You can also omit intermediate identifiers as long as their position is indicated by periods. The valid formats of object names are as follows:

```
server.database.owner.object
server.database..object
server..owner.object
server...object
database.owner.object
database..object
owner.object
object
```

When you create an object, SQL Server uses the following defaults if different parts of the name are not specified:

- Server defaults to the local server.
- Database defaults to the current database.
- Owner defaults to the user name in the specified database associated with the login ID of the current connection.

A user that is a member of a role can explicitly specify the role as the object owner. A user that is a member of the db_owner or db_ddladmin role in a database can specify the dbo user account as the owner of an object. This practice is recommended because administration is simpler when all objects are owned by the same user account.

Example

The following example creates an order_history table in the Northwind database:

```
CREATE TABLE Northwind.dbo.order_history
         (
           OrderID INT
         , ProductID int
         , UnitPrice money
         , Quantity int
         , Discount decimal
         )
```

Most object references use three-part names and default to the local server. Four-part names are generally used for distributed queries or remote stored procedure calls.

Data Manipulation Language Statements

DML statements work with the data in the database. By using DML statements, you can change data or retrieve information. DML statements include:

- SELECT
- INSERT
- UPDATE
- DELETE

By default, only members of the sysadmin, dbcreator, db_owner, db_datawriter, or db_datareader roles can execute some or all DML statements.

Example

This example retrieves the category ID, product name, product ID, and unit price of the products in the Northwind database.

```
SELECT CategoryID, ProductName, ProductID, UnitPrice
FROM Northwind..Products
```

- **To write a SELECT statement that returns ordered data**

In this exercise, you will write a statement that returns all the rows and columns from the Products table and sorts the results in ascending order by the ProductName column. Log on your computer as Administrator or another user that is a member of the Windows NT Administrators group.

1. Open SQL Server Query Analyzer and log on the (local) server with Microsoft Windows NT authentication. Your account is a member of the Windows NT Administrators group, which is automatically mapped to the SQL Server sysadmin role.

2. In the DB list box, click Northwind. The current database for your connection is now Northwind. You can also change you database context by executing the USE statement.

```
USE Northwind
```

3. Write a SELECT statement that returns all the rows and columns from the Products table and sorts the results in ascending order by the ProductName column.

```
SELECT * FROM Products ORDER BY ProductName
```

4. From the Query menu, click Results in Grid.
5. On the toolbar, click Execute Query (the green arrowhead).
6. The results of the query are displayed in a grid in the results pane. Click the Messages tab in the results pane to see messages that were returned with the results but are displayed in a grid:

- **To execute a system stored procedure to find information about a table and execute a SELECT statement that retrieves specific rows**

In this exercise, you will execute the sp_help system stored procedure on the Products table to find the column names and other information for the table. Then you will write a statement that retrieves products from a specific category.

Write and execute a statement that calls the sp_help system stored procedure to find the names of the columns of the Products table in the Northwind database.

```
EXEC sp_help Products
```

TIP

Change the execution mode to Results in Text when you execute sp_help. In the Results in Grid mode, you will get eight grids, each with different rows of information about the Products table.

1. Do not delete the text of the sp_help statement from the previous step. In the same query window, on a new line, write a SELECT statement that retrieves all products with a CategoryID of 4 from the Products table.

```
SELECT * FROM Products WHERE CategoryID = 4
```

2. Highlight the SELECT statement you typed in step 2.
3. Execute the query and notice that only the highlighted statement is executed.

TIP

To see more information about the SELECT statement (or any Transact-SQL statement, system stored procedure, or system table), highlight the statement name in the query window and press SHIFT+F1.

Additional Language Elements

Some additional elements of the Transact-SQL language include local variables, operators, functions, control of flow statements, and comments.

Local Variables

Variables are language elements with assigned values. You can use local variables in Transact-SQL.

Local Variables (User-defined)

A *local variable* is defined in a DECLARE statement, assigned an initial value in a SET statement, and then used within the statement, batch or procedure in which it was declared. A local variable name always begins with one @ symbol preceding its name.

Syntax

```
DECLARE {@variable_name data_type } [,...n]

SET @local_variable_name = expression
```

Example

The following code creates the @lvname local variable, assigns a value to it, and then queries the database to select the record containing the value of the local variable:

```
DECLARE @lvname char(20)
SET @lvname = 'Dodsworth'
SELECT LastName, FirstName, Title FROM Northwind..Employees
     WHERE LastName = @lvname
```

Result

```
LastName             FirstName  Title
-------------------- ---------- ------------------------------
```

```
Dodsworth            Anne       Sales Representative

(1 row(s) affected)
```

Including it in a SELECT list can also set the value of a local variable.

Example

The following example stores the highest employee ID to a variable named @EmpIDVariable:

```
DECLARE @EmpIDVariable int
SELECT @EmpIDVariable = MAX(EmployeeID)
FROM Northwind..Employees
```

If SELECT returns multiple rows, then the variable is set to the value from the last row returned by SELECT.

Example

The following example stores the product ID of the last product returned by SELECT from the Products table to a variable named @ProdIDVariable:

```
DECLARE @ProdIDVariable int
SELECT @ProdIDVariable = ProductID
FROM Northwind..Products
```

NOTE

Previous versions of SQL Server had objects called *global variables*. These global values returned system information. They were not used as variables because you could not set their values. The names of these values begin with @@. In SQL Server 7.0, these values are still available but they are now called *functions*. Specifically, these functions are part of a group of functions called *scalar functions*. SQL Server 7.0 uses global variables in Data Transformation Services but these are not part of Transact-SQL.

Operators

Operators are symbols that perform operations on one or more values called *operands*. The operations that can be performed include mathematical computations, string concatenations, logical operations, and comparisons between columns, constants, and variables. They can be combined in expressions and search conditions. When you combine them, the order in which the operators are processed is based on a predefined precedence.

Types of Operators

SQL Server supports six commonly used types of operators.

- Assignment Operator. Transact-SQL supports a single assignment operator, the equals (=) sign.
- Arithmetic Operators. Arithmetic operators perform computations with numeric columns or constants. Transact-SQL supports multiplication (*), division (/), addition (+), and subtraction (-) arithmetic operators. The *modulo (%) operator*, which returns the integer remainder after integer division, is also supported.
- Unary Operators. Unary operators perform an operation on a single numeric expression. The unary operators are + (positive), - (negative), and ~ (bitwise NOT). A unary operator is placed before the expression on which it operates.
- Comparison Operators. Comparison operators compare two expressions. Comparisons can be made between variables, columns, and expressions of similar type. Comparison operators include less than (<), greater than (>), equal to (=), and not equal to (<>).
- String Concatenation Operator. The string concatenation operator (+) concatenates string values. All other string manipulation is handled through string functions.
- By default, the empty string (**' '**) is interpreted as an empty string in INSERT statements on varchar data. In concatenating varchar, char, nvarchar, nchar, text, or ntext data, the empty string is interpreted as an empty string. For example, 'abc' + ' ' + 'def' is stored as 'abcdef'. The empty string is never evaluated as a null value.

  NOTE

  The default behavior is different from previous versions of SQL Server, which interpreted an empty string as a single space. If sp_dbcmptlevel is set to 65, then SQL Server 7.0 will use the old behavior.

- Logical Operators. The logical operators are AND, OR, and NOT. AND and OR are used to connect search conditions in WHERE clauses. NOT reverses the result of a search condition.

  AND connects two conditions and returns TRUE only when both conditions are true.

  OR connects two conditions and returns TRUE when either of the conditions is true.

Operator Precedence Levels

If you use multiple operators (logical or arithmetic) to combine expressions, SQL Server processes the operators in order of their precedence, which may affect the resulting value. Operators have the following precedence levels (from highest to lowest):

- + (Positive), - (Negative), ~ (Bitwise NOT)
- * (Multiply), / (Division), % (Modulo)
- + (Add), (+ Concatenate), - (Subtract)
- =, >, <, >=, <=, <>, !=, !>, !< (Comparison operators)
- ^ (Bitwise Exclusive OR), & (Bitwise AND), | (Bitwise OR)
- NOT
- AND
- ALL, ANY, BETWEEN, IN, LIKE, OR, SOME
- = (Assignment)

Use parentheses to override the defined precedence of the operators in an expression. SQL Server handles the most deeply nested expression first. When two operators in an expression share the same level of precedence, they are processed in order from left to right.

Functions

Transact-SQL provides many functions that return information. *Functions*return values that can be used in expressions. Some functions take input parameters while others return values without any input. Functions can be grouped into three types:

- Rowset functions return an object that can be used in place of a table in Transact-SQL statements.
- Aggregate functions return a single summary value from a set of values.
- Scalar functions return a single value. Scalar functions may take none, one, or more input arguments.

The following table lists the major categories of scalar functions:

Function category	Description
Configuration	Return information about the current configuration.
Cursor	Return information about cursors.
Date and Time	Perform operations on a date and time input value, and return a string, numeric, or date and time value.
Mathematical	Perform mathematical calculations on input arguments and return a numeric value.
Metadata	Return information about the database and database objects.

Security	Return information about users, roles, and permissions.
String	Perform operations on string (char, nchar, varchar, or nvarchar) input arguments and return a string or numeric value.
System	Perform operations, such as testing and converting data types, and return information about values, objects, and SQL Server settings.
System Statistical	Return statistical information about the system.
Text and Image	Perform operations on text or image value or column arguments, and return information about the arguments.

Syntax

The syntax for most functions is similar, as shown here:

```
function_name(arguments)
```

OR

```
@@function_name
```

NOTE

Function names that begin with @@ used to be called *global variables* in previous versions of SQL Server.

Function Examples

The following examples show how some of the commonly used functions can be applied:

Example 1

This example shows the use of the @@VERSION function to return version information about SQL Server and the operating system:

```
SELECT @@VERSION
```

Result

```
Microsoft SQL Server  7.00 - 7.00.623 (Intel X86)
     Nov 27 1998 22:20:07
     Copyright (c) 1988-1998 Microsoft Corporation
     Standard Edition on Windows NT 4.0 (Build 1381: Service Pack 4)
```

Example 2

The following code segment uses the @@TRANCOUNT function to determine whether any transactions are open before a query or stored procedure is executed. The @@TRANCOUNT function returns a number that represents the number of transactions that you currently have open.

```
IF (@@TRANCOUNT > 0)
BEGIN
RAISERROR('Task cannot be executed within a transaction.',
10, 1)
RETURN
END
```

Example 3

The following code segment updates an employee's last name. When you change a last name, you may want to confirm that there is a qualifying row to update. If a qualifying row is not found, you would want to be notified that the change did not take place and then take appropriate action. The @@ROWCOUNT function returns a value that represents the number of rows affected by the last executed query.

```
USE Northwind
UPDATE Employees SET LastName = 'Brooke'
     WHERE LastName = 'Brook'
IF (@@ROWCOUNT = 0)
     BEGIN
          PRINT 'Warning: No rows were updated'
          RETURN
     END
```

Result

```
Warning: No rows were updated
```

Functions are commonly used when converting date data from the format of one country to that of another.

To change date formats, use the CONVERT function with the style option to determine the date format that will be returned. Go to the "CONVERT" topic in SQL Server Books Online for a full listing of the date style options.

Example 4

This example demonstrates how you can use CONVERT to convert a datetime value returned by the GETDATE function to a varchar string using different styles. The GETDATE function returns the current date and time on the server.

```
SELECT 'ANSI:', CONVERT(varchar(30), GETDATE(), 102) AS Style
UNION
SELECT 'Japanese:', CONVERT(varchar(30), GETDATE(), 111)
UNION
SELECT 'European:', CONVERT(varchar(30), GETDATE(), 113)
```

Result

```
          Style
--------- ------------------------------
European: 25 Jan 1999 19:29:44:893
Japanese: 1999/01/25
ANSI:     1999.01.25

(3 row(s) affected)
```

Example 5

This example returns the current users login name and the name of the application that they are using for the current session or connection.

```
SELECT SUSER_SNAME(), APP_NAME()
```

Result

```
---------------------- ----------------------
STUDYSQL\Administrator  MS SQL Query Analyzer

(1 row(s) affected)
```

Example 6

This example determines whether the firstname column in the member table of the library database allows null values.

A result of zero (false) means that null values are not allowed, and a result of one (true) means that null values are allowed. Notice that the OBJECT_ID function is embedded in the COLUMNPROPERTY function. This function returns the internal SQL Server object ID of the member table in the database. The first argument of the COLUMNOBJECT function requires an object ID rather than an object name.

```
USE Northwind
SELECT COLUMNPROPERTY(OBJECT_ID('Products'),
'ProductName', 'AllowsNull')
```

Result

```
-----------
0

(1 row(s) affected)
```

The CASE function evaluates a list of conditions and returns the value of a result expression that corresponds to the selected condition. The conditions can either be "when expressions" that are compared to an input expression or Boolean expressions. You can use a CASE function in any expression.

Syntax 1

```
CASE input_expression

WHEN when_expression THEN result_expression

     […n]

     [ELSE else_result_expression]

END
```

Syntax 2

```
CASE

    WHEN Boolean_expression THEN result_expression

     […n]

     [ELSE else_result_expression]

END
```

Example 7

The following example reviews the inventory status of products in the Products table using the Boolean form of the CASE function and returns messages based on the quantities available, quantities backordered, and whether the product has been discontinued:

```
SELECT ProductID, 'Product Inventory Status' =
CASE
     WHEN (UnitsInStock < UnitsOnOrder AND Discontinued = 0)
          THEN 'Negative Inventory - Order Now!'
     WHEN ((UnitsInStock-UnitsOnOrder) < ReorderLevel AND
```

```
            Discontinued = 0)
          THEN 'Reorder level reached- Place Order'
     WHEN (Discontinued = 1) THEN '***Discontinued***'
     ELSE 'In Stock'
END
FROM Northwind..Products
```

Result

```
ProductID   Product Inventory Status
----------- ----------------------------------
1           In Stock
2           Negative Inventory - Order Now!
3           Negative Inventory - Order Now!
4           In Stock
5           ***Discontinued***
...
76          In Stock
77          In Stock

(77 row(s) affected)
```

- **To determine the server process ID**

In this exercise, you will observe current server activity and determine the activity that your session is generating.

1. Execute the sp_who system stored procedure.

   ```
   EXEC sp_who
   ```

   SQL Server displays all activity that is occurring on the server.

2. To determine which activity is yours, execute the following statement:

   ```
   SELECT @@spid
   ```

   The server process ID (spid) number of your process is returned in the results.

3. Execute the sp_who system stored procedure again, using your spid number as an additional parameter. (In the following statement, *n* represents your spid number.)

   EXEC sp_who n

Only the activity related to your spid is displayed.

- **To retrieve environmental information**

In this exercise, you will determine which version of SQL Server you are running and you will retrieve connection, database context, server, and error information. You will perform these tasks using system functions. This exercise also demonstrates the use of the CASE function.

1. Execute the following statement:

```
SELECT @@version
```

2. Execute the following statement:

```
SELECT SUSER_SNAME(), DB_NAME(), @@servername
```

3. Execute the following statement:

```
SELECT 3 / 0
PRINT CASE @@error
        WHEN 8134 THEN 'You tried to divide by zero!!'
        WHEN 0    THEN 'No error occurred.'
        ELSE 'An unknown error occurred'
     END
```

   Execute the statement again after changing the zero in the SELECT to 3.

Control of Flow Language Elements

Transact-SQL contains several language elements that control the flow of execution of statements in scripts, batches, and stored procedures.

BEGIN … END Blocks

The BEGIN … END blocks enclose a series of Transact-SQL statements so that they are treated as a unit called a *statement block*.

Syntax

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

IF … ELSE Blocks

The IF statement contains a condition. If the condition is true, then the statement or statement block following the IF keyword is executed; otherwise, the statement or statement block following the ELSE keyword is executed. The ELSE keyword is optional.

Syntax

```
IF Boolean_expression

   {sql_statement | statement_block}

[ELSE

   {sql_statement | statement_block}]
```

Example

This example determines whether a customer has any orders before deleting the customer from the customer list.

```
USE Northwind
IF EXISTS (SELECT OrderID FROM Orders
               WHERE CustomerID = 'frank')
     PRINT '*** Customer cannot be deleted ***'
ELSE
     BEGIN
          DELETE Customers WHERE CustomerID = 'frank'
          PRINT '*** Customer deleted ***'
     END
```

WHILE Constructs

The *WHILE* constructs execute a statement or statement block repeatedly as long as the specified condition is true. BREAK and CONTINUE statements inside a statement block control the operation of the statements inside a WHILE loop. A *BREAK* statement transfers execution to the first statement following the END at the bottom of the loop. A *CONTINUE* statement transfers execution back to the BEGIN at the top of the loop, skipping any statements following the CONTINUE statement.

Syntax

```
WHILE Boolean_expression

    {sql_statement | statement_block}

For WHILE loops, the statement block takes the form

BEGIN
```

```
     {sql_statement | statement_block}

     [BREAK | CONTINUE | sql_statement | statement_block ]

     […n]

END
```

Example

This example creates a temporary table and then declares a local variable, which is used as a counter for inserting rows with incrementing values in a WHILE loop. The BREAK statement terminates the loop before the counter reaches its maximum value.

```
CREATE TABLE #Test
(Counter int,
 DateCreated datetime)
GO
DECLARE @counter int
SET @counter = 1
WHILE @counter <= 10
     BEGIN
          INSERT #Test VALUES (@counter, GETDATE())
          IF (SELECT COUNT(*) FROM #Test) > 8
               BEGIN
                    PRINT 'Maximum number of rows reached'
                    BREAK
               END
          SET @counter = @counter + 1
     END
```

RETURN Statement

The *RETURN* statement exits unconditionally from a query or procedure. RETURN is immediate and can be used at any point to exit from a procedure, batch, or statement block. Statements following RETURN are not executed.

Syntax

```
RETURN [integer_expression]
```

where the integer expression is an optional return value. System stored procedures return a value of 0, unless otherwise specified. A nonzero return value is usually used to indicate an error.

Example

In this example, RETURN is used to exit if no products with a UnitPrice greater than 30 are found.

```
USE Northwind
SELECT ProductName, UnitPrice FROM Products
WHERE UnitPrice > 30
IF @@ROWCOUNT = 0
     RETURN
. . . code to process the selected rows.
```

Comments

Comments are nonexecuting strings of text placed between statements to describe the action that a statement is performing or to disable one or more statements. Comments can be used in one of two ways—in line with a statement or as a block.

In-Line Comments

You can create in-line comments using two hyphens (--) to set a comment apart from a statement. Transact-SQL ignores text to the right of the comment characters. An in-line comment can be used on the same line as a statement, following the statement or at the beginning of a line to comment the whole line. If more than one line must be commented, you must repeat the hyphens (--) on each line.

Example 1

This example uses an in line comment to explain what a calculation is doing.

```
SELECT ProductName
     , (UnitsInStock - UnitsOnOrder) -- Calculates inventory
     , SupplierID
FROM Northwind..Products
```

Example 2

This example uses an in line comment to prevent the execution of a statement section.

```
SELECT ProductName
     , (UnitsInStock - UnitsOnOrder) -- Calculates inventory
-- , SupplierID
FROM Northwind..Products
```

Block Comments

You create multiple line comments by placing the opening comment pair—forward slash, asterisk (/*)—at the start of the comment text and then concluding the comment with a closing comment pair— asterisk forward and slash (*/).

Use multiple line comment pairs to create one or more lines of comments or comment headers—descriptive text that documents the statements that follow it. Comment headers often include the author's name, date of the creation of the script and last modification, version information, and a description of the action that the statement performs.

Example 3

This example shows a comment header that spans several lines. The two asterisks preceding each line improve readability but are not required.

```
/*
** This code retrieves all rows of the Products table
** and displays the unit price, the unit price increased
** by 10 percent and the name of the product.
*/
SELECT UnitPrice, (UnitPrice * 1.1), ProductName
FROM Northwind..Products
```

TIP

You should place comments throughout a script to describe the actions that the statements are performing. This placement is especially important if others must review or implement the script.

Example 4

This section of a script is commented to prevent it from executing. This prevention can be helpful when debugging or troubleshooting a script file.

```
/*
DECLARE @v1 int
SET @v1 = 0
WHILE @v1 < 100
   BEGIN
     SELECT @v1 = (@v1 + 1)
     SELECT @v1
   END
*/
```

Creating and Dropping Databases

The CREATE DATABASE and DROP DATABASE statements are used to create and remove databases. In this lesson, you will learn how to use these statements.

Creating Databases

You can create a database using the Database Creation wizard, SQL Server Enterprise Manager, or the CREATE DATABASE statement. The process of creating a database also creates a transaction log for that database.

Creating a database is a process of specifying the name of the database and designating the size and location of the database files. When the new database is created, it is a duplicate of the model database. Any options or settings in the model database are copied into the new database. New databases can be created by default by members of the *sysadmin* and *dbcreator* fixed server roles; however, permissions to create databases can be given to other users.

TIP

Information about each database in SQL Server is stored in the sysdatabases system table in the master database. Therefore, you should back up the master database each time you create, modify, or drop a database.

Syntax

NOTE

The CREATE DATABASE syntax has changed significantly from SQL Server version 6.5. SQL Server version 7.0 does not use devices, so it is no longer necessary to create a device, using the DISK INIT statement, before creating a database. You now specify file information as part of the CREATE DATABASE statement.

```
CREATE DATABASE database_name
[ON
     { [PRIMARY] (NAME = logical_file_name,
          FILENAME =  ' os_file_name'
          [,  SIZE =  size]
          [,  MAXSIZE =  max_size]
          [,  FILEGROWTH =  growth_increment] )
     } [, ...n]
]
[LOG ON
     { ( NAME = logical_file_name,           FILENAME =  '
os_file_name'
          [,  SIZE =  size] )
     } [, ...n]
]
[FOR RESTORE]
```

When you create a database, you can set the following options:

- PRIMARY. This option specifies the files in the primary filegroup. The primary filegroup contains all the database system tables. It also contains all objects not

assigned to user filegroups (covered later in this chapter). A database can have only one primary data file. The primary data file is the starting point of the database and points to the rest of the files in the database. Every database has one primary data file. The recommended file extension for a primary data file is .mdf. If the PRIMARY keyword is not specified, the first file listed in the statement becomes the primary data file.

- FILENAME. This option specifies the operating system file name and path for the file. The path in the os_file_name must specify a folder on a local hard drive of the server on which SQL Server is installed.

  NOTE

  If your computer has more than one disk and you are not using RAID (Redundant Array of Inexpensive Disks), place the data file and transaction log file on separate physical disks. This separation increases performance and can be used, in conjunction with disk mirroring, to decrease the likelihood of data loss in case of media failure.

- SIZE. This option specifies the size of the file. You can specify sizes in megabytes using the MB suffix (the default), or kilobytes using the KB suffix. The minimum value is 512 KB. If size is not specified for the primary data file, it defaults to the size of the model database primary data file. If no log file is specified, a default log file that is 25 percent of the total size of the data files is created. If size is not specified for secondary data files or log files, it defaults to a size of 1 MB.

  The SIZE option sets the minimum size of the file. The file can grow larger but cannot shrink smaller than this size. To reduce the minimum size of a file, use the DBCC SHRINKFILE statement.

  NOTE

  SQL Server 7.0 has a maintenance wizard that can be set to control the size of databases.

- MAXSIZE. This option specifies the maximum size to which the file can grow. You can specify sizes in megabytes using the MB suffix (the default), or kilobytes using the KB suffix. If no size is specified, the file grows until the disk is full.
- FILEGROWTH. This option specifies the growth increment of the file. When SQL Server needs to increase the size of a file, it will increase the file by the amount specified by the FILEGROWTH option. A value of 0 indicates no growth. The value can be specified in megabytes (the default), in kilobytes, or as a

percentage (%). The default value, if FILEGROWTH is not specified, is 10 percent, and the minimum value is 64 KB. The specified size is rounded to the nearest 64 KB.

Example

The following example creates a database called *sample* with a 10-MB primary data file and a 3-MB log file. The primary data file can grow to a maximum of 15 MB and it will grow in 20 percent increments; for example, it would grow by 2 MB the first time that it needed to grow. The log file can grow to a maximum of 5 MB and it will grow in 1-MB increments.

```
CREATE DATABASE sample
ON
  PRIMARY ( NAME=sample_data,
  FILENAME='c:\mssql7\data\sample.mdf',
  SIZE=10MB,
  MAXSIZE=15MB,
  FILEGROWTH=20%)
LOG ON
  ( NAME=sample_log,
  FILENAME='c:\mssql7\data\sample.ldf',
  SIZE=3MB,
  MAXSIZE=5MB,
  FILEGROWTH=1MB)
```

- **To create a database**

In this exercise, use the CREATE DATABASE statement to create a database called *sample_db* based on the information in the following table:

File	NameFile name	Initial size	File growth	Maximum file size
Database	sample_datac:\mssql7\data\sample.mdf	2 MB	20 %	15 MB
Log sample	_logc\ mssql7\data\sample.ldf	1 MB	1 MB	5 MB

1. Open SQL Server Query Analyzer and log in to the (local) server with Microsoft Windows NT authentication. Your account is a member of the Windows NT Administrators group, which is automatically mapped to the SQL Server sysadmin role.
2. Execute the CREATE DATABASE statement to create the database.

```
CREATE DATABASE sample_db
ON
```

```
  PRIMARY (NAME=sample_data,
  FILENAME='c:\mssql7\data\sample.mdf',
  SIZE=2MB,
  MAXSIZE=15MB,
  FILEGROWTH=20%)
LOG ON
  (NAME=sample_log,
  FILENAME='c:\mssql7\data\sample.ldf',
  SIZE=1MB,
  MAXSIZE=5MB,
  FILEGROWTH=1MB)
```

3. View the database propertiesin sp_helpdb to verify that the database has been created properly.

```
EXEC sp_helpdb sample_db
```

Dropping a Database

You can drop a database when you no longer need it. When you drop a database, you permanently delete the database and the disk files used by the database. Permission to drop a database defaults to the database owner and members of the *sysadmin* fixed server role. Permission to drop a database cannot be transferred.

You can drop databases using SQL Server Enterprise Manager or by executing the DROP DATABASE statement.

Syntax

```
DROP DATABASE database_name [,…n]
```

Example

This example drops multiple databases by using one statement.

```
DROP DATABASE mydb1, mydb2
```

When you drop a database, consider the following facts and guidelines:

- With SQL Server Enterprise Manager, you can drop only one database at a time.
- With Transact-SQL, you can drop several databases at one time.
- After you drop a database, login IDs for which that particular database was the default database will not have a default database.
- Back up the master database after you drop a database.

- SQL Server does not let you drop master, model, and tempdb databases but does allow you to drop the msdb system database.

Restrictions on Dropping a Database

The following restrictions apply to dropping databases. You cannot drop a database that is:

- In the process of being restored.
- Open for reading or writing by any user.
- Publishing any of its tables as part of SQL Server replication.
- Although SQL Server allows you to drop the msdb system database, you should not drop it if you use or intend to use any of the following:
- SQL Server Agent
- Replication
- SQL Server Web wizard
- Data Transformation Services (DTS)

- **To delete a database**

In this exercise, you will use Transact-SQL statements to drop the sample_db database that you created in a previous exercise.

Open or switch to SQL Server Query Analyzer.

1. Execute the DROP DATABASE statement to delete the sample_db database.

```
DROP DATABASE sample_db
```

2. Execute a system stored procedure to generate a list of databases. Verify that you have deleted the sample_db database.

```
EXEC sp_helpdb
```

TIP

After deleting a database in SQL Server Query Analyzer, it may still show in SQL Server Enterprise Manager. This is because Enterprise Manager does not automatically refresh information that has been changed by other connections. You will need to use the Refresh option, which is available from the right-click shortcut menu on the various folders, to see updated information.

Creating Data Types

Before you can create a table, you must define the data types for the table. Data types specify the type of information (characters, numbers, or dates) that a column can hold, as

well as how the data is stored. Microsoft SQL Server supplies various system data types. SQL Server also allows user-defined data types that are based on system data types.

System-Supplied Data Types

SQL Server provides several different data types. Certain types of data have several associated SQL Server system supplied data types. For example, you could use the int, decimal, or float data type to store numeric data. However, you should choose appropriate data types in order to optimize performance and conserve disk space.

Categories of System-Supplied Data Types

The following table maps common types of data to SQL Server system-supplied data types. The table includes data type synonyms for ANSI compatibility.

Type of data	System-supplied data types	ANSI synonym	Number of bytes
Binary	binary[(n)] varbinary[(n)]	- binary varying[(n)]	1-8000
Character	char[(n)] varchar[(n)]	character[(n)] char[acter] varying[(n)]	1-8000 (8000 characters)
Unicode character	nchar[(n)] nvarchar[(n)]	national char[acter][(n)] national char[acter] varying[(n)]	2-8000 (1 - 4000 characters)
Date and time	Datetime, smalldatetime	-	8 (2 4-byte integers) 4 (2 2-byte integers)
Exact numeric	decimal[(p[, s])] numeric[(p[, s])]	dec	5-17
Approximate numeric	float[(n)] real	Double precision or float[(n)] float[(n)]	4-8 4
Global identifier	uniqueidentifier	-	16
Integer	int smallint, tinyint	integer -	4 2, 1
Monetary	money, smallmoney	-	8, 4

Special	bit, cursor, sysname, timestamp,	-	1, 08
Text and image	text, image	-	0-2 GB
Unicode text	ntext	national text	0-2 GB

NOTE

SQL Server supports multiple languages with the nchar, nvarchar, and ntext Unicode string data types. Unicode strings use two bytes per character.

Exact and Approximate Numeric Data

The terms, exact and approximate numeric data, can be confusing because, in practice, they are often used differently than their names imply.

Exact Numeric Data Types

Exact numeric data types let you specify *exactly* the scale and precision to use. For example, you can specify three digits to the right of the decimal and four to the left. A query always returns exactly what you entered. SQL Server supports exact numeric data with two synonymous data types, *decimal* and *numeric*, for ANSI compatibility.

Most of the time, you would use exact numeric data types for financial applications in which you want to portray the data consistently (always two decimal places) and to query on that column (for example, to find all loans with an interest rate of 8.75 percent).

Approximate Numeric Data Types

Approximate numeric data types store data as accurately as is possible using binary numbers. For example, the fraction 1/3 is represented in a decimal system as .33333 (repeating). The number cannot be stored accurately, so an approximation is stored.

The approximate numeric data types, *real* and *float*, are unable to store all decimal numbers with complete accuracy. Therefore, it is necessary to store an approximation of the decimal number.

Most of the time, you would use approximate numeric data types for scientific or engineering applications.

TIP

Avoid referencing columns with the float or real data types in WHERE clauses. Calculations performed on float and real decimal values are likely to have small rounding errors due to the approximate representation of decimal numbers.

Creating and Dropping User-Defined Data Types

User-defined data types are based on system-supplied data types. They allow you to refine data types further to ensure consistency when working with common data elements in different tables or databases. User-defined data types do not allow you to define structures or complex data types. A user defined data type is defined for a specific database.

NOTE

User-defined data types that you create in the model database are automatically included in all databases that are subsequently created. Each user-defined data type is added as a row in the systypes table.

You can create and drop user-defined data types with system stored procedures. Data type names must follow the rules for identifier names and must be unique to each database. Define each user-defined data type in terms of a system-supplied data type. Specify a default null type of NULL or NOT NULL, which indicates whether the user-defined data type may or may not store nulls. The default null type can be overridden when the user-defined data type is used in a column definition.

Creating a User-Defined Data Type

The *sp_addtype* system stored procedure creates user-defined data types.

Syntax

```
sp_addtype type, system_data_type [,'NULL' | 'NOT NULL']
```

Example

The following example creates three user-defined data types:

```
EXEC  sp_addtype isbn, 'smallint', NOT NULL
EXEC  sp_addtype zipcode, 'char(10)', NULL
EXEC  sp_addtype longstring, 'varchar(63)', NULL
```

Dropping a User-Defined Data Type

The *sp_droptype* system stored procedure deletes user-defined data types from the systypes system table. A user-defined data type cannot be dropped if it is referenced by tables or other database objects.

Syntax

```
sp_droptype type
```

Example

The following example drops a user-defined data type:

```
EXEC sp_droptype isbn
```

Execute the sp_help system stored procedure or query the information_schema.domains view to retrieve a list of user-defined data types for the current database.

- **To create user-defined data types**

In this exercise, you will write and execute statements that create user-defined data types in the library database.

1. Log on your computer as Administrator or another account that is a member of the Administrators group.
2. Open SQL Server Query Analyzer and log on the (local) server with Microsoft Windows NT authentication. Your account is a member of the Windows NT Administrators group, which is automatically mapped to the SQL Server sysadmin role.
3. From the DB list box, click library.
4. Write and execute statements to create the user-defined data types described in the following table:

Data type	Description of data
Zipcode	Character data of 10 characters
Member_no	A whole number that will not exceed 30,000
Phonenumber	Character data of 13 characters that allows NULL
Shortstring	Variable character of up to 15 characters

5. The following statements are used to create these user-defined data types.

```
EXEC sp_addtype zipcode, 'char(10)'
EXEC sp_addtype member_no, 'smallint'
EXEC sp_addtype phonenumber, 'char(13)', NULL
EXEC sp_addtype shortstring, 'varchar(15)'
```

6. Type and execute the following statement to verify that the data type was created:

```
SELECT domain_name, data_type, character_maximum_length
  FROM information_schema.domains
  ORDER BY domain_name
```

Guidelines for Creating User-Defined Data Types

Consider the following guidelines for creating user-defined data types and balancing storage size with requirements:

- If column length varies, use one of the variable data types. For example, if you have a list of names, you can set it to varchar instead of char (fixed). Using a variable data type will limit space usage.
- Integer, date and time, and monetary data all support different ranges based on storage size. For example, if you use the tinyint data type for a store identifier in a database, you will have prblems when you decide to open store number 256.
- For numeric data types, the size and required level of precision helps to determine your choice. In general, use decimal.
- If the storage is greater than 8000 bytes, use the text or image data type. If it is less than 8000, use binary, char, or varchar. When possible, it is best to use char or varchar because they have more functionality than text and image.

Creating Tables

After you define all the data types for your table, you can create tables, add and drop columns, and generate column values.

Creating and Dropping a Table

When you create a table, you must specify the table name, column names, and column data types. Column names must be unique to a specific table, but you can use the same column name in different tables within the same database. You must name a data type for each column. Other options can also be specified for columns.

Creating a Table

Consider the following facts when you create tables in SQL Server. You can have up to:

- Two billion tables per database
- 1024 columns per table
- 8060 bytes per row (image and text data types each use 16 bytes per row)

Specifying NULL or NOT NULL

You can specify in the table definition whether to allow null values in each column. If you do not specify NULL or NOT NULL, SQL Server determines whether the column

may or may not accept null values based on the session- or database-level default. However, these defaults can change, so do not rely on them. NOT NULL is the SQL Server default; NULL is the ANSI default

Partial Syntax

```
CREATE TABLE table_name
     (column_name data_type  [NULL | NOT NULL]
     [,...n])
```

TIP

To view table properties, right-click a table in Microsoft SQL Server Enterprise Manager or execute the sp_help system stored procedure, passing the table name as an argument.

Example

The following example creates the member table, specifying the columns of the table, a data type for each column, and whether that column allows null values:

```
 CREATE TABLE member
(member_no     member_no   NOT NULL,
 lastname      shortstring NOT NULL,
 firstname     shortstring NOT NULL,
 middleinitial letter      NULL,
 photograph    image       NULL)
```

Dropping a Table

Dropping a table removes the table definition and all the data, as well as the permission specifications for that table.

Before you can drop a table, you must remove any dependencies between the table and other objects. To view existing dependencies, execute the sp_depends system stored procedure.

You must drop any view on the dropped table, explicitly using the DROP VIEW statement.

Syntax

```
DROP TABLE table_name
```

Generating Column Values

SQL Server allows you to automatically generate column values, using the IDENTITY property or the NEWID function with the uniqueidentifier data type.

Using the IDENTITY Property

You can use the *IDENTITY property* to create columns (referred to as *identity columns*) that contain system-generated sequential numeric values, which identify each row inserted into a table. An identity column is often used for primary key values.

Having SQL Server automatically provide key values rather than having the client application provide key values can improve performance. It simplifies programming, keeps primary key values short, and reduces user-transaction bottlenecks.

Partial Syntax

```
CREATE TABLE table
(column_name numeric_data_type
 IDENTITY [(seed, increment)] [NOT NULL])
```

Consider the following facts about the IDENTITY property and retrieving information about it:

- Only one identity column is allowed per table.
- An identity column cannot be updated.
- An identity column does not allow null values.
- An identity column must be used with the integer (int, smallint, or tinyint), numeric, or decimal data types. The numeric and decimal data types must be specified with a scale of 0. When you determine what data type to use in defining the column, estimate the number of rows that the table will contain.
- Two system functions return information about an identity column definition: *IDENT_SEED* (returns the seed, or starting, value) and *IDENT_INCR* (returns the increment value).
- You can use the @@IDENTITY function to retrieve the identity value of the last row inserted during a session.
- You can use the IDENTITYCOL keyword in place of the column name of the identity column when querying any table that has an identity column.

TIP

The IDENTITY property does not enforce uniqueness. To enforce uniqueness, create a unique index.

Example

This example creates the class table with two columns, student_id and name. The IDENTITY property is used to increment the value of the student_id column

automatically for each row added to the table. The seed is set to 100, and the increment value is 5. The values in the column would be 100, 105, 110, 115, and so on. Using 5 as an increment value allows you to insert records between the values at a later time.

```
 CREATE TABLE class
(student_id INT IDENTITY(100, 5) NOT NULL,
 name       VARCHAR(16))
```

Using the NEWID Function and the uniqueidentifier Data Type

The uniqueidentifier data type and the NEWID function are used together, in the following ways:

- The uniqueidentifier data type stores a unique identification number as a 16-byte (128-bit) binary value. This data type is used for storing a globally unique identifier (GUID).
- The NEWID function creates a unique identifier (GUID) that can be stored using the uniqueidentifier data type. The same GUID will never be created again so it uniquely identifies a row across tables, databases, servers, and organizations. Here is an example of a GUID:

  ```
  24F1B644-9DD7-11D2-993C-204C4F4F5020
  ```

Example

In this example, the customer table cust_ ID column is created with the uniqueidentifier data type, using a default value generated by the NEWID function. A unique value for the cust_id column will be generated for each new row.

```
 CREATE TABLE customer
(cust_id   uniqueidentifier NOT NULL DEFAULT NEWID(),
 cust_name char(30)         NOT NULL)
```

- **To create a table using statements**

In this exercise, you will write and execute a statement that creates a table in the library database.

At this time, do not create the primary and foreign keys, indexes, or other items listed in the library schema. Make sure you specify NULL or NOT NULL for each column.

1. Verify that you are using the library database.
2. Write and execute a statement to create the member table, defining columns as specified in the following table:

Column name	Data type	Allows NULLs?	IDENTITY property?
member_no	member_no	No	Seed = 1 Increment = 1
lastname	shortstring	No	No
firstname	shortstring	No	No
middleinitial	leter	Yes	No
photograph	image	Yes	No

```
 CREATE TABLE member
(member_no     member_no   NOT NULL IDENTITY(1,1),
 lastname      shortstring NOT NULL,
 firstname     shortstring NOT NULL,
 middleinitial letter      NULL,
 photograph    image       NULL)
```

4. Which data types are user-defined?
5. Answer

- **To create a table using SQL Server Enterprise Manager**

In this exercise, you will use SQL Server Enterprise Manager to create a table in the library database. At this time, do not create the primary and foreign keys, indexes, or other items listed in the library schema. Make sure Allow Nulls is checked for the phone_no column only.

1. Open SQL Server Enterprise Manager.
2. Expand your server group; then expand your server.
3. Expand Databases; then expand the library database.
4. Right-click Tables; then click New Table.
5. In the Choose Name dialog box, type the name **adult** for the table. Click OK.
6. Fill in the columns as specified in the following table. Each row represents one column in the table.

Column name	Data type	Allows NULLs?
member_no	member_no	No
street	shortstring	No
city	shortstring	No
state	statecode	No
zip	zipcode	No
phone_no	phonenumber	Yes

expr_date	datetime	No

7. Close the New Table window and save changes to the adult table.

Adding, Dropping, and Altering a Column

Adding a Column

The type of information that you specify when you add a column is similar to that which you supply when you create a table.

Partial Syntax

```
ALTER TABLE table
     ADD column_name data_type [NULL | NOT NULL] [,...n]
```

Example

This example adds a column that allows null values.

```
ALTER TABLE sales
 ADD commission money NULL
```

Dropping a Column

Dropped columns are unrecoverable. Therefore, be certain that you want to remove a column before doing so.

Partial Syntax

```
ALTER TABLE table
  DROP COLUMN column_name
```

Example

This example drops a column from a table.

```
ALTER TABLE sales
 DROP COLUMN customer_name
```

TIP

All indexes and constraints that are based on a column must be removed before you drop the column.

Altering a Column

You can make changes to a column, such as changing it to use a different data type. See ALTER TABLE in Books Online for restrictions on changing columns.

Partial Syntax

```
ALTER TABLE table
     ALTER COLUMN column_name new_data_type [NULL | NOT NULL]
```

Example

This example changes a column to use a larger numeric data type.

```
ALTER TABLE stores
ALTER COLUMN store_id smallint
```

- **To add a column to a table**

In this exercise, you will write and execute a statement that adds a column to the adult table in the library database.

1. Verify that you are using the library database.
2. Write and execute a statement to add a column to the adult table. The column should be named *age* (use the tinyint data type) and allow null values.

   ```
   ALTER TABLE adult
     ADD age tinyint NULL
   ```

3. Execute the sp_help system stored procedure on the adult table to verify that the age column was defined as you specified.

   ```
   EXEC sp_help adult
   ```

- **To drop a column from a table**

In this exercise, you will write and execute a statement that drops the age column from the adult table in the library database.

Verify that you are using the library database.

1. Write and execute a statement that drops the age column from the adult table.

   ```
   ALTER TABLE adult
   ```

```
DROP COLUMN age
```

2. Execute the sp_help system stored procedure on the adult table to verify that the age column was dropped.

```
EXEC sp_help adult
```

Using Constraints

Constraints are the preferred method of enforcing data integrity.

Determining Which Type of Constraint to Use

Constraints are an ANSI-standard method of enforcing data integrity. Each type of data integrity domain, entity, and referential is enforced with separate types of constraints. Constraints ensure that valid data values are entered in columns and those relationships are maintained between tables.

The following table describes the different types of constraints:

Type of integrity	Constraint type	Description
Domain	DEFAULT	Specifies the value that will be provided for the column when a value has not been supplied explicitly in an INSERT statement.
	CHECK	Specifies a validity rule for the data values in a column.
	FOREIGN KEY	Values in the foreign key column(s) must match values in the primary key column(s) of the referenced table.
Entity	PRIMARY KEY	Uniquely identifies each row—ensures that users do not enter duplicate values and that an index is created to enhance performance. Null values are not allowed.
	UNIQUE	Prevents duplicates in non-primary keys and ensures that an index is created to enhance performance. Null values are allowed.
Referential	FOREIGN KEY	Defines a column or combination of columns whose values match the primary key of either the same or another table.
User-defined	CHECK	Specifies a validity rule for the data values in a

column.

Defining Constraints

You define constraints by using the CREATE TABLE or ALTER TABLE statement.

You can add constraints to a table that has existing data, and you can place constraints on single or multiple columns. When adding single or multiple column constraints, use the following criteria:

- If the constraint is defined on a single column, it can reference only that column and it is called a *column-level constraint*.
- If a constraint is not defined on a single column, it can reference multiple columns and it is called a *table level constraint*.
- DEFAULT constraints must be column-level.
- Column-level CHECK constraints can reference only a single column.
- Table-level CHECK constraints can reference all columns in the table.
- Constraints cannot reference columns in other tables.

Partial Syntax

```
CREATE TABLE table_name
(     { <column_definition>
      | <table_constraint> }
      [,…n]

)

ALTER TABLE table_name
     ADD { <column_definition> } [,…n]
     | [ WITH CHECK | WITH NOCHECK ] ADD
          { <table_constraint>
          | [CONSTRAINT constraint_name]
               DEFAULT constant_expression FOR column
          } [,…n]
     | DROP { [CONSTRAINT] constraint_name } [,…n]
     | {CHECK | NOCHECK} CONSTRAINT
          {ALL | constraint_name[,...n]}
```

Where

```
<column_definition> ::= {column_name data_type}
     [ [CONSTRAINT constraint_name]
     { DEFAULT constant_expression
     | CHECK (logical_expression)
     | PRIMARY KEY [CLUSTERED | NON CLUSTERED]
     | UNIQUE [CLUSTERED | NON CLUSTERED]
     | [FOREIGN KEY] REFERENCES ref_table [(ref_column)]
```

```
    } ]
    [...n]

<table_constraint> ::=
    [CONSTRAINT constraint_name]
    { CHECK (logical_expression)
    | PRIMARY KEY [CLUSTERED | NON CLUSTERED] (column[,...n])
    | UNIQUE [CLUSTERED | NON CLUSTERED] (column[,...n])
    | FOREIGN KEY
        (column[,...n])
        REFERENCES ref_table [(ref_column[,...n])]
    }
```

Example

This example creates an authors table and defines column-level PRIMARY KEY and CHECK constraints on two of the columns in the table. The CHECK constraint ensures that the status for an author can only be 'CONTRACT' or 'EMPLOYEE'.

```
CREATE TABLE authors
(au_id     int      NOT NULL CONSTRAINT au_PK PRIMARY KEY,
 firstname char(30) NOT NULL,
 lastname  char(30) NOT NULL,
 status    char(10) NOT NULL
    CONSTRAINT cat_CHK CHECK (status IN ('CONTRACT', 'EMPLOYEE'))
)
```

Example

This example creates an employee table and defines table-level PRIMARY KEY and CHECK constraints on the table. The CHECK constraint ensures that the date the employee was employed is earlier than the date the employee was terminated.

```
CREATE TABLE employee
(emp_num    int      NOT NULL,
 lastname   char(30) NOT NULL,
 firstname  char(30) NOT NULL,
 employed   datetime NOT NULL,
 terminated datetime NULL,
 CONSTRAINT emp_PK   PRIMARY KEY (emp_num),
 CONSTRAINT date_CHK CHECK       (employed < terminated)
)
```

Considerations for Using Constraints

Consider the following facts when you implement or modify constraints:

- You can create, change, and drop constraints without having to drop and re-create a table.

- You must build error-handling logic into your client applications to test whether a constraint has been violated whenever data is modified on the server.
- By default, SQL Server verifies existing data when you add a constraint to a table.

You should specify names for constraints when you create them because SQL Server provides complicated, system-generated names. Names must be unique to the database object owner and follow the rules for SQL Server identifiers.

To obtain information about constraints, execute the sp_helpconstraint or sp_help system stored procedures or query information schema views, such as check_constraints, referential_constraints, table_constraints, constraint_column_usage, and constraint_table_usage.

The following system tables store constraint definitions: syscomments, sysreferences, and sysconstraints.

DEFAULT Constraints

A *DEFAULT constraint* enters a value in a column when one is not specified in an INSERT statement. DEFAULT constraints enforce domain integrity.

Example

This example adds a DEFAULT constraint that inserts the value 'Unknown' into the first name column if a value is not provided when a row is inserted into the adult table.

```
USE library
ALTER TABLE adult
ADD
CONSTRAINT firstname DEFAULT 'Unknown' FOR firstname
```

Consider the following facts when you apply a DEFAULT constraint:

- It only applies to INSERT statements.
- Only one DEFAULT constraint can be defined per column.
- It cannot be placed on columns with the IDENTITY property or on columns with the timestamp data type.
- You can use functions to supply default values. For example, you can use the SUSER_SNAME function to record which users insert data or the GETDATE function to record the date and time that data was inserted.

- **To define a DEFAULT constraint**

  In this exercise, you will execute a script that creates a default for the state column in the adult table, and then you will modify the same script to change the default state.

CHECK Constraints

A *CHECK constraint* restricts the data values that can be stored in one or more columns. A CHECK constraint specifies a logical expression that must be true in order for data to be accepted.

Example

This example adds a CHECK constraint to ensure that a phone number conforms to accepted phone number formatting.

```
USE library
ALTER TABLE adult
ADD CONSTRAINT phone_no
CHECK (phone_no LIKE '(212)[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]')
```

Consider the following facts when you apply a CHECK constraint:

- It verifies data every time that you execute an INSERT or UPDATE statement.
- It can reference other columns inthe same table. For example, a salary column could reference a value in a job_grade column.
- It cannot be placed on columns with the IDENTITY property or columns with the timestamp or uniqueidentifier data type.
- It cannot contain subqueries.

PRIMARY KEY Constraints

A *PRIMARY KEY constraint* defines a primary key on a table. The value in the primary key uniquely identifies each row in the table. This constraint enforces entity integrity.

Example

This example adds a constraint that specifies that the primary key value of the member table is the member number and indicates that a clustered index will be created to enforce the constraint.

```
USE library
ALTER TABLE member
ADD CONSTRAINT PK_member_member_no
PRIMARY KEY CLUSTERED (member_no)
```

Consider the following facts when you apply a PRIMARY KEY constraint:

- Only one PRIMARY KEY constraint can be defined per table.
- The primary key values must be unique.
- Null values are not allowed.

- You can specify whether a *clustered* or a *nonclustered index* is created (clustered is the default if it does not already exist). If a clustered index does exist, you must drop it first or specify that a nonclustered index be created.

- **To define a PRIMARY KEY constraint**

In this exercise, you first will execute a script that creates a primary key on the item table, and then you will write a statement to create a PRIMARY KEY constraint on the title table. Finally, you will execute a script that adds PRIMARY KEY constraints to the other tables in the library database.

UNIQUE Constraints

A *UNIQUE constraint* specifies that two rows in a column cannot have the same value. This constraint enforces entity integrity. A unique index is created automatically to support the constraint.

A UNIQUE constraint is helpful when you already have a primary key (such as an employee number) but you want to guarantee that other columns (such as an employee's driver's license number) are also unique.

Example

In a sample database, this example creates a UNIQUE constraint on the drivers license column in the employee table.

```
ALTER TABLE employee
ADD CONSTRAINT u_driver_lic_no
UNIQUE NONCLUSTERED (driver_lic_no)
```

Consider the following facts when you apply a UNIQUE constraint:

- It can allow null values.
- You can place multiple UNIQUE constraints on a table.
- You can apply the UNIQUE constraint to one or more columns that must have unique values but are not the primary key of a table.
- The UNIQUE constraint is enforced through the creation of a unique index on the specified column or columns.
- The UNIQUE constraint defaults to using a unique nonclustered index unless a clustered index is specified.

FOREIGN KEY Constraints

A *FOREIGN KEY constraint* enforces referential integrity. The FOREIGN KEY constraint defines a reference to the column(s) of a PRIMARY KEY constraint or a UNIQUE constraint of the either the same or another table.

Example 1

This example uses a FOREIGN KEY constraint to ensure that any juvenile member is associated with a valid adult member.

```
USE library
ALTER TABLE juvenile
ADD CONSTRAINT FK_adult_memberno
 FOREIGN KEY (adult_memberno)
 REFERENCES adult(member_no)
```

Consider the following facts and guidelines when you apply a FOREIGN KEY constraint:

- It provides single or multicolumn referential integrity. The number of columns and data types that are specified in the FOREIGN KEY statement must match the number of columns and data types in the REFERENCES clause. Column names do not need to match.
- Unlike PRIMARY KEY or UNIQUE constraints, FOREIGN KEY constraints do not create indexes automatically. Nevertheless, a FOREIGN KEY is a good candidate for indexing and you should consider adding an index on the FOREIGN KEY column(s) to improve join performance.
- To modify data in a table that has a FOREIGN KEY constraint defined on it, users must have SELECT or REFERENCES permissions on the tables that are referenced by the FOREIGN KEY constraint.
- You must use only the REFERENCES clause without the FOREIGN KEY clause when you reference a column in the same table.

Example 2

This example creates an employee table with a PRIMARY KEY constraint and a FOREIGN KEY constraint. The FOREIGN KEY constraint in this example references the employee table. This reference is made because an employee's manager is another employee.

```
USE library
CREATE TABLE employee
(emp_num int NOT NULL CONSTRAINT emp_id PRIMARY KEY,
 mgr_num int NOT NULL CONSTRAINT mgr_ref
   REFERENCES employee (emp_num))
```

- **To define a FOREIGN KEY constraint**

In this exercise, you will first execute a script that creates a FOREIGN KEY constraint on the adult table, and then you will write a statement to create a FOREIGN KEY constraint on the item table. Finally, you will execute a script that adds FOREIGN KEY constraints to the other tables in the library database.

At the end of the script, the sp_help system stored procedure is executed to show information about the member table. The procedure shows that the member table is now referenced by the adult table.

You may notice an extra index, with a long automatically generated name, in the list of indexes for the member table. This is not really an index but the statistics for the member_ident index.

Disabling Data Checking When Adding Constraints

When you define a constraint on a table that already contains data, SQL Server checks the data automatically to verify that it meets the constraint requirements. However, you can disable constraint checking of existing data, if you do so at the time that the constraint is added to the table.

Consider the following guidelines for disabling constraint checking on existing data:

- You can disable checking only when adding CHECK and FOREIGN KEY constraints. Data is always checked when PRIMARY KEY and UNIQUE constraints are added.
- To disable constraint checking when you add a CHECK or FOREIGN KEY constraint to a table with existing data, include the WITH NOCHECK option in the ALTER TABLE statement.
- Use the WITH NOCHECK option only if existing data is known to conform to the new constraint or if existing data will never be updated.

NOTE

Data modifications that are made after you add the constraint must satisfy the constraint even if the columns used in the constraint are not modified. For example, if you add a CHECK constraint to ColumnA and specify the WITH NOCHECK option, the existing values in ColumnA are not checked. If you later update the value of ColumnB and the existing value of ColumnA for the row being updated does not meet the CHECK constraint requirements, the update fails.

Example

In this example, you add a CHECK constraint that verifies that all amounts entered in the sales column are greater than 0. Existing data is not checked when the constraint is added.

```
USE library
ALTER TABLE sales
WITH NOCHECK
```

```
ADD CONSTRAINT correct_amount
CHECK (amount >=0)
```

Disabling Constraint Checking

Constraint checking can be disabled for existing CHECK and FOREIGN KEY constraints so that any data that you modify or add to the table is not checked against the constraint.

To avoid the overhead of constraint checking, you might want to disable constraints when:

- You are loading a large amount of data that is known to conform to the constraints.
- You want to load a large amount of data that does not conform to the constraints. Later, you can execute queries to change the data and then re-enable the constraints.

To disable a constraint, use the ALTER TABLE statement and the NOCHECK CONSTRAINT clause. To enable a disabled constraint, use the ALTER TABLE statement and the CHECK CONSTRAINT clause. You can disable or enable all the constraints on a table with the ALTER TABLE statement and either the NOCHECK CONSTRAINT ALL or the CHECK CONSTRAINT ALL clause.

Example

This example disables the correct_amount constraint on the sales table.

```
USE library
ALTER TABLE sales
NOCHECK CONSTRAINT correct_amount
```

The constraint can be re-enabled by executing another ALTER TABLE statement with the CHECK CONSTRAINT clause.

```
USE library
ALTER TABLE sales
CHECK CONSTRAINT correct_amount
```

To determine whether a constraint is enabled or disabled on a table, execute the sp_help *table_name* or the sp_helpconstraint*table_name* system stored procedure.

Combining Data from Multiple Tables

A *join* is an operation that allows you to query two or more tables to produce a single result set that incorporates rows and columns from each table. You join tables based on columns in each table that contain common data values. Joins allow you to put back together the data that you break apart during normalization.

There are three types of joins: inner joins, outer joins, and cross joins. Additionally, you can join more than two tables by using a series of joins within a SELECT statement, or you can join a table to it by using a self-join.

Introduction to Joins

You join tables to produce a single result set that incorporates rows and columns from two or more tables. When you join tables, Microsoft SQL Server version 7.0 uses the values in the join column(s) from one table to select rows from another table. Rows with matching join column values are combined into new rows, which make up the result set.

For example, in a properly normalized database, only a product identifier will be stored in an orders table. If you need to retrieve a list of orders that includes product descriptions, you will join the orders table and the products table on the product identifier. The product description and the other columns from the product table may then be used in the select list of the query. In this case, the join performs a lookup function to retrieve the product description for each order.

Partial Syntax

```
SELECT {column_name} [,…n]
FROM {table_or_view_name}
  [
  [ INNER | {{LEFT | RIGHT | FULL} [OUTER] }]
  JOIN
  table_or_view_name ON search_conditions]
  ] […n]
```

Selects Specific Columns from Multiple Tables

A join allows you to select columns from multiple tables by expanding on the FROM clause of the SELECT statement. Two additional keywords are included in the FROM clause—JOIN and ON:

- The JOIN keyword and its options specify which tables are to be joined and how to join them.
- The ON keyword specifies the columns that the tables have in common.

Queries Two or More Tables to Produce a Result Set

A join allows you to query two or more tables to produce a single result set. When you implement joins, consider the following facts and guidelines:

- Most join conditions are based on the primary key of one table and the foreign key of another table.
- If a table has a composite primary key, you must reference the entire key in the ON clause when you join tables.
- Use columns common to the specified tables to join the tables. These columns should have the same or similar data types, but do not have to have the same names.
- Qualify each column name using the *table_name.column_name* format if the column names are the same.
- Try to limit the number of tables in a join because queries that join a large number of tables are slower than queries that join fewer tables.

Using Inner Joins

Inner joins combine tables by comparing values in columns that are common to both tables. SQL Server returns only rows that match the join conditions.

When to Use Inner Joins

Use inner joins to obtain information from two separate tables and combine that information into one result set. When you use inner joins, consider the following facts and guidelines:

- Inner joins are the SQL Server default. You can abbreviate the INNER JOIN clause to JOIN.
- Specify the columns that you want to display in your result set by including the column names in the select list.
- Include a WHERE clause to restrict the rows that are returned in the result set.
- Do not use a null value as a join condition because null values do not evaluate equally with one another.
- SQL Server does not guarantee an order in the result set unless one is specified with an ORDER BY clause.
- You can use multiple JOIN clauses and join more than two tables in the same query.

Example 1

This example returns the buyer_name, buyer_id, and qty values for the buyers who purchased products. Buyers who did not purchase any products are not included in the result set. Buyers who bought more than one product are listed for each purchase. The buyer_id column from either table can be specified in the select list.

```
USE joindb
SELECT buyer_name, sales.buyer_id, qty
FROM buyers INNER JOIN sales
ON buyers.buyer_id = sales.buyer_id
```

Result

```
buyer_name                buyer_id    qty
------------------------- ----------- -----------
Adam Barr                 1           15
Adam Barr                 1           5
Erin O'Melia              4           37
Eva Corets                3           11
Erin O'Melia              4           1003

(5 row(s) affected)
```

Example 2

This example returns the names of products and the companies that supply the products. Products without listed suppliers, and suppliers without current products, are not included in the result set.

```
USE Northwind
SELECT ProductName, CompanyName
FROM Products INNER JOIN Suppliers
ON Products.SupplierID = Suppliers.SupplierID
```

Result

```
ProductName                       CompanyName
--------------------------------- ------------------------------
-----
Chai                              Exotic Liquids
Chang                             Exotic Liquids
Aniseed Syrup                     Exotic Liquids
Chef Anton's Cajun Seasoning      New Orleans Cajun Delights
…
Lakkalikööri                      Karkki Oy
Original Frankfurter grüne Soße   Plutzer Lebensmittelgroßmärkte
AG

(77 row(s) affected)
```

Example 3

This example returns the customer name and order date for each order placed after 1/1/95. The WHERE clause is used to restrict the rows that are returned in the result set. The DISTINCT option of the SELECT clause removes any rows from the result set that

are complete duplicates of another row. In this example, the DISTINCT option ensures that only one row is returned per customer, per day, even if the customer placed several orders on the same day.

```
USE Northwind
SELECT DISTINCT CompanyName, OrderDate
FROM Orders INNER JOIN Customers
ON Orders.CustomerID = Customers.CustomerID
WHERE OrderDate > '1/1/95'
```

Result

```
CompanyName                              OrderDate
---------------------------------------- --------------------------
Alfreds Futterkiste                      1997-08-25 00:00:00.000
Alfreds Futterkiste                      1997-10-03 00:00:00.000
Alfreds Futterkiste                      1997-10-13 00:00:00.000
Alfreds Futterkiste                      1998-01-15 00:00:00.000
…
Wolski  Zajazd                           1998-04-03 00:00:00.000
Wolski  Zajazd                           1998-04-23 00:00:00.000

(823 row(s) affected)
```

Example 4

This example returns the title number and the member number of the borrower of each book that is currently on loan in the library database. Both the copy and loan tables have a composite primary key consisting of the isbn and copy_no columns. When joining these tables, you must specify both columns in the join condition because together they uniquely identify a particular copy of a book. The AND operator is used to specify multiple columns in the join condition.

```
USE library
SELECT copy.title_no, loan.member_no
FROM copy INNER JOIN loan
ON copy.isbn = loan.isbn
AND copy.copy_no = loan.copy_no
WHERE copy.on_loan = 'Y'
```

Result

```
title_no    member_no
----------- ---------
1           338
1           364
1           3588
1           3614
```

```
…
50          9490
50          9529

(2000 row(s) affected)
```

It is possible to create joins without specifying the JOIN and ON clauses by listing the tables in a comma-separated list in the FROM clause and specifying the join condition in the WHERE clause. You should use the JOIN and ON clauses because they separate join conditions from other search conditions and make queries easier to read.

Using Outer Joins

Like inner joins, outer joins return combined rows that match the join condition. However, in addition to the rows that match the join condition, left and right outer joins return unmatched rows from one of the tables and full outer joins return unmatched rows from both of the tables. Columns from the source table of the unmatched rows contain data, but columns from the other table contain null values.

When to Use Left or Right Outer Joins

Use left or right outer joins when you require a complete list of data from one of the joined tables—for example, if you generate a query of product sales by joining a products table and a sales table. An inner join will only return rows for products that were sold. If you use a left outer join and specify the products table as the left table, rows will be returned for every product. Sales columns will contain data for products that were sold and null for products that were not sold.

When you use left or right outer joins, consider the following facts and guidelines:

- Use a left outer join to display all rows from the first-named table (the table on the left of the JOIN clause).
- Use a right outer join to display all rows from the second-named table (the table on the right of the JOIN clause).
- Do not use a null value as a join condition because null values do not evaluate equally with one another.
- You can abbreviate the LEFT OUTER JOIN or RIGHT OUTER JOIN clause as LEFT JOIN or RIGHT JOIN.

Example 1

This example returns the buyer_name, buyer_id, and qty values for all buyers and their purchases. Notice that the buyers who did not purchase any products are listed in the result set, but null values appear in the buyer_id and qty columns, which are columns from the righthand table (sales).

```
USE joindb
SELECT buyer_name, sales.buyer_id, qty
FROM buyers LEFT OUTER JOIN sales
ON buyers.buyer_id = sales.buyer_id
```

Result

```
buyer_name                buyer_id    qty
------------------------- ----------- -----------
Adam Barr                 1           15
Adam Barr                 1           5
Sean Chai                 NULL        NULL
Eva Corets                3           11
Erin O'Melia              4           37
Erin O'Melia              4           1003

(6 row(s) affected)
```

To change this example query to yield the same result with a RIGHT OUTER JOIN clause, reverse the order of the tables in the FROM clause and use the RIGHT OUTER JOIN clause as follows:

```
USE joindb
SELECT buyer_name, sales.buyer_id, qty
FROM sales RIGHT OUTER JOIN buyers
ON buyers.buyer_id = sales.buyer_id
```

Example 2

This example displays the customer name and order date for each order in the Northwind database. By using a left outer join, even customers that have not placed orders are returned in the result set. NULL is returned in the OrderDate column for customers who have not placed an order.

```
USE Northwind
SELECT CompanyName, OrderDate
FROM Customers LEFT OUTER JOIN Orders
ON Customers.CustomerID = Orders.CustomerID
```

Result

```
CompanyName                              OrderDate
---------------------------------------- --------------------------
Vins et alcools Chevalier                1996-07-04 00:00:00.000
Toms Spezialitäten                       1996-07-05 00:00:00.000
```

```
Hanari Carnes                              1996-07-08 00:00:00.000
…
FISSA Fabrica Inter. Salchichas S.A.       NULL
Paris spécialités                          NULL

(832 row(s) affected)
```

The *= and =* syntax used for outer joins in previous versions of SQL Server is supported in SQL Server 7.0 for backward compatibility, but you should use the LEFT and RIGHT OUTER JOIN syntax, which is the ANSI SQL-92 standard syntax.

- **To use an outer join to query member reservations**

In this exercise, you will write and execute a query to retrieve the full name and member_no from the member table, as well as the isbn and log_date values from the reservation table, for member numbers 250, 341, and 1675. Order the results by member_no. You should show information for these members, even if they have no books on reserve. Write the select list of the query.

1. Create the name column by concatenating the lastname, firstname, and middleinitial for each member.
2. Create the date column by converting the log_date to the char (8) data type.
3. Write a FROM clause that creates an outer join between the member and reservation tables on the member_no columns.
4. Compose a WHERE clause to retrieve member numbers 250, 341, and 1675 from the member table. Write the ORDER BY clause to sort the result by the member numbers.
5. Execute the script.

```
USE library
SELECT
  me.member_no,
  me.lastname + ', ' + me.firstname + ' ' + me.middleinitial AS
[name],
  re.isbn,
  CONVERT(char(8),re.log_date,1) AS [date]
FROM member me LEFT OUTER JOIN reservation re
ON me.member_no = re.member_no
WHERE me.member_no IN (250,341,1675)
ORDER BY me.member_no
```

Using Cross Joins

Cross joins display every combination of all rows in the joined tables. A common column is not required, and the ON clause is not used for cross joins.

When to Use Cross Joins

Cross joins are rarely used on a normalized database, but you can use them to generate test data for a database or lists of all possible combinations for checklists or business templates.

When you use cross joins, SQL Server produces a Cartesian product in which the number of rows in the result set is equal to the number of rows in the first table, multiplied by the number of rows in the second table. For example, if there are 8 rows in one table and 9 rows in the other table, SQL Server returns a total of 72 rows.

Example 1

This example lists all possible combinations of the values in the Buyers.buyer_name and Sales.qty columns.

The use of a cross join displays all possible row combinations between these two tables. The Buyers table has 4 rows and the Sales table has 5 rows; therefore, the result set contains 20 rows.

```
USE joindb
SELECT buyer_name, qty
FROM buyers CROSS JOIN sales
```

Result

```
buyer_name                qty
------------------------- -----------
Adam Barr                 15
Adam Barr                 5
Adam Barr                 37
…
Erin O'Melia              11
Erin O'Melia              1003

(20 row(s) affected)
```

Example 2

This example displays a cross join between the Shippers and Suppliers tables that is useful for listing all the possible ways that suppliers can ship their products.

The use of a cross join displays all possible row combinations between these two tables. The Shippers table has 3 rows and the Suppliers table has 29 rows; therefore, the result set contains 87 rows.

```
USE Northwind
SELECT Suppliers.CompanyName, Shippers.CompanyName
FROM Suppliers CROSS JOIN Shippers
```

Result

```
CompanyName                              CompanyName
---------------------------------------- --------------------------
---
Aux joyeux ecclésiastiques               Speedy Express
Bigfoot Breweries                        Speedy Express
Cooperativa de Quesos 'Las Cabras'       Speedy Express
…
Tokyo Traders                            Federal Shipping
Zaanse Snoepfabriek                      Federal Shipping

(87 row(s) affected)
```

Advanced Joins

It is possible to join up to 256 tables in a single query. A table can be joined to many other tables within the same query. The joins can use either the same common column or different common columns.

Joining More Than Two Tables

When you use multiple joins, it is easier to consider each join independently—for example, if tables A, B, and C are joined. The first join combines table A and B to produce a result set; this result set is combined with table C in the second join to produce the final result set. Columns from any of the tables can be specified in the other clauses of the SELECT statement. If two tables have a column with the same name, qualify each column name using the *table_name.column_name* format.

NOTE

The query optimizer may produce a plan that processes the tables in a different order, but the result set is the same.

Example 1

This example returns the buyer_name, prod_name, and qty columns from the buyers, sales, and produce tables. The buyer_id column is common to the buyers and sales tables and is used to join these two tables. The prod_id column is common to the sales and produce tables and is used to join the produce table to the result of the join between the buyers and sales tables.

```
USE joindb
SELECT buyer_name, prod_name, qty
FROM buyers JOIN sales
ON buyers.buyer_id = sales.buyer_id
JOIN produce
```

```
ON sales.prod_id = produce.prod_id
```

Result

```
buyer_name                prod_name                 qty
------------------------- ------------------------- -----------
Erin O'Melia              Apples                    37
Adam Barr                 Pears                     15
Erin O'Melia              Pears                     1003
Adam Barr                 Oranges                   5
Eva Corets                Peaches                   11

(5 row(s) affected)
```

Example 2

This example displays information from the Orders and Products tables using the Order Details table as a link. For example, if you want a list of products that are ordered each day, you would need information from the Orders and Products tables. An order can consist of many products, and a product can be ordered on many orders. The Products and Orders tables have a many-to-many relationship.

To retrieve information from both the Orders and Products tables, you need to use an intermediate table and two joins. Although you do not need to retrieve any columns from the Order Details table, you must include this table in the query to relate the Orders table to the Products table. In this example, the OrderID column is common to the Orders and Order Details tables, and the ProductID column is common to the Order Details and Products tables.

```
USE Northwind
SELECT OrderDate, ProductName
FROM Orders O INNER JOIN [Order Details] OD
ON O.OrderID = OD.OrderID
JOIN Products P
ON OD.ProductID = P.ProductID
WHERE OrderDate = '6/16/97'
```

Result

```
OrderDate                 ProductName
------------------------- ----------------------------------------
1997-06-16 00:00:00.000   Gorgonzola Telino
1997-06-16 00:00:00.000   Lakkalikööri

(2 row(s) affected)
```

- **To join several tables and order the results**

In this exercise, you will write and execute a query on the title, item and copytables that returns the isbn, copy_no, on_loan, title, translation, and cover values for rows in the copy table with an ISBN of 1 (one), 500 (five hundred), or 1000 (one thousand). Order the results by the isbn column.

Write the select list of the query. Qualify the name of each column with a table alias of at least two characters (for example, ti.title_no for title.title_no).

1. Write a FROM clause that creates a join between the title and copytables on the title_no columns. Set up the table aliases in the FROM clause that you used in the select list.
2. Add a second JOIN clause to create a join between the item and copy tables on the isbn columns.
3. Compose a WHERE clause to restrict the rows that are retrieved from the copy table to those with an ISBN of 1 (one), 500 (five hundred), or 1000 (one thousand).
4. Write the ORDER BY clause to sort the result by ISBN.
5. Execute the script.

```
USE library
SELECT
  co.isbn,
  co.copy_no,
  co.on_loan,
  ti.title,
  it.[translation],
  it.cover
FROM copy co JOIN title ti
ON co.title_no = ti.title_no
JOIN item it
ON co.isbn = it.isbn
WHERE (co.isbn IN (1, 500, 1000))
ORDER BY co.isbn
```

Performance

When joining more than two tables, especially if the tables are large, you should analyze the query using the SET SHOWPLAN_TEXT, SET SHOWPLAN_ALL, and SET STATISTICS IO options and the graphical execution plan tools to the query can have a significant effect on the amount of logical reads that are needed to process the query.

If you refer to columns that are common to two or more tables in the WHERE clause of these queries, the table prefix you choose will affect the performance of the query.

Example

In this example, a list of titles and the names of members who have borrowed each title is retrieved by joining the title, loanhist, and member tables in the library database. A

search condition on the member number is specified in the WHERE clause. The result set is the same whether the member_no column of the member table or the member_no column of the loanhist table is specified in the WHERE clause. However, if you turn on Show stats I/O and Show stats time in the Current Connection Options in SQL Server Query Analyzer, you will see that the query that specifies the member_no column of the member table executes about three times faster.

The following is a query specifying the member_no column of the member table in the WHERE clause:

```
USE library
SELECT DISTINCT title, firstname + ' ' + lastname
FROM title t
JOIN loanhist l ON t.title_no = l.title_no
JOIN member m ON m.member_no = l.member_no
WHERE m.member_no < 100
```

Executing this query will yield the following Show stats I/O and stats time results:

```
Table 'title'. Scan count 624, logical reads 1248...
Table 'loanhist'. Scan count 99, logical reads 2071...
Table 'member'. Scan count 1, logical reads 101...

SQL Server Execution Times:
   CPU time = 160 ms,  elapsed time = 157 ms.
```

The following is a query specifying the member_no column of the loanhist table in the WHERE clause:

```
USE library
SELECT DISTINCT title, firstname + ' ' + lastname
FROM title t
JOIN loanhist l ON t.title_no = l.title_no
JOIN member m ON m.member_no = l.member_no
WHERE l.member_no < 100
```

Executing this query will yield the following Show stats I/O and stats time results:

```
Table 'title'. Scan count 624, logical reads 1248...
Table 'member'. Scan count 624, logical reads 1872...
Table 'loanhist'. Scan count 1, logical reads 511...

SQL Server Execution Times:
   CPU time = 561 ms,  elapsed time = 565 ms.
```

Joining a Table to Itself

If you want to find rows that have values in common with other rows in the same table, you can join a table to another instance of itself; this is know as a *self-join*.

When to Use Self-Joins

Self-joins are used to represent hierarchies or tree structures. For example, a company employment structure is typically hierarchical. Each employee has a manager who is also an employee. In an employee table, the primary key is the employee ID and the manager ID column is th foreign key that relates the employee table to itself.

Self-joins are also useful for finding matching data in a table. For example, in the pubs database you can create a query on the authors table that lists for each author the other authors that live in the same city.

When you use self-joins, consider the following guidelines:

- Table aliases are required to reference two copies of the table. Remember that table aliases are different from column aliases. A table alias is specified in the FROM clause after the table name.
- It may be necessary to use conditions in the WHERE clause to filter out duplicate matching rows if the same column is used more than once in the select list.

The next three examples develop the same query to show how conditions in the WHERE clause are used to create a meaningful self-join query.

Example 1

This example displays a list of all buyers who purchased the same products. The first, third, fourth, fifth, and seventh rows of the result set are simply rows that match themselves. The sixth row mirrors the second row. These rows should be eliminated from the result set.

```
USE joindb
SELECT a.buyer_id AS buyer1, a.prod_id, b.buyer_id AS buyer2
FROM sales a JOIN sales b
ON a.prod_id = b.prod_id
```

Result

```
buyer1      prod_id     buyer2
----------- ----------- -----------
1           2           1
4           2           1
1           3           1
4           1           4
3           5           3
1           2           4
```

```
4           2           4

(7 row(s) affected)
```

Example 2

This example displays a list of buyers who all purchased the same products, and it eliminates some of the extra rows that occurred in Example 1. Compare the result sets of Examples 1 and 2. The WHERE clause with the not equal to (<>) operator eliminates rows in the result set that match themselves. However, duplicate rows that are mirror images of one another are still returned in the result set.

```
USE joindb
SELECT a.buyer_id AS buyer1, a.prod_id, b.buyer_id AS buyer2
FROM sales a JOIN sales b
ON a.prod_id = b.prod_id
WHERE a.buyer_id <> b. buyer_id
```

Result

```
buyer1      prod_id     buyer2
----------- ----------- -----------
4           2           1
1           2           4

(2 row(s) affected)
```

Example 3

This example displays a list of buyers who all purchased the same products. The WHERE clause with the greater than (>) operator eliminates the mirrored row and the self-matched rows that occurred in Examples 1 and 2.

```
USE joindb
SELECT a.buyer_id AS buyer1, a.prod_id, b.buyer_id AS buyer2
FROM sales a JOIN sales b
ON a.prod_id = b.prod_id
WHERE a.buyer_id > b. buyer_id
```

Result

```
buyer1      prod_id     buyer2
----------- ----------- -----------
4           2           1

(1 row(s) affected)
```

Example 4

This example lists the names of employees and their managers.

```
USE Northwind
SELECT E.FirstName, E.LastName, M.FirstName, M.LastName
FROM Employees E LEFT OUTER JOIN Employees M
ON M.EmployeeID = E.ReportsTo
```

Result

```
FirstName  LastName             FirstName  LastName
---------- -------------------- ---------- --------------------
Nancy      Davolio              Andrew     Fuller
Andrew     Fuller               NULL       NULL
Janet      Leverling            Andrew     Fuller
Margaret   Peacock              Andrew     Fuller
Steven     Buchanan             Andrew     Fuller
Michael    Suyama               Steven     Buchanan
Robert     King                 Steven     Buchanan
Laura      Callahan             Andrew     Fuller
Anne       Dodsworth            Steven     Buchanan

(9 row(s) affected)
```

Combining Multiple Result Sets

The UNION operator combines the result of two or more SELECT statements into a single result set. Unlike joins, which combine rows from the base tables into single rows, UNION appends rows from the result sets after each other.

Use the UNION operator when the data that you want to retrieve cannot be accessed with a single query. UNION is often used to recombine tables that have been partitioned. For example, old rows from an orders table can be moved to archive tables that each hold orders from a three-month period. To retrieve a single result set from more than one of these tables, create individual SELECT statements and combine them with the UNION operator.

When you use the UNION operator, consider the following facts and guidelines:

- The result sets must have matching columns. The columns do not have to have the same names, but there must be the same number of columns in each result set and the columns must have compatible data types and be in the same order.
- SQL Server removes duplicate rows from the result set. However, if you use the ALL option, all rows (including duplicates) are included in the result set.

- The column names in the result set are taken from the first SELECT statement. Therefore, if you want to define new column headings for the result set, you must create the column aliases in the first SELECT statement.
- If you want the entire result set to be returned in a specific order, you must specify a sort order by including an ORDER BY clause after the last SELECT statement. Otherwise, the result set may not be returned in the order that you want. SQL Server generates an error if you specify an ORDER BY clause for any of the other SELECT statements in the query.
- An ORDER BY clause used with the UNION operator can only use items that appear in the select list of the first SELECT statement.
- You may experience better performance if you break a complex query into multiple SELECT statements and then use the UNION operator to combine them.

Syntax

```
select_statement
UNION [ALL]
select_statement
[,...n]
```

Example 1

This example combines two result sets. The first result set returns the name, city, and postal code of each employee from the Employees table. The second result set returns the name, city, and postal code of each customer from the Customers table. Notice that the customers are listed before the employees, although the Employees table is referenced first in the statement. This is because ordering is not guaranteed unless you specify an ORDER BY clause.

```
USE Northwind
SELECT FirstName + ' ' + LastName AS [Name], City, PostalCode
FROM Employees
UNION
SELECT CompanyName, City, PostalCode
FROM Customers
```

Result

```
Name                                     City            PostalCode
---------------------------------------- --------------- ----------
Alfreds Futterkiste                      Berlin          12209
Ana Trujillo Emparedados y helados       México D.F.     05021
Antonio Moreno Taquería                  México D.F.     05023
…
Michael Suyama                           London          EC2 7JR
Anne Dodsworth                           London          WG2 7LT

(100 row(s) affected)
```

Example 2

This example adds an ORDER BY clause to the statement from Example 1.

```
USE Northwind
SELECT FirstName + ' ' + LastName AS [Name], City, PostalCode
FROM Employees
UNION
SELECT CompanyName, City, PostalCode
FROM Customers
ORDER BY [Name]
```

Result

```
Name                                     City            PostalCode
---------------------------------------- --------------- ----------
Alfreds Futterkiste                      Berlin          12209
Ana Trujillo Emparedados y helados       México D.F.     05021
Andrew Fuller                            Tacoma          98401
…
Wilman Kala                              Helsinki        21240
Wolski  Zajazd                           Warszawa        01-012

(100 row(s) affected)
```

- **To produce a single result set from two SELECT statements**

In this exercise, you will write a query to retrieve a single list of members, both adult and juvenile, who have reserved books with ISBN 'numbers 1, 43, or 288. The list must include the isbn, title, member_no, and name (lastname and firstname) of each member who has the reservation. Additionally, the list should indicate whether the member is an adult or a juvenile.

Write a SELECT statement that returns information about juvenile members from the reservation, item, title, member, and juvenile tables.

1. Write a SELECT statement that returns information about adult members from the reservation, item, title, member, and adult tables.
2. Add WHERE clauses to both statements to restrict the rows that are returned to those with ISBN numbers of 1, 43, and 288.
3. Combine these statements with the UNION operator, add an ORDER BY clause to sort the results by ISBN, and then execute the query.

```
USE library
SELECT
  re.isbn,
  ti.title,
  me.member_no,
```

```
  me.lastname + ', ' + substring(firstname,1,1) AS [name],
  'juvenile' AS age
FROM reservation re JOIN item it ON re.isbn = it.isbn
JOIN title ti ON it.title_no = ti.title_no
JOIN member me ON re.member_no = me.member_no
JOIN juvenile ju ON re.member_no = ju.member_no
WHERE re.isbn in (1,43, 288)
UNION
SELECT
  re.isbn,
  ti.title,
  me.member_no,
  me.lastname + ', ' + substring(firstname,1,1) AS [name],
  'adult' AS age
FROM reservation re JOIN item it ON re.isbn = it.isbn
JOIN title ti ON it.title_no = ti.title_no
JOIN member me ON re.member_no = me.member_no
JOIN adult ad ON re.member_no = ad.member_no
WHERE re.isbn in (1,43,288)
ORDER BY re.isbn
```

Creating a Table from a Result Set

You can place the result set of any query into a new table with the SELECT INTO statement. You may also use the SELECT INTO statement to create and populate new tables and to create temporary tables and break down complex problems that require a data set from various sources. If you first create a temporary table, the queries that you execute on it may be simpler than those you would execute on multiple tables or databases.

When you use the SELECT INTO statement, consider the following facts and guidelines:

- SQL Server creates a table and inserts a result set into the table.

  Ensure that the table name that is specified in the SELECT INTO statement is unique. If a table exists with the same name, the SELECT INTO statement fails.

- You can create a local or global temporary table.

  Create a local temporary table by preceding the table name with a number sign (#) or create a global temporary table by preceding the table name with a double number sign (##).

  A local temporary table is available only to the connection that created it, while a global temporary table is available to all connections:

  - A local temporary table is deleted when the user closes the connection.
  - A global temporary table is deleted when the table is no longer used by any connections.

- Set the select into/bulkcopy database option on in order to create a permanent table.
- You must create column aliases for calculated columns in the select list. You may use aliases to rename other columns; otherwise, the column name will be used.

Syntax

```
SELECT select_list
INTO new_table_name
FROM table_source
[WHERE search_condition]
```

Example

This example creates a local temporary table based on a query made on the Products table.

```
USE Northwind
SELECT ProductName, UnitPrice AS Price, (UnitPrice * 0.1) AS Tax
INTO #pricetable
FROM Products
```

No results are returned by the SELECT INTO statement. Query the new table to see its data. For example:

```
SELECT * FROM #pricetable
```

Result

```
ProductName                              Price            Tax
---------------------------------------- ---------------- -------------
-----
Chai                                     18.0000          1.80000
Chang                                    19.0000          1.90000
Aniseed Syrup                            10.0000          1.00000
…
Lakkalikööri                             18.0000          1.80000
Original Frankfurter grüne Soße          13.0000          1.30000

(77 row(s) affected)
```

- **To create and populate a temporary table by using the SELECT INTO statement**

In this exercise, you will create and populate a temporary table named *#overdue* by using the INTO clause of the SELECT statement.

1. Write and execute a query that returns the member_no, lastname, out_date, due_date, and title columns from the loan, member, and title tables. Convert the datetime values from the out_date and due_date columns to char (12), format 101. Do not add the INTO clause.
2. Write a WHERE clause that restricts the results to past due loans. Use the GETDATE () function (which returns the current date and time) in the search argument to check for past due loans. Do not add the INTO clause. Execute the query and confirm that it returns approximately 1120 overdue loans.

```
USE library
SELECT
  lo.member_no,
  me.lastname,
  CONVERT(char(12),lo.out_date,101) AS out_date,
  CONVERT(char(12),lo.due_date,101) AS date_due,
  ti.title
FROM loan lo
JOIN member me ON lo.member_no = me.member_no
JOIN title ti ON lo.title_no = ti.title_no
WHERE (lo.due_date < GETDATE())
```

3. Now that you have tested your query, add an INTO clause that creates a temprary table called *#overdue*. Leave the query window open after you create the temporary table.
4. Write and execute a query that returns the member number and a calculated column called *pastdue* that lists the number of past due loans of each member from the # overdue table. Use the COUNT(*) aggregate function and group the results by the member_no column.

```
USE library
SELECT member_no, COUNT(*) AS pastdue
FROM #overdue
GROUP BY member_no
```

Using Aggregate Functions

Functions that calculate summary values such as an average or a sum are called *aggregate functions*. SQL Server provides a rich set of aggregate functions for quickly calculating summary values on columns of a table using a single statement.

When an aggregate function is executed, Microsoft SQL Server version 7.0 summarizes the values in a particular column for an entire table or for groups of rows within the table. A single aggregate value is produced for the entire table or for each group of rows. Use aggregate functions with the SELECT statement to summarize the values for an entire table. Add the GROUP BY clause to the SELECT statement to summarize values for groups of rows.

Introduction to the Aggregate Functions

The following table briefly describes each of the Transact-SQL aggregate functions:

Aggregate function	Description
AVG	The average of all the values
COUNT	The number of non-null values
COUNT(*)	The number of rows in the table or group including null values and duplicates
MAX	The maximum value from all the values
MIN	The minimum value from all the values
SUM	The sum of all the values
STDEV	The statistical deviation of all values
STDEVP	The statistical deviation for the population of all values
VAR	The statistical variance of all values
VARP	The statistical variance for the population of all values

All aggregate functions, except for the COUNT (*) function, return NULL if no rows satisfy the WHERE clause of the SELECT statement. The COUNT (*) function returns a value of zero if no rows satisfy the WHERE clause.

The data type of a column determines the functions that you can use with it. The following table describes the relationships between functions and data types.

Function	Supported data types
COUNT	Any type except uniqueidentifier, text, image, or ntext
MIN and MAX	Char, varchar, datetime, and all numeric data types except the bit data type
SUM, AVG, STDEV, STDEVP, VAR and VARP	All numeric types except the bit data type

NOTE

Index frequently aggregated columns to improve query performance. For example, if you frequently use aggregate functions on the quantity column when querying a sales table, indexing the quantity column will improve the performance of these queries.

When you use aggregate functions in the select list of a SELECT statement, you cannot use column names in the same statement because an aggregate function returns a single value and a column reference returns a value for each row. In the next lesson, you will learn how to group rows on columns that have common values in many rows.

Partial Syntax

```
SELECT {{AVG | COUNT | MAX | MIN | SUM
     | STDEV | STDEVP | VAR | VARP}(expression | *) } [,...n]
FROM table_list
[ WHERE search_conditions ]
```

Example 1

This example calculates the average unit price of all products in the Products table.

```
USE Northwind
SELECT AVG(UnitPrice)
FROM Products
```

Result

```
----------------------
28.8663
(1 row(s) affected)
```

Example 2

This example sums all rows in the Quantity column in the [Order Details] table.

```
USE Northwind
SELECT SUM(Quantity)
FROM [Order Details]
```

Result

```
-----------
51317

(1 row(s) affected)
```

Using Aggregate Functions with Null Values

Null values can cause aggregate functions to produce unexpected results. For example, if you execute a SELECT statement that includes a COUNT function on a column that contains 18 rows, two of which contain null values, the COUNT function returns a result of 16. SQL Server ignores the two rows that contain null values.

Therefore, use caution when using aggregate functions on columns that contain null values, because the result set may not be representative of your data. However, if you decide to use aggregate functions with null values, consider the following facts:

- All SQL Server aggregate functions, with the exception of the COUNT(*) function, ignore null values in columns.
- The COUNT (*) function counts all rows, even if every column contains a null value. For example, if you execute a SELECT statement that includes the COUNT (*) function on a column that contains 18 rows, two of which contain null values, the COUNT (*) function returns a result of 18.

Example 1

This example lists the number of employees in the Employees table.

```
USE Northwind
SELECT COUNT(*)
FROM Employees
```

Result

```
-----------
9
(1 row(s) affected)
```

Example 2

This example lists the number of employees who have a null value in the ReportsTo column in the Employees table, indicating that no reporting manager is defined for that employee.

NOTE

You should always use IS NULL or IS NOT NULL when performing comparisons with null values. SET ANSI_NULLS controls SQL Server's behavior when performing comparisons to null values. If you use = to perform a comparison to a null value, the result will be null rather than true or false if SET ANSI_NULLS is ON, which is the default setting.

```
USE Northwind
SELECT COUNT(*)
FROM Employees
WHERE ReportsTo IS NULL
```

Result

```
-----------
1
(1 row(s) affected)
```

GROUP BY Fundamentals

An aggregate function used in a simple SELECT statement that has only a FROM and a WHERE clause produces a single summary value for all rows in a column.

If you want to generate summary values for a column based on groups of rows, use aggregate functions in SELECT statements with the GROUP BY clause. Use the HAVING clause with the GROUP BY clause to restrict which groups of rows that are returned in the result set.

Using the GROUP BY Clause

Use the GROUP BY clause on columns or expressions to organize rows into groups and to summarize those groups.

Partial Syntax

```
SELECT select_list
FROM table_source
[ WHERE search_condition ]
[ GROUP BY [ALL] group_by_expression [,...n]]
[HAVING search_condition ]
```

When you use the GROUP BY clause, consider the following facts and guidelines:

- SQL Server produces a row of values for each defined group.
- SQL Server returns only single rows for each group that you specify; it does not return detail information.
- All columns that are specified in the select list must be included in the GROUP BY clause. Columns included in the GROUP BY clause do not need to be specified in the select list.
- If you include a WHERE clause, SQL Server groups only the rows that satisfy the WHERE clause conditions.

- The total size of the columns listed in the GROUP BY clause can be up to 8,060 bytes.
- If you use the GROUP BY clause on columns that contain null values, the null values are processed as a group. To exclude the rows with null values, use the IS [NOT] NULL comparison operator in the WHERE clause of the SELECT statement.
- Use the ALL keyword with the GROUP BY clause to display summary rows for every group, regardless of whether the rows satisfy the WHERE clause. For groups for which no rows satisfy the WHERE clause, NULL is returned in the summary columns.

Example 1

This example returns information about orders from the OrderHist table. The query groups the orders for each product ID and calculates the total quantity ordered for that product. The total quantity is calculated with the SUM aggregate function, which displays one value for each product in the result set.

```
USE Northwind
SELECT ProductID, SUM(Quantity) AS Total_Quantity
FROM OrderHist
GROUP BY ProductID
```

Result

```
ProductID   Total_Quantity
----------- --------------
1           15
2           35
3           45

(3 row(s) affected)
```

Example 2

This example adds a WHERE clause to the query in Example 1. The query restricts the rows to product ID 2 and then groups these rows and calculates the total quantity ordered. Compare this result set to that in Example 1.

```
USE Northwind
SELECT ProductID, SUM(Quantity) AS Total_Quantity
FROM OrderHist
WHERE ProductID = 2
GROUP BY ProductID
```

Result

```
ProductID   Total_Quantity
----------- --------------
2           35

(1 row(s) affected)
```

Example 3

This example summarizes information from the [Order Details] table. The query groups the orders by product ID and calculates the total quantity ordered for each product. This example does not include a WHERE clause and, therefore, returns a total for each product ID.

```
USE Northwind
SELECT ProductID, SUM(Quantity) AS Total_Quantity
FROM [Order Details]
GROUP BY ProductID
```

Result

```
ProductID   Total_Quantity
----------- --------------
61          603
3           328
32          297
…
47          485
38          623

(77 row(s) affected)
```

Example 4

This example adds a WHERE clause to the query in Example 3. The query returns only rows that have a product ID less than 5. Compare this result set with the result set in Example 3.

CAUTION

The WHERE clause must be listed before the GROUP BY clause in the SELECT statement or SQL Server will return an error.

```
USE Northwind
SELECT ProductID, SUM(Quantity) AS Total_Quantity
FROM [Order Details]
WHERE ProductID < 5
GROUP BY ProductID
```

Result

```
ProductID   Total_Quantity
----------- --------------
1           828
2           1057
3           328
4           453

(4 row(s) affected)
```

Using the GROUP BY Clause with the HAVING Clauses

Use the HAVING clause on columns or expressions to set conditions on the groups included in a result set. The HAVING clause sets conditions on the GROUP BY clause in much the same way that the WHERE clause sets conditions for the rows returned by the SELECT statement.

The WHERE clause determines which rows are grouped. The HAVING clause determines which groups are returned.

When you use the HAVING clause, consider the following facts and guidelines:

- Use the HAVING clause only with the GROUP BY clause to restrict the grouping. Using the HAVING clause without the GROUP BY clause is not meaningful.
- You can have up to 128 conditions in a HAVING clause. When you have multiple conditions, combine them with logical operators (AND, OR, or NOT).
- In the HAVING clause, you can reference any of the columns that appear in the select list, including the aggregate functions. You cannot reference aggregate functions in the WHERE clause.
- The HAVING clause is applied after the ALL keyword so the HAVING clause overrides the ALL keyword and only returns groups that satisfy the HAVING clause.

Example 1

This example lists each product from the OrderHist table that has orders of 30 or more units.

```
USE Northwind
SELECT ProductID, SUM(Quantity) AS Total_Quantity
FROM OrderHist
GROUP BY ProductID
HAVING SUM(Quantity) >= 30
```

Result

```
ProductID   Total_Quantity
----------- --------------
2           35
3           45

(2 row(s) affected)
```

Example 2

This example lists the product ID and quantity for products that have orders for more than 1,200 units in the [Order Details] table.

```
USE Northwind
SELECT ProductID, SUM(Quantity) AS Total_Quantity
FROM [Order Details]
GROUP BY ProductID
HAVING SUM(Quantity) > 1200
```

Result

```
 ProductID   Total_Quantity
----------- --------------
59          1496
56          1263
60          1577
31          1397

(4 row(s) affected)
```

- **To use the GROUP BY and HAVING clauses to summarize data**

In this exercise, you will write queries that include the GROUP BY and the HAVING clauses.

Open SQL Server Query Analyzer.

1. Write a query that summarizes the quantity sold by category for all products, regardless of category in the [Order Details] table in the Northwind database. Include CategoryID and Total_Quantity in the select list of the query. Execute the query.

   ```
   USE Northwind
   SELECT CategoryID, SUM(Quantity) AS Total_Quantity
   FROM [Order Details]
   JOIN Products ON [Order Details].ProductID = Products.ProductID
   GROUP BY CategoryID
   ORDER BY CategoryID
   ```

Result

Your result will look similar to the following result set:

```
CategoryID  Total_Quantity
----------- --------------
1           9532
2           5298
3           7906
4           9149
5           4562
6           4199
7           2990
8           7681

(8 row(s) affected)
```

2. Modify the query to summarize the quantity by the OrderID column rather than the CategoryID column, and then execute the query.

```
USE Northwind
SELECT OrderID, SUM(Quantity) AS Total_Quantity
FROM [Order Details]
GROUP BY OrderID
```

Result

Your result will look similar to the following partial result set:

```
OrderID     Total_Quantity
----------- --------------
10248       27
10249       49
10250       60
…
11076       50
11077       72

(830 row(s) affected)
```

3. Modify the query to limit the group summaries to orders with a total quantity of 250 and higher, and then execute the query.

```
USE Northwind
SELECT OrderID, SUM(Quantity) AS Total_Quantity
FROM [Order Details]
GROUP BY OrderID
HAVING SUM(Quantity) >= 250
```

Result

Your result will look similar to the following result set:

```
 OrderID     Total_Quantity
----------- --------------
10515       286
10612       263
10658       255
10678       280
10847       288
10895       346
10990       256
11030       330

(8 row(s) affected)
```

Managing Transactions

This lesson describes how to define transactions, what to consider when you use them, how to set an implicit transaction option, and the restrictions on using transactions. Transaction processing and recovery is also discussed.

Transactions ensure that multiple data modifications are processed as a unit; this is known as *atomicity*. For example, a banking transaction might credit one account and debit another. Both steps must be completed together. SQL Server supports transaction processing to manage multiple transactions.

Locks prevent update conflicts. Users cannot read or modify data that other users are in the process of changing. For example, if you want to compute an aggregate and ensure that another transaction does not modify the set of data that is used to compute the aggregate, you can request that the system hold locks on the data.

For a video demonstration that covers SQL Server transactions, run the trans.htm file from the \Media folder on the Supplemental Course Materials CD-ROM.

SQL Server transactions employ the following syntax. These statements are used to specify explicit transactions, which were known as user-defined transactions in earlier versions of SQL Server. If explicit transactions are not specified, every data modification statement automatically uses its own transaction.

Syntax

```
BEGIN TRAN[SACTION] [transaction_name]
```

The *transaction_name* option specifies a user-defined transaction name. Transaction names are not required but make the code easier to read.

Syntax

```
BEGIN DISTRIBUTED TRAN[SACTION]
 [transaction_name | @tran_name_variable]
```

The Microsoft Distributed Transaction Coordinator (MS DTC) manages all distributed transactions

Syntax

```
COMMIT TRAN[SACTION] [transaction_name]
```

Example

This example defines a transaction to transfer funds between the checking and savings accounts of a customer.

```
BEGIN TRAN fund_transfer
EXEC debit_checking 100, 'account1'
EXEC credit_savings 100, 'account1'
COMMIT TRAN fund_transfer
```

Syntax

```
ROLLBACK TRAN[SACTION] [transaction_name]
```

Example

This example defines a transaction to transfer funds between the checking and savings accounts of a customer.

```
BEGIN TRAN fund_transfer
EXEC debit_checking 100, 'account1'
IF (some test of the checking balance)
BEGIN
EXEC credit_savings 100, 'account1'
COMMIT TRAN fund_transfer
END
ELSE
     ROLLBACK TRAN fund_transfer
```

Transaction Recovery and Checkpoints

Because the transaction log records all transactions, SQL Server can recover data automatically in case of a power loss, system software failure, client problem, or a

transaction cancellation request, SQL Server automatically guarantees that all committed transactions are reflected in the database in case of a failure. It uses the transaction log to roll forward all committed transactions and to roll back any uncommitted transactions.

How Data is modified by a Transaction

Before a modification takes place, pages are the same in the data cache and on the disk. The following process then occurs:

1. Changes appear in the data cache as transactions are committed. These changes are recorded in the transaction log.
2. When a checkpoint occurs, the cache is written to disk. The disk is once again the same as the cache, and the transaction log entries are marked (check-pointed) as having already been written to disk.

CAUTION

Do not use a write-caching disk controller with SQL Server; it compromises the ability of SQL Server to manage transactions because it can appear that write-ahead logging has completed, even when it has not. The only exception is if the write-caching mechanism on the server was designed for a database server.

Considerations for Using Transactions

It is usually a good idea to keep transactions short and to avoid nesting transactions.

Transaction Guidelines

Transactions should be as short as possible. Longer transactions increase the likelihood that users will not be able to access locked data. Some methods to keep transactions short include the following:

- To minimize transaction time, use caution when you use certain Transact-SQL statements, such as a WHILE statement.
- Address issues that require user interaction before you start the transaction. For example, if you are updating a customer record, it is most efficient to obtain the necessary information from the user before you begin the transaction.
- INSERT, UPDATE, and DELETE should be the primary statements in a transaction, and they should be written to affect the fewest possible number of rows.

Issues in Nesting Transactions

Consider the following issues regarding nesting transactions:

- It is possible to nest transactions, but nesting does not affect how SQL Server processes the transaction. Nesting is not recommended.

  Only the outermost BEGIN…COMMIT statement pair is actually used by SQL Server. SQL Server takes no action when inner nested transactions are committed. Usually, transaction nesting occurs when stored procedures or triggers, that have BEGIN...COMMIT statement pairs, invoke each other.

- You can use the @@TRANCOUNT function to determine whether any open transactions exist and how deeply they are nested:
  - @@TRANCOUNT equals zero when no open transactions exist.
  - A BEGIN TRAN statement increments @@TRANCOUNT by one, and a ROLLBACK TRAN statement sets @@TRANCOUNT to zero.
- You also can use the DBCC OPENTRAN statement to retrieve information on active transactions.

Setting the Implicit Transactions Option

In most cases, it is best to define transactions explicitly with the BEGIN TRANSACTION statement. However, for applications that were originally developed on systems other than SQL Server, the SET IMPLICIT_TRANSACTIONS option can be useful. It sets the implicit transaction mode for a connection.

Syntax

```
SET IMPLICIT_TRANSACTIONS {ON ý OFF}
```

Consider the following facts when you set implicit transactions:

- When the implicit transaction mode for a connection is on, executing any of the following statements triggers the start of a transaction: ALTER TABLE, INSERT, CREATE, OPEN, DELETE, REVOKE, DROP, SELECT, FETCH, TRUNCATE TABLE, GRANT, and UPDATE.
- Nested transactions are not allowed. If the connection is already in an open transaction, the statements do not start a new transaction.
- When the setting is on, the user must commit or roll back the transaction explicitly at the end of the transaction. Otherwise, the transaction and all data changes that it contains are rolled back when the user disconnects.
- The setting is off by default. When the setting is off, autocommit mode is used. In autocommit mode, each individual statement is committed if it completes without an error.

Restrictions on Explicit Transactions

A couple of restrictions exist on explicit transactions:

- Certain system stored procedures may not be included inside explicit transactions because they create temporary tables. For example, you cannot use the sp_dboption system stored procedure to set database options inside a transaction.
- Certain statements may not be included inside an explicit transaction. For example, some are long-running operations that you are not likely to use within the context of a transaction. Restricted statements include the following: ALTER DATABASE, BACKUP LOG, CREATE DATABASE, DROP DATABASE, RECONFIGURE, RESTORE DATABASE, RESTORE LOG, and UPDATE STATISTICS.

SQL Server Locking

This lesson describes concurrency issues, the resource items that can be locked, and the types of locks that can be placed on those resources, and how locks can be combined.

Consider the following facts about locks:

- Locks make possible the serialization of transactions so that only one person at a time can change a data element. For example, locks in an airline reservation system ensure that only one person is assigned a particular seat.
- SQL Server automatically sets the appropriate level of locking for each transaction. You can also control how some of the locks are used.
- Locks are necessary for concurrent transactions to allow users to access and update data at the same time. High concurrency means that there are many users who are experiencing good response time with little conflict. From the system administrator's perspective, the primary concerns are the number of users, the number of transactions, and the throughput. From the user's perspective, the overriding concern is response time.

Concurrency Problems Prevented by Locks

When you think about locking, you tend to focus on what happens when the data is written and how locks must ensure that the data is written cleanly. It is interesting to note that one of the main problems that locks have to deal with is ensuring the reliability of data that is read, because in most cases data that is written is dependent on data reads. For example, if an order is being created for a customer, the customer record must be read before the order is created to ensure that the customer has not been placed on hold. It is very important that the customer data not be changed while the order is being placed. Locks can prevent the following situations that compromise transaction integrity:

Lost Update

An update can get lost when a transaction overwrites the changes from another transaction. For example, two users can update the same information, but only the last change saved is reflected in the database.

Uncommitted Dependency (Dirty Read)

An uncommitted dependency occurs when a transaction reads uncommitted data from another transaction. The transaction can potentially make changes to data that is either inaccurate or nonexistent.

Inconsistent Analysis (Nonrepeatable Read)

An inconsistent analysis occurs when a transaction reads the same row more than one time and when, between the two (or more) readings, another transaction modifies that row. Because the row was modified between readings within the same transaction, each reading produces different values, which introduces inconsistency.

For example, an editor reads the same document twice, but between each reading, the writer rewrites the document. When the editor reads the document for the second time, it has completely changed. The original reading is not repeatable, leading to confusion. It would be better if the editor only start reading the document after the writer has completely finished writing it.

Phantoms

Phantoms can occur when transactions are not isolated from one another. For example, you could perform an update on all records in a region at the same time that another transaction inserts a new record for the region. The next time that the transaction reads the data, an additional record is present.

Lockable Resources

For optimal performance, the number of locks that SQL Server maintains must be balanced with the amount of data that each lock holds. To minimize the cost of locking, SQL Server automatically locks resources at a level that is appropriate to the task. SQL Server can lock the following types of items. The items are listed in increasing locking levels.

Item	Description
RID	A row identifier—used to lock a single row within a table
Key	A row lock within an index that protects key ranges in serializable transactions
Page	An 8-KB data page or index page
Extent	A contiguous group of eight data pages or index pages—used during space allocation

Table	An entire table, including all data and indexes
Database	An entire database—used during the restoration of a database

Types of Locks

SQL Server has two main types of locks: basic locks and locks for special situations.

Basic Locks

In general, read operations acquire shared locks, and write operations acquire exclusive locks.

Shared Locks

SQL Server typically uses shared (read) locks for operations that do not update data. If SQL Server has applied a shared lock to a resource, a second transaction also can acquire a shared lock, although the first transaction has not completed.

Consider the following facts about shared locks:

- They are used for read-only operations; data cannot be modified.
- SQL Server releases shared locks on a record as soon as the next record is read.
- A shared lock will exist until all rows that satisfy the query have been returned to the client.

Exclusive Locks

SQL Server uses exclusive (write) locks for the INSERT, UPDATE, and DELETE data modification statements.

Consider the following facts about exclusive locks:

- Only one transaction can acquire an exclusive lock on a resource.
- A transaction cannot acquire a shared lock on a resource that has an exclusive lock.
- A transaction cannot acquire an exclusive lock on a resource until all shared locks are released.

Special Situation Locks

Depending on the situation, SQL Server may use other types of locks such as intent, update, and schema locks.

Intent Locks

SQL Server uses intent locks internally to minimize locking conflicts. Intent locks establish a locking hierarchy. For example, if a transaction acquires exclusive row-level or page-level locks in a table, it will take an intent lock at the table-level. This is not because it intends to lock the table; rather, it intends to take lower-level locks in the table. SQL Server can now tell just by looking at the table that another transaction cannot take a table-level lock. Without intent locks, SQL Server would have to scan the whole table looking for row-level and page-level locks when another transaction wants to take a table-level lock.

Intent locks include intent share (IS), intent exclusive (IX), and shared with intent exclusive (SIX).

Update Locks

SQL Server uses update locks when it will modify a page at a later point. Before it modifies the page, SQL Server promotes the update page lock to an exclusive page lock to prevent locking conflicts. Without update locks, shared locks being acquired by a large number of users reading data could prevent a transaction from ever acquiring an exclusive lock.

Consider the following facts about update locks. Update locks are:

- Acquired during the initial portion of an update operation when the pages are first being read.
- Compatible with shared locks.
- Promoted to exclusive locks if the pages are modified.

Schema Locks

Schema locks ensure that a table or index is not dropped, or its schema modified, when it is referenced by another session.

SQL Server provides two types of schema locks:

- Schema stability (Sch-S), which ensures that a resource is not dropped.
- Schema modification (Sch-M), which ensures that other sessions do not reference a resource that is under modification.

Lock Compatibility

Locks may or may not be compatible with other locks. Locks are compatible if they can be held simultaneously on the same resource. Locks have a compatibility matrix that shows which locks are compatible with other locks that are obtained on the same resource. The locks in the following table are listed in order from the least restrictive (shared) to the most restrictive (exclusive).

Existing granted lock

Requested lock	IS	S	U	IX	SIX	X
Intent shared (IS)	Yes	Yes	Yes	Yes	Yes	No
Shared (S)	Yes	Yes	Yes	No	No	No
Update (U)	Yes	Yes	No	No	No	No
Intent exclusive (IX)	Yes	No	No	Yes	No	No
Shared with intent exclusive (SIX)	Yes	No	No	No	No	No
Exclusive (X)	No	No	No	No	No	No

An IX lock is compatible with other IX locks because IX means the intention to update only some of the rows, rather than all of them.

In addition, compatibility for schema locks is as follows:

- The schema modification lock (Sch-M) is incompatible with all locks.
- The schema stability lock (Sch-S) is compatible with all locks except the schema modification lock (Sch-M).

Optimistic and Pessimistic Concurrency

SQL Server locking ensures that updates do not interfere with each other but when you develop multi-user applications, you can use SQL Server locking in different ways depending on the nature of the application. Two types of concur-rency control are used when developing multi-user applications: pessimistic and optimistic.

Pessimistic Concurrency Control

Pessimistic concurrency control assumes that users will try to modify the same data concurrently. When a user reads data, the data is locked. The lock is held until the data is updated or the user moves on to other data. This lock guarantees that other users cannot modify the data while the first user is working with the data. When using pessimistic concurrency, locks may be held for a long time, increasing the likelihood of locking conflicts.

Optimistic Concurrency Control

Optimistic concurrency control assumes that users will not try to modify the same data concurrently. When a user reads data, the data is not locked. If the user needs to modify the data, the data is checked to see that it has not been changed since it was read. If it has not changed, a lock is taken, the data is modified, and the lock is released. If the data has

changed, it is read again to see the latest changes, and the user must make their changes again. When using optimistic concurrency, locks are held for a very short time but users may have to make changes again when they modify the same data as another user.

Implementing Concurrency Control

Whether you implement pessimistic or optimistic concurrency control is dependent on the nature of each client application. In a single application, it is likely that you would use both mechanisms. SQL Server cursors use pessimistic concurrency by default, but you can specify options so that optimistic concurrency is used. The OLE DB, ODBC, ADO, and RDO database interfaces all allow you to specify which type of concurrency will be used.

Managing Locks

This lesson describes locking options that you can specify at the session and table levels. It also describes how SQL Server handles deadlocks and how you can view information about locks.

Session-Level Locking Options

SQL Server allows you to control locking options at the session level by setting the transaction isolation level.

Transaction Isolation Level

An isolation level protects a specified transaction from other transactions. The transaction isolation level allows you to set the isolation level for all transactions during a session. When you set the isolation level, you specify the default locking behavior for all statements in your session.

You can override a session-level isolation level in individual statements by using locking hints. You can use the DBCC USEROPTIONS statement to determine the current session settings, including the transaction isolation level.

Syntax

```
SET TRANSACTION ISOLATION LEVEL {READ COMMITTED |
READ UNCOMMITTED | REPEATABLE READ | SERIALIZABLE}
```

The following table describes the locking isolation level options:

Option	Description

READ COMMITTED	Directs SQL Server to use shared locks while reading. At this level, you cannot experience "dirty reads". This option is the default.
READ UNCOMMITTED	Directs SQL Server not to issue shared locks and does not honor exclusive locks. You can experience "dirty reads."
REPEATABLE READ	Indicates that "dirty reads" and nonrepeatable reads cannot occur. Shared locks are taken when the data is read and held until the end of the transaction. Other users can insert new rows, causing phantoms.
SERIALIZABLE	Prevents other users from updating or inserting new rows that match the criteria in the WHERE clause of the transaction. Phantoms cannot occur.

Example

The following example sets the isolation level for the current session to READ UNCOMMITTED and then checks DBCC USEROPTIONS to verify that SQL Server has made the change.

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
DBCC USEROPTIONS
```

Result

```
Set Option                  Value
--------------------------- ----------------------------------------
-----
textsize                    64512
language                    us_english
dateformat                  mdy
datefirst                   7
ansi_null_dflt_on           SET
ansi_warnings               SET
ansi_padding                SET
ansi_nulls                  SET
concat_null_yields_null     SET
isolation level             read uncommitted

(10 row(s) affected)
```

DBCC always prints the following message when it is executed, this is normal and does not indicate an error:

```
DBCC execution completed. If DBCC printed error messages, contact
your
```

```
system administrator.
```

Locking Timeout

With the SET LOCK_TIMEOUT option, it is possible to set the maximum amount of time that SQL Server allows a transaction to wait for the release of a blocked resource.

Syntax

```
SET LOCK_TIMEOUT timeout_period
```

timeout_period is the number of milliseconds that pass before SQL Server returns a locking error. A value of -1 (the default) indicates no timeout period. After you change it, the new setting is in effect for the remainder of the session.

Example

This example sets the lock timeout period to 180,000 milliseconds (3 minutes).

```
SET LOCK_TIMEOUT 180000
```

To determine the current session value, query the @@LOCK_TIMEOUT function.

Example

This example displays the current @@LOCK_TIMEOUT setting.

```
SELECT @@LOCK_TIMEOUT
```

Result

```
-----------
180000

(1 row(s) affected)
```

Table-Level Locking Hints

SQL Server automatically uses a dynamic locking architecture to select the best lock for your client. For performance tuning, however, it is possible to specify table-level locking hints.

CAUTION

Use table-level locking hints with caution—only after you thoroughly understand how your application works and when you are certain that the lock that you request is better than that which SQL Server would use.

The following characteristics apply to table-level locking hints:

- You can specify one or more locking hints for a table.
- Use the *optimizer_hints* portion of the FROM clause in a SELECT, INSERT, UPDATE, or DELETE statement.
- These locking options override corresponding session-level (transaction isolation level) options that were previously specified with the SET statement.

Syntax

```
FROM table_name [WITH (table_hint [,…n])]
```

The following table describes the *table_hint* options.

Option	Description
ROWLOCK	Directs SQL Server to use row locks.
PAGLOCK	Directs SQL Server to use page locks.
TABLOCK	Directs SQL Server to use a shared table lock, thereby allowing others to read—but not update—a table.
TABLOCKX	Directs SQL Server to use an exclusive table lock, thereby preventing other users from reading or updating the table.
NOLOCK	Directs SQL Server not to issue shared locks or to honor exclusive locks—also known as "dirty reads." Equivalent to setting transaction isolation level to READ UNCOMMITED.
HOLDLOCK	Instructs SQL Server to hold a shared lock until completion of the transaction. The HOLDLOCK option is equivalent to setting the transaction isolation level to SERIALIZABLE.
UPDLOCK	Directs SQL Server to use update page locks instead of shared locks while reading a table and holds the locks until the end of the statement or transaction.

Example 1

When your transaction involves an update that modifies a large number of rows in a table, it may be more efficient to specify a table lock.

For example, assume that a library wants to extend the due date for all books by one day to compensate for extreme weather conditions that made it difficult for people to return books on time. The following transaction updates all rows from the loan table by using a shared table lock that is not released until the final transaction is complete.

```
USE library
UPDATE loan WITH (TABLOCKX, HOLDLOCK)
SET due_date = due_date + 1
```

Example 2

In this code segment, locking hints are used with a SELECT statement at the beginning of a transaction. A shared lock is held on the table until the transaction is committed.

```
USE library
BEGIN TRAN
SELECT member_no, lastname, firstname
FROM member WITH (TABLOCK, HOLDLOCK)

… statements that perform updates based on the values returned by
the
SELECT statement

COMMIT TRAN
```

Deadlocks

A deadlock occurs when two transactions have locks on separateobjects and each transaction requests a lock on the other transaction's object.Each transaction must wait for the other to release the lock.

A deadlock can occur when several long running transactions execute concurrently in the same database. A deadlockalso can occur because of the order in which the optimizer processes a complex query, such as a join, in which you cannot necessarily control the order of processing.

How SQL Server Ends a Deadlock

SQL Server ends a deadlock by automatically terminating one of the transactions. The process SQL Server uses is as follows:

1. Roll back the transaction of the deadlock victim.

   In a deadlock, SQL Server gives priority to the transaction that has been processing the longest; that transaction "wins." SQL Server rolls back the transaction with the least amount of time invested.

2. Notify the deadlock victim's application (with message number 1205).
3. Cancel the deadlock victim's current request.
4. Allow the other transaction to continue.

Because of the possibility of termination, each client application should check regularly for message number 1205, which indicates that the transaction was rolled back. For example, in a Visual Basic application a run-time error would occur. If message 1205 is found, the application should attempt the transaction again.

How to Minimize Deadlocks

Although it is not always possible to eliminate deadlocks, you can reduce the risk of a deadlock by observing the following guidelines:

- Use resources in the same sequence in all transactions. For example, if possible, reference tables in the same order in all transactions that reference more than one table.
- Shorten transactions by minimizing the number of steps.
- Shorten transaction times by avoiding queries that affect many rows.
- Use a low isolation level if possible.
- Avoid user interaction in transactions.
- Use bound connections. In a bound connection, two or more connections share a transaction and locks.

Displaying Locking Information

Typically, you use SQL Server Enterprise Manager or the sp_lock system stored procedure to display a report of active locks. You can use SQL Server Profiler to get information on a specific set of transactions. You can also use Microsoft Windows NT Performance Monitor to display SQL Server locking histories.

Current Activity Window

Use the Current Activity option located under the Management folder in SQL Server Enterprise Manager to display information on current locking activity. You can view server activity by user, detail activity by connection, and locking information by object.

sp_lock System Stored Procedure

The sp_lock system stored procedure returns information about active locks in SQL Server.

Syntax

```
EXECUTE sp_lock
```

Result

A typical result set resembles the following.

```
spid   dbid   ObjId       IndId  Type Resource         Mode
Status
------ ------ ----------- ------ ---- ---------------- -------- ----
--
1      1      0           0      DB                    S
GRANT
6      1      0           0      DB                    S
GRANT
8      1      0           0      DB                    S
GRANT
8      2      0           0      DB                    S
GRANT
8      7      0           0      DB                    S
GRANT
8      1      117575457   0      TAB                   Sch-S
GRANT
9      6      0           0      DB                    S
GRANT
10     7      0           0      DB                    S
GRANT
```

The first three columns refer to various IDs: server process (spid), database (dbid), and object (ObjId).

The Type column shows the type of resource that is currently locked. Resource types include: DB (database), EXT (extent), TAB (table), KEY (key), PAG (page), or RID (row identifier), FIL (file), IDX (index).

The Resource column provides information on the resource type that is being locked. For example, a resource description of 1:528:0 indicates that row number 0, on page number 528, on file 1 has a lock applied to it.

The Mode column describes the type of lock that is being applied to the resource. Types of locks include: shared (S), exclusive (X), update (U), intent (IS, IU, or IX), or schema (Sch-S or Sch-M).

The Status column shows whether the lock has been obtained (GRANT), is blocking on another process (WAIT), or is in the process of being converted (CNVT).

SQL Server Profiler

SQL Server Profiler is a tool that monitors server activities. You can collect information about a variety of events by creating traces, which provide a detailed profile of server events. You can use this profile to analyze and resolve server resource issues, monitor login attempts and connections, and correct deadlock problems.

Windows NT Performance Monitor

To view SQL Server locking information with Windows NT Performance Monitor, you can use the SQLServer:Locks object.

Additional Information

To find additional information about locks and current server activity, you can query the syslockinfo, sysprocesses, sysobjects, and syslogins system tables or you can execute the sp_who system stored procedure.

Creating, Executing, and Modifying Stored Procedures

Creating Stored Procedures

You create stored procedures in the current database. Creating a stored procedure is similar to creating a view. First, write and test the Transact-SQL statements that you want to include in the stored procedure. Then, if you receive the results that you expect, create the stored procedure.

You create stored procedures with the CREATE PROCEDURE statement. Consider the following facts when you create stored procedures:

- Stored procedures can reference tables, views, stored procedures, and temporary tables.
- If a stored procedure creates a local temporary table, the temporary table only exists for the purpose of the stored procedure. The table is removed when the stored procedure execution completes.
- A CREATE PROCEDURE statement cannot be combined with other SQL statements in a single batch.
- The CREATE PROCEDURE definition can include any number and type of Transact-SQL statements, with the exception of the following object creation statements: CREATE DEFAULT, CREATE PROCEDURE, CREATE RULE, CREATE TRIGGER, and CREATE VIEW. Other database objects can be created within a stored procedure.
- To execute the CREATE PROCEDURE statement, you must be a member of the sysadmin role, db_owner role, or db_ddladmin role, or you must have been granted CREATE PROCEDURE permission by a member of the sysadmin or db_owner roles.
- The maximum size of a stored procedure is 128 MB.

Partial Syntax

The partial syntax of the CREATE PROCEDURE statement is as follows:

```
CREATE PROC[EDURE] procedure_name
[;number]
     [
          {@parameter data_type} [= default] [OUTPUT]
     ]
     [,…n]
     [WITH {RECOMPILE | ENCRYPTION | RECOMPILE,ENCRYPTION}]
AS
     sql_statement [...n]
```

Example

The following statements create a stored procedure in the library database called *overdue_books* that lists all overdue books.

```
USE library
GO
CREATE PROC dbo.overdue_books
AS
SELECT *
FROM loan
WHERE due_date < GETDATE()
GO
```

The following statement executes the overdue_books stored procedure after it has been created.

```
EXEC overdue_books
```

- **To create a stored procedure**

In this exercise, you will create a stored procedure that lists the total number of books that are loaned to each member.

Open the SQL Server Query Analyzer.

1. Verify that you are using the library database.
2. Write a query that lists the member number, member full name, and total number of books that are loaned. (Hint: You will query the loanhist table.) The result set should sort the list so that the member with the most number of books on loan is displayed at the top.
3. Test your query to ensure that it returns the expected result set.
4. Modify your query to create a stored procedure named *TotalBooksLoaned*.
5. Execute your stored procedure to verify that it works as expected.

Guidelines for Creating Stored Procedures

Consider the following facts and guidelines when you create stored procedures:

- To avoid situations in which the owner of a stored procedure and the owner of the underlying tables differ, the dbo user should own all objects in a database. Because a user can be a member of multiple roles, always specify the dbo user as the owner name when you create the object. Otherwise, the object will be created with your user name as the owner.
- Design each stored procedure to accomplish a single task.
- Create, test, and debug your stored procedure on the server; then test it from the client.
- To distinguish system stored procedures easily, avoid using the "sp_" prefix when you name user-defined stored procedures.
- If you do not want users to be able to view the text of your stored procedures, you must create them using the WITH ENCRYPTION option. Do not delete entries from the syscomments system table. If you do not use WITH ENCRYPTION, users can use SQL Server Enterprise Manager or execute the sp_helptext system stored procedure to view the text of stored procedures in the syscomments system table.
- Stored procedures that require a particular option setting should issue a SET statement at the start of the stored procedure. When an Open Database Connectivity (ODBC) application connects to SQL Server, ODBC automatically sets the following options for the session:

```
SET QUOTED_IDENTIFIER ON
SET TEXTSIZE 2147483647
SET ANSI_DEFAULTS ON
SET CURSOR_CLOSE_ON_COMMIT OFF
SET IMPLICIT_TRANSACTIONS OFF
```

These options are set to increase the portability of ODBC applications. Because DB-Library-based applications generally do not set these options, stored procedures should be tested with the SET options turned both on and off. This ensures that the stored procedures work correctly, regardless of the options that a particular connection may have set, when they are invoked.

Nesting Stored Procedures

Stored procedures can be*nested* , which is when one stored procedure calls another. Characteristics of nesting include the following:

- Stored procedures can be nested to 32 levels. Attempting to exceed 32 levels of nesting causes the entire calling stored procedure chain to fail.
- The current nesting level is returned by the @@NESTLEVEL function.
- If one stored procedure calls a second stored procedure, the second stored procedure can access all the objects that are created by the first stored procedure, including temporary tables.

- If a stored procedure starts a transaction and later calls a second stored procedure that issues a ROLLBACK TRAN statement, the transaction and all modifications made in both stored procedures are rolled back.

Viewing Information about Stored Procedures

You can use the system stored procedures in the following table to find additional information about all types of stored procedures. You can also use SQL Server Enterprise Manager.

Stored procedure	Information
sp_help *procedure_name*	Displays a list of parameters and their data types for the specified stored procedure
sp_helptext *procedure_name*	Displays the text of the specified stored procedure if it is not encrypted
sp_depends *procedure_name*	Lists objects that are dependent upon the specified stored procedure and objects upon which the specified stored procedure is dependent
sp_stored_procedures	Returns a list of stored procedures in the current database

- **To display information about stored procedures**

In this exercise, you will use system stored procedures and SQL Server Enterprise Manager to display information about stored procedures.

1. In SQL Server Query Analyzer, verify that you are using the library database.
2. Execute the sp_help and sp_helptext stored procedures to get information about the TotalBooksLoaned stored procedure.

```
EXEC sp_help TotalBooksLoaned
EXEC sp_helptext TotalBooksLoaned
```

3. Switch to SQL Server Enterprise Manager.
4. Expand the library database, and then click Stored Procedures.
5. If the TotalBooksLoaned stored procedure is not shown in the details pane, right-click Stored Procedures and select Refresh to refresh the view.
6. Double-click the TotalBooksLoaned stored procedure in the details pane to view information about the stored procedure.

Executing Stored Procedures

You can execute a stored procedure by itself or as part of an INSERT statement. Before you execute, however, you must first have been granted EXECUTE permission on the stored procedure.

You can execute system stored procedures from any database because SQL Server always searches the master database when a stored procedure with the "sp_" prefix is executed. Before executing other stored procedures, you must make the database, in which the stored procedure was created, current. Otherwise, you can execute a stored procedure by specifying a full three-part name for the stored procedure.

Executing a Stored Procedure by Itself

You can execute a stored procedure by issuing the EXECUTE statement along with the name of the stored procedure and any parameters.

Syntax

```
[EXEC[UTE]]
     {
          [@return_status =] {procedure_name [;number]
@procedure_name_var}
     }
     [[@parameter =] {value | @variable [OUTPUT] | [DEFAULT]] [,
…n]}
   [WITH RECOMPILE]
```

Example

The following statement executes the overdue_books stored procedure that lists all overdue books in the library database after making the library database current.

```
USE library
EXEC overdue_books
```

The following statement uses a three-part name to execute the overdue_books stored procedure that lists all overdue books in the library database, without making the library database current.

```
EXEC library.dbo.overdue_books
```

TIP

You can execute a stored procedure by specifying its name without the EXEC[UTE] statement if it is the first line of a batch. However, you should always use the EXEC[UTE] statement to prevent errors occurring if you insert another line at the beginning of the batch. If you try to execute a stored procedure on any line other than the

first line in a batch and you do not specify the EXEC[UTE] statement, an "Incorrect syntax" error is returned.

Executing a Stored Procedure Within an INSERT Statement

The INSERT statement can use a result set that is returned from a local or remote stored procedure. The INSERT statement loads a table with data that is returned from SELECT statements in the stored procedure. The data types returned by the stored procedure must match those of the table referenced in the INSERT statement.

Stored procedures that are executed within an INSERT statement must return a relational result set. For example, you could not use a COMPUTE BY clause in a SELECT statement in a stored procedure that will be called from an INSERT statement.

Example

The following statements create the Employee_Customer stored procedure, which is called from an INSERT statement to insert employees into the Customers table of the Northwind database.

```
USE Northwind
GO
CREATE PROC dbo.Employee_Customer
AS
SELECT
   UPPER(SUBSTRING(LastName, 1, 4) + SUBSTRING(FirstName, 1, 1)),
   'Northwind Traders', RTRIM(FirstName) + ' ' + LastName,
   'Employee', Address, City, Region, PostalCode, Country,
   ('(206) 555-1234' + ' x' + Extension), NULL
FROM Employees
WHERE CONVERT(varchar(10), HireDate, 101) =
CONVERT(varchar(10), GETDATE(), 101)
GO
```

The following statements execute the stored procedure:

```
INSERT Customers
EXEC Employee_Customer
```

The number of employeeshired on today's date is added to the Customers table.

Explicitly Recompiling Stored Procedures

You may need to recompile a stored procedure explicitly when:

- The stored procedure executes infrequently.

- You want to execute the stored procedure with very different parameter values. When you pass values that return widely, varying result sets to a stored procedure, it is not always optimal to execute the cached execution plan.

SQL Server provides three methods for recompiling a stored procedure explicitly: the CREATE PROCEDURE…WITH RECOMPILE statement, the EXECUTE…WITH RECOMPILE statement, and the sp_recompile system stored procedure.

CREATE PROCEDURE… WITH RECOMPILE

The CREATE PROCEDURE... WITH RECOMPILE statement creates a stored procedure for which the query plan is not cached when the stored procedure is executed. Instead, the stored procedure is recompiled and optimized, and a new query plan is created each time the stored procedure is executed.

Use this method if the stored procedure that you are creating is executed only periodically, such as monthly or quarterly.

Example

The following statement creates a stored procedure called *titlecount* that is recompiled each time it is executed:

```
USE library
GO
CREATE PROC titlecount @title_no title_no WITH RECOMPILE
AS
     SELECT COUNT(*) FROM loanhist
     WHERE title_no = @title_no
GO
```

EXECUTE… WITH RECOMPILE

The EXECUTE... WITH RECOMPILE statement creates a new query plan during execution of the stored procedure. The new execution plan is placed in the cache.

Use this method if the parameter that you are passing varies greatly from those that are usually passed to the stored procedure. Because this optimized plan is the exception rather than the rule, the first time that you execute the stored procedure with a typical parameter, you should specify the WITH RECOMPILE option with the EXECUTE statement again.

Example

This example recompiles the sp_help system stored procedure at the time it is executed.

```
EXEC sp_help WITH RECOMPILE
```

Execute the sp_recompile System Stored Procedure

Execute the sp_recompile system stored procedure to force a recompile of a specific stored procedure or of all stored procedures or triggers that reference a specified object in the current database. The stored procedures are recompiled the next time they are executed.

For example, if you have added a new index to a table that is referenced by a stored procedure and you believe that the performance of the stored procedure will benefit from the new index, execute the sp_recompile system stored procedure and specify the name of the table.

Example

This example forces a recompile of stored procedures that reference the title table in the library database.

```
EXEC sp_recompile title
```

Executing Extended Stored Procedures

Extended stored procedures are DLLs that increase SQL Server functionality. They are executed the same way as stored procedures, and they support return status codes and output parameters. Note that extended stored procedures:

- Communicate with SQL Server by using the Open Data Services application programming interface (API), not ODBC or OLE DB.
- Can include C and C++ features that are not available in Transact-SQL.
- Can contain multiple functions.
- Can be called from a client or SQL Server.
- Must be executed from the master database or by explicitly specifying *master* as part of the extended stored procedure name. You can call extended stored procedures from user-defined system stored procedures that you create.

Example

This example executes the xp_cmdshell extended stored procedure, which makes it possible to execute operating system commands from within SQL Server. The command returns a list of files and subdirectories.

```
EXEC master..xp_cmdshell 'dir c:\mssql7'
```

The following table lists some commonly used extended stored procedures:

Extended stored procedure	Description
xp_cmdshell	Executes a given command string as an operating system command shell and returns output as rows of text
xp_logevent	Logs a user-defined message in the SQL Server error log and in the Microsoft Windows NTapplication log

You can also create your own extended stored procedures. Generally, you call extended stored procedures to communicate with other applications or the operating system. For example, extended stored procedures in Sqlmap70.dll, such as xp_sendmail and xp_readmail, allow you to work with e-mail messages from within SQL Server.

Altering and Dropping Stored Procedures

Stored procedures are often modified in response to requests from users or in response to changes in the underlying table definitions.

Altering Stored Procedures

To modify an existing stored procedure and retain permission assignments, use the ALTER PROCEDURE statement. SQL Server replaces the previous definition of the stored procedure when it is altered with ALTER PROCEDURE.

Caution

You should not modify system stored procedures. If you want to change the functionality of a system stored procedure, create a new user-defined stored procedure by copying the statements from an existing system stored procedure, and then modify it to meet your needs.

Consider the following facts when you use the ALTER PROCEDURE statement:

- If you want to modify a stored procedure that was created with any options, such as the WITH ENCRYPTION option, you must include the option in the ALTER PROCEDURE statement to retain the functionality that the option provides.
- ALTER PROCEDURE alters only a single procedure. If your procedure calls other stored procedures, the nested stored procedures are not affected.
- Permission to execute this statement defaults to the creators of the initial stored procedure and members of the sysadmin, db_owner, and db_ddladmin roles. You cannot grant permission to execute ALTER PROCEDURE.

Syntax

```
ALTER PROC[EDURE] procedure_name [;number]
     [{@parameter data_type }[VARYING][ = default] [OUTPUT]]
          [,...n]
 [WITH {RECOMPILE | ENCRYPTION | RECOMPILE,ENCRYPTION}]

[FOR REPLICATION]
 AS
     sql_statement [...n]
```

Example

The following example modifies the overdue_books stored procedure to select specific columns from the OverdueView view rather than all columns from the loan table.

```
USE library
GO
ALTER PROC overdue_books
AS
SELECT CONVERT(char(8), due_date, 1) date_due,
isbn, copy_no,
SUBSTRING(title, 1, 30) title, member_no, lastname
FROM OverdueView
ORDER BY due_date
GO
```

Dropping Stored Procedures

Use the DROP PROCEDURE statement to remove user-defined stored procedures from the current database.

Before you drop a stored procedure, execute the sp_depends stored procedure to determine whether objects depend on the stored procedure you want to drop.

Syntax

```
DROP PROC[EDURE]  procedure [, ...n]
```

Example

This example drops the overdue_books stored procedure.

```
USE library
GO
DROP PROC overdue_books
```

Introduction to Views

Views are a powerful way to create permanent query definition. This lesson defines views and discusses the advantages of views to administrators, programmers, and users.

After this lesson you will be able to:

- Define what a view is.
- Discuss the advantages of using views.

Estimated lesson time: 25 minutes

What Is a View?

When you use a *view*, you can store a predefined query as an object in the database for later use.The tables queried in a view are called *base tables*. With a few exceptions, any SELECT statement can be named and stored as a view.

Example

This example creates the TitleView view in the library database. The view displays two columns in the title table.

```
USE library

CREATE VIEW dbo.TitleView
AS
SELECT title, author
FROM title
```

Query

```
SELECT * FROM TitleView
```

Result

```
title                                author
------------------------------------ ------------------------------
Last of the Mohicans                 James Fenimore Cooper
The Village Watch-Tower              Kate Douglas Wiggin
Self Help; Conduct & Perseverance    Samuel Smiles
...
Julius Caesar's Commentaries on the... Julius Caesar
Frankenstein                         Mary Wollstonecraft Shelley

(50 row(s) affected)
```

Advantages of Views

Views offer several advantages, including focusing data for users, masking data complexity, simplifying permission management, and organizing data for export to other applications.

Focus the Data for Users

Views create a controlled environment that allows access to specific data and conceals other data. Data that is unnecessary, sensitive, or inappropriate can be left out of a view. Users can manipulate the display of data in a view, similar to a table. In addition, with the proper permissions and a few restrictions, users can modify the data that a view produces.

Mask Database Complexity

Views shield the complexity of the database design from the user. This means that developers can change the design without affecting user interaction with the database. In addition, users can see a friendlier version of the data by using names that are easier to understand than the cryptic names that are often used in databases.

Complex queries, including distributed queries to heterogeneous data, can also be masked through views. The user queries the view instead of writing the query or executing a script.

Simplify Management of User Permissions

Instead of granting permission for users to query specific columns in base tables, database owners can grant permission for users to query data through views only. This type of querying also protects changes in the design of the underlying base tables. Users can continue to query the view without interruption.

Organize Data for Export to Other Applications

You can create a view based on a complex query that joins two or more tables and then export the data to another application for further analysis.

Information Schema Views

Information schema views are a good example of the benefits of using views instead of querying base tables directly. With*information schema views* , SQL Server can present system data in a consistent manner, even when significant changes have been made to the system tables. This ability allows applications that retrieve system information from information schema views to work properly.

For example, to retrieve information about the tables in the current database, query the information_schema.tables information schema view rather than the sysobjects system table.