Designed for Microsoft® Windows NT® Windows® 95

CD-ROM Included

Microsoft® Programming Series

Inside Microsoft® SQL Server™ 6.5

The Developer's Guide to Design, Architecture, and Implementation from a Leading Microsoft Expert

Ron Soukup

Foreword by Jim Gray, Head of the Microsoft San Francisco Research Lab

Microsoft Press

Inside Microsoft® SQL Server™ 6.5

Ron Soukup

Microsoft Press

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data
Soukup, Ron.
Inside Microsoft SQL Server 6.5 / Ron Soukup.
p. cm.
Includes index.
ISBN 1-57231-331-5
1. Database management. 2. SQL Server. I. Title.
QA76.9.D3S66 1997
005.75'85--dc21 97-37611
CIP

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 MLML 2 1 0 9 8 7

Distributed to the book trade in Canada by Macmillan of Canada, a division of Canada Publishing Corporation.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office. Or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at mspress.microsoft.com.

Acquisitions Editor: David Clark
Project Editor: Lisa Theobald
Technical Editor: John Conrow

To Kay, Kelly, and Jamie,
for your love and support during
the years of ship crunch.
And
To the SQL Server Development Team.
Working with you has been the greatest
privilege of my career.

CONTENTS

2

A Tour of Microsoft SQL Server

Introduction

Microsoft SQL Server is a high-performance, client/server relational database management system (RDBMS). It was designed to support high-volume transaction processing (such as that for online order entry, inventory, accounting, or manufacturing) as well as data warehousing and decision-support applications (such as sales analysis applications) on Microsoft Windows NT Server–based networks. SQL Server is fully operational on all hardware architectures supported by Windows NT, including Intel, DEC Alpha AXP, MIPS R4000, and Motorola PowerPC-based systems. For all these hardware platforms, SQL Server versions are built simultaneously from the same source code baseline, and all versions ship together on the same CD-ROM. SQL Server also provides many client tools and networking interfaces for the Microsoft Windows 95, Windows 3.1, and MS-DOS operating systems. And because of SQL Server's open architecture, other systems (for example, UNIX-based systems) can interoperate with it as well.

SQL Server is part of the core of a family of integrated products, including development tools, systems management tools, distributed system components, and open development interfaces, as shown in Figure 2-1 on the following page. It is also a key part of Microsoft BackOffice.

This book focuses on the capabilities and uses of the SQL Server engine; this chapter provides an overview of the entire SQL Server family of components and describes the features and benefits of each component. Understanding these features and benefits will prove helpful to you as you develop applications.

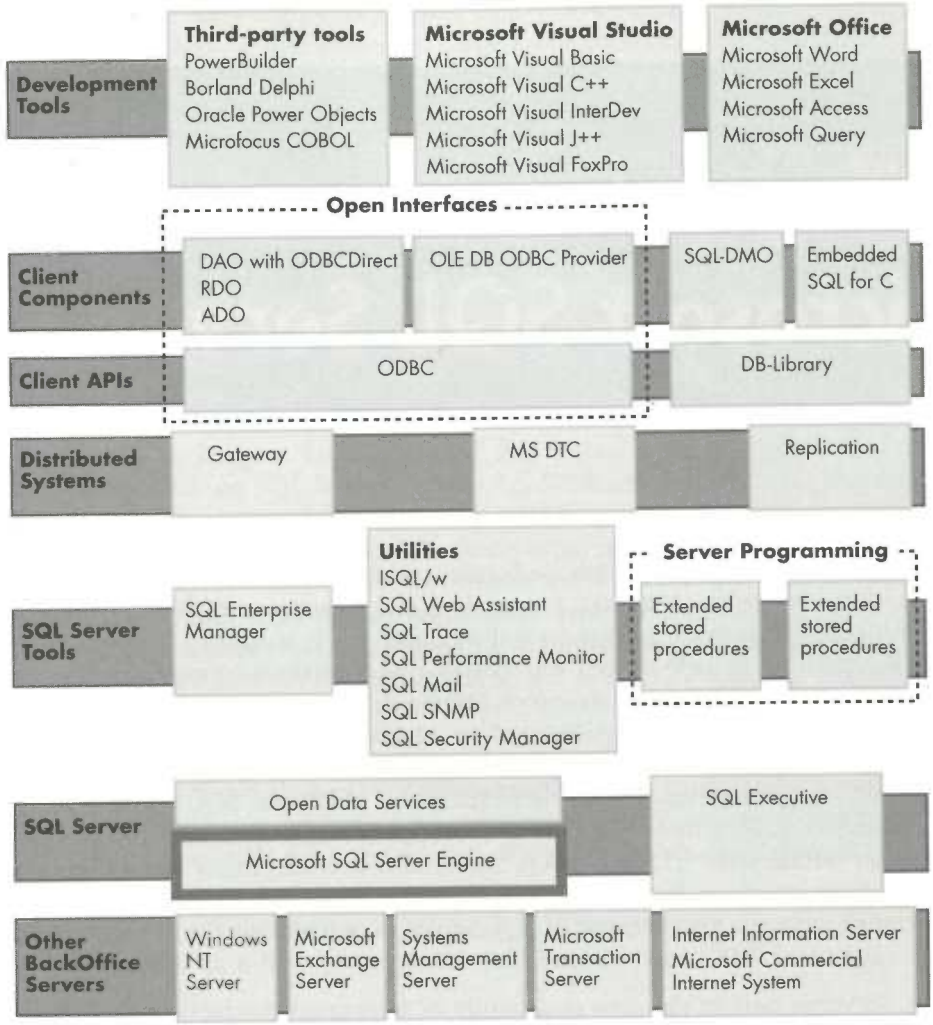


Figure 2-1. *SQL Server and its family of integrated components.*

The SQL Server Engine

The Microsoft SQL Server engine is designed to support a variety of demanding applications, such as online transaction processing (OLTP) and decision-support applications. At the core of its decision-support capabilities is Transact-SQL, Microsoft's version of Structured Query Language. Beneath this query language are the components that support transaction processing and recoverability.

Transact-SQL

Industrywide, SQL is a well-known and widely used data access tool. Every mainstream database management system (DBMS) product implements SQL in some way. Transact-SQL (often referred to as "T-SQL") is a powerful and unique superset of the SQL standard.

The SQL SELECT statement provides tremendous power and flexibility for retrieving information. Data from multiple tables can be easily projected and the results returned in tabular format with information chosen and correctly combined from the multiple tables. Check out the following two tables from the *pubs* sample database. (The *pubs* database, used for many examples in this book, is installed when Microsoft SQL Server is installed. For brevity, an abbreviated amount of the data will sometimes be used, as is true in this example.)

publishers Table

pub_id	*pub_name*	*city*	*state*
0736	New Moon Books	Boston	MA
0877	Binnet & Hardley	Washington	DC
1389	Algodata Infosystems	Berkeley	CA

titles Table

title_id	*title*	*pub_id*
BU1032	The Busy Executive's Database Guide	1389
BU1111	Cooking with Computers: Surreptitious Balance Sheets	1389
BU2075	You Can Combat Computer Stress!	0736
BU7832	Straight Talk About Computers	1389
MC2222	Silicon Valley Gastronomic Treats	0877
MC3021	The Gourmet Microwave	0877
MC3026	The Psychology of Computer Cooking	0877

The following simple SELECT statement logically joins the *titles* and *publishers* tables to project the names of the book titles with the names of the companies publishing each title.

```
SELECT title, pub_name, city, state
FROM titles, publishers
WHERE titles.pub_id = publishers.pub_id
```

Here's the result:

```
title                                    pub_name                 city          state
The Busy Executive's Database Guide      Algodata Infosystems     Berkeley      CA
Cooking with Computers: Surreptitious Balance Sheets
                                         Algodata Infosystems     Berkeley      CA
You Can Combat Computer Stess!           New Moon Books           Boston        MA
Straight Talk About Computers            Algodata Infosystems     Berkeley      CA
Silicon Valley Gastronomic Treats        Binnet & Hardley         Washington    DC
The Gourmet Microwave                    Binnet & Hardley         Washington    DC
The Psychology of Computer Cooking       Binnet & Hardley         Washington    DC
```

This query, a simple SQL statement, shows that standard SQL provides a powerful way to query and manipulate data. (In Chapters 7 and 10, we'll explore SQL queries in much greater depth.)

The National Institute of Standards and Technology (NIST) has certified Microsoft SQL Server version 6.5 as compliant with the American National Standards Institute (ANSI) SQL-92 standard. However, considerably more power is available in Transact-SQL because of its unique extensions to the standard.

Standards and Testing

Although the ANSI standard is commonly referred to as "SQL-92," the official standard is ANSI X3.135-1992 and is entitled "American National Standards Institute Database Language-SQL." "X3H2" is the designator for the ANSI SQL committee. NIST, a division of the United States Department of Commerce, conducts a suite of tests (which vendors pay the costs of running) to certify compliance with the standard. You can find a summary of products currently certified as compliant at ftp://speckle.ncsl.nist.gov/sql-testing/VPLs.

Transact-SQL extensions

Transact-SQL provides a number of capabilities that extend beyond typical implementations of SQL. Queries that are difficult to write in standard SQL can be easily and efficiently written using these capabilities. Some of my favorites include the ability to embed additional SELECT statements in the SELECT list and the ability to drill into a result set by further selecting data directly from a SELECT statement,

a feature known as a *derived table*. Transact-SQL provides many system functions for dealing with strings (for finding substrings and so on), for converting datatypes, and for manipulating and formatting date information. Transact-SQL also provides mathematical operations such as square root. In addition, special operators, such as CUBE and ROLLUP, allow multidimensional analysis to be efficiently projected at the database server, where the analysis can be optimized as part of the execution plan of a query. The CASE operator allows for complex conditional substitutions to be made easily in the SELECT statement. Multidimensional (sometimes referred to as OLAP, or online analytic processing) operators, such as CUBE, and conditional operators, such as CASE, are especially useful in implementing data warehousing solutions with SQL Server.

The query optimizer

In Transact-SQL, a cost-based query optimizer determines the likely best way to access data. This allows you to concentrate on defining your query criteria rather than defining how the query should be executed. For example, this nonprocedural approach eliminates the need for you to know which indexes exist and which, if any, should be used. Would it be more efficient to incur additional I/Os to read index pages in addition to data pages, or would it be better just to scan the data and then sort it? The optimizer automatically, invisibly, and efficiently resolves these types of important questions for you.

The SQL Server optimizer maintains statistics about the volume and dispersion of data, which it then uses to estimate the plan most likely to work best for the operation requested. Because a cost-based optimizer is by definition probability-based, an application might want to override the optimizer in some specialized cases. In your application, you can specify *optimizer hints* that will direct the execution plan chosen. In addition, you can use SQL Server's SHOWPLAN feature, which explains the execution plan chosen, provides insight into why it was chosen, and even allows for tuning of the application and database design.

The programmable server

Transact-SQL provides programming constructs—such as variables, conditional operations (IF-THEN-ELSE), and looping—that can dramatically simplify application development by allowing you to use a simple SQL script rather than a third-generation programming language (3GL). These branching and looping constructs can dramatically improve performance in a client/server environment by eliminating the need for network conversations. Minimizing network latency is a key aspect of maximizing client/server application performance. For example, instead of returning a value to the calling application, which requires that the application evaluate and subsequently issue another request, you can build conditional logic directly into the SQL batch file so that the routine is completely evaluated and executed at the server.

You can use Transact-SQL to write complex batches of SQL statements. (A batch of SQL statements in a complex application can be up to several hundred lines long.) An important new capability of SQL Server 6.5 is the SQL Debugging Interface (SDI), which allows debuggers such as those available with Microsoft Visual Studio 97 to fully debug Transact-SQL routines, including stepping through the statements, setting breakpoints, and setting watchpoints on Transact-SQL variables.

Stored procedures

Simply put, *stored procedures* are collections of SQL statements stored within a SQL Server database. You can code complex queries and transactions into stored procedures and then invoke them directly from the front-end application. Whenever a dynamic SQL command is sent to a database server for processing, the server must parse the command, check its syntax for sense, determine whether the requester has the permissions necessary to execute the command, and formulate an optimal execution plan to process the request. Stored procedures execute faster than dynamic SQL batches, sometimes dramatically faster, because they eliminate the need for reparsing and reoptimizing the requests each time they are executed. SQL Server supports stored procedures that let developers store *groups* of compiled SQL statements on the server for later recall, to limit the overhead when the procedures are subsequently executed.

Stored procedures differ from ordinary SQL statements and from batches of SQL statements in that they're checked for syntax and compiled the *first time* they are executed. SQL Server stores this compiled version and then uses it to process subsequent calls, resulting in faster execution times. Stored procedures can also accept parameters, so a single procedure can be used by multiple applications using different input data.

Even if stored procedures provided no performance advantage (which, of course, they do), there would still be a compelling reason to use them: they provide an important layer of insulation from changes in business practices. Suppose, for example, that an application is used to maintain a mailing list for a retailer's catalog distribution. Subsequent to the application being deployed, a change in criteria and logic (that is, the business rules) occurs, thus affecting which customers should automatically receive new catalogs. If the business rules had been programmed directly into the company's applications, every application would need to be modified, likely an expensive and time-consuming operation. Furthermore, if multiple developers worked on the applications, the rules might not have been programmed with the exact same semantics by every programmer. A stored procedure, on the other hand, could be modified *once,* in seconds, at the server. The applications would not need to be changed or even restarted. The next time each application executed the stored procedure, the new rules would be in place automatically.

In addition to providing a performance advantage, stored procedures can provide an important security function. By granting users access to a stored procedure but not to the underlying tables, you can allow them to access or manipulate data only in the way prescribed by the stored procedure.

Extended stored procedures

A unique capability of Microsoft SQL Server, *extended stored procedures* allow developers to extend the programming capabilities provided by Transact-SQL and to access resources outside of SQL Server. Messaging integration, security integration, the ability to write HTML (Hypertext Markup Language) files (files formatted for use on the Internet), and much of the power of SQL Enterprise Manager are all implemented using extended stored procedures. You can create extended stored procedures as external dynamic link libraries (DLLs). (DLLs are typically written in C and C++, although implementation in other languages is also possible.)

For example, you could write a DLL to establish a modem connection, dial the ACME Credit Service, and return a status indicating credit approval or rejection. (The C language more readily lends itself to particular tasks because of such language constructs as arrays, structures, and pointers.) For example, writing a financial function that uses recursion in C (for example, the internal rate of return, or IRR) might be more efficient that writing it as a Transact-SQL stored procedure. *Open Data Services (ODS)* is an application programming interface that lets you build extended stored procedures that can return self-describing result sets to the calling client applications, just as a "normal" procedure would.

Extended stored procedures allow even Microsoft to extend SQL Server. Good engineering practices dictate that where code does not benefit from being shared or is not in common, it should be segregated and isolated. With this principle in mind, Microsoft added integration with messaging via MAPI as a set of extended stored procedures (**xp_sendmail, xp_readmail,** and so on) instead of directly modifying the SQL Server engine. Extended stored procedures allow us to add powerful features without any chance of disrupting the core server engine so that more features can be added quickly, with less risk of destabilizing the server. And because the code is loaded dynamically, the DLL is loaded only if a routine is implemented as an extended stored procedure, so the memory footprint of SQL Server does not grow for services that aren't being used.

DBMS-Enforced Data Integrity

A database is only as useful as the user's confidence in it. That's why the server must enforce data integrity rules and business policies. SQL Server enforces data integrity within the database itself, guaranteeing that complex business policies will be followed and that mandatory relationships between data elements are complied with.

Because SQL Server's client/server architecture allows you to use a variety of front-end applications to manipulate and present the same data from the server, it would be cumbersome to encode all the necessary integrity constraints, security permissions, and business rules into each application. If business policies were all coded in the front-end applications, *every* application would need to be modified *every time* a business policy changed. Even if you attempted to encode business rules into every client application, the danger of an application misbehaving still exists. Most applications cannot be fully trusted. Only the server can act as the final arbiter, and the server must not provide a back door for a poorly written or malicious application to subvert its integrity.

SQL Server uses advanced data integrity features, such as declarative referential integrity (DRI), datatypes, defaults, constraints, rules, stored procedures, and triggers, to enforce data integrity. Each of these features has its own use within a database; combining these integrity features can make your database flexible and easy to manage, yet secure.

Declarative Referential Integrity

A central tenet of relational database theory is that every *tuple* of every *relation* (more colloquially, every *row* of every *table*) can be uniquely identified. The attribute or combination of attributes (the column or combination of columns) that ensures uniqueness is known as the *primary key*. A table can have only one primary key. SQL Server allows you, when defining a table, to designate the column(s) that make up the primary key. This is known as a *PRIMARY KEY constraint*. SQL Server uses this PRIMARY KEY constraint to guarantee that the uniqueness of the designated column(s) is never violated.

Sometimes multiple columns of a table can uniquely identify a row—for example, an employee table might have an employee ID (*emp_id*) and a social security number (*soc_sec_num*) column, and both are considered unique. Such columns are often referred to as *alternate* or *candidate keys*. These keys must also be unique. Although a table can have only one primary key, it can have multiple alternate keys. SQL Server supports the multiple alternate key concept via *UNIQUE constraints*. When a column or combination of columns is declared unique, SQL Server prevents any record being added or updated that would violate this uniqueness.

Assigning an arbitrary unique number as the primary key when no natural or convenient key exists is often most efficient. For example, businesses commonly use customer numbers or account numbers as unique identifiers or primary keys. SQL Server makes it easy to efficiently generate unique numbers by allowing one column in a table to have the *Identity property*. You use the Identity property to make sure that each value in the column is unique and that the values will

increment (or decrement) by the amount you specify from a starting point that you specify. (A column having the Identity property will typically also have a PRIMARY KEY or UNIQUE constraint, but this is not required.)

SQL Server enforces logical relationships between tables with *FOREIGN KEY constraints*. A *foreign key* in a table is a column or combination of columns that match the primary key (or possibly an alternate key) of another table. The logical relationship between those two tables is the basis of the relational model.

For example, the simple SELECT example shown earlier in this chapter includes a *titles* table and a *publishers* table. The *titles* table column *title_id* (title ID) is its primary key. The *publishers* table column *pub_id* (publisher ID) is its primary key. The *titles* table also includes a *pub_id* column, which is not the primary key because a publisher can publish multiple titles. Instead, *pub_id* is a *foreign key,* and it references the primary key of the *publishers* table. After this relationship is declared when the table is defined, SQL Server ensures that a title cannot be entered unless a valid publisher for it is in the database and that a publisher cannot be deleted if any titles in the database reference that publisher.

To further enforce data integrity, SQL Server makes sure that any data entered matches the type and range of the specified data type and, for example, allows a NULL value to be entered only if the column has been declared as allowing NULLs. SQL Server supports a wide range of datatypes, allowing for great flexibility with efficient storage.

Datatypes

SQL Server datatypes provide the simplest form of data integrity by restricting the types of information (for example, characters, numbers, or dates) that can be stored in the columns of the database tables. You can also design your own datatypes (*user-defined* datatypes) to supplement those supplied by the system. For example, you could define a *state_code* datatype as two characters (CHAR(2)); SQL Server would then accept only two-character state codes. A user-defined datatype can be used to define columns in *any* table. An advantage of user-defined datatypes is that rules and defaults, which are discussed in the next two sections, can be bound to them for use in multiple tables, eliminating the need to include these types of checks in the front-end application.

CHECK Constraints and Rules

CHECK constraints and *rules* are integrity constraints that go beyond those implied by a column's datatype. Whenever a user enters a value, SQL Server checks that value against any CHECK constraint or rule created for the specified column to ensure that only values that adhere to the definition of the constraint or rule are accepted. Although CHECK constraints and rules are essentially equivalent in functionality, CHECK constraints are easier to use and provide more

flexibility. A CHECK constraint can be conveniently defined when a column is defined, and constraints can be defined on multiple columns. Rules, however, must be defined and then bound to a column or user-defined datatype separately. While a column or user-defined datatype can have only one rule associated with it, a CHECK constraint can reference multiple columns in the same table or it can reference one of the built-in functions that SQL Server provides.

Both CHECK constraints and rules can require that a value fall within a particular range, match a particular pattern, or match one of the entries in a specified list. An advantage of CHECK constraints is that they can depend on either the value of another field or fields in the row or on the value returned by one of the system-supplied functions. A rule cannot reference other fields. As an example of applying a CHECK constraint or rule, a database containing information on senior citizens could have the CHECK constraint or rule *"age field must contain a value between 65 and 120 years."* A birth certificate database could require that the date in the *birth_date* field be the current date—checking the value returned by SQL Server's built-in GETDATE()function—or that it be some date prior to the current date.

Defaults

Defaults allow you to specify a value that SQL Server inserts if no explicit value is entered in a particular field. For example, you could set the current date as the default value for an *order_date* field in a customer order record. Then, if a user or front-end application doesn't make an entry in the *order_date* field, SQL Server automatically inserts the current date. You can also use the keyword DEFAULT as a placeholder in an INSERT or UPDATE statement, instructing SQL Server to set the value to the declared default value.

Triggers

Triggers are a special type of stored procedure. Stored procedures can be executed only when explicitly called; triggers are automatically invoked, or "triggered," by SQL Server, and this is their main advantage. Triggers are associated with particular pieces of data and are called automatically whenever an attempt to modify that data is made, no matter what causes the modification (a user's entry or an application action).

Conceptually, triggers are similar to a CHECK constraint or rule. SQL Server automatically activates triggers, constraints, and rules whenever an attempt is made to modify the data they protect. CHECK constraints and rules then perform fairly simple types of checks on the data—for example, *"make sure the age field has a value between 0 and 120."* Triggers, on the other hand, can perform extremely elaborate restrictions on the data, which helps to ensure that the rules by which your business operates cannot be subverted. Because triggers are a form

of stored procedure, they have the full power of the Transact-SQL language at their disposal and they can invoke other stored and extended stored procedures. You can write a trigger that enforces complex business rules, such as this:

> Don't accept an order
> If the customer has any past due accounts with us
> OR
> If the customer has a bad credit rating by ACME Credit Service (with the trigger calling an extended procedure that automatically dials up ACME to get the credit rating)
> OR
> If the order is for more than $50,000 and the customer has had an account with us for less than six months

This is a powerful integrity check. Yet the trigger to enforce it is simple to write.

Triggers can also enforce *referential integrity*, ensuring that relationships between tables are maintained. For example, a trigger can prohibit a customer record from being deleted if open orders exist for the customer or it can prohibit any new order for a customer for which no record exists. Triggers can go beyond simply insisting that relationships exist: they can perform *referential actions*. This means that triggers can cause changes to ripple through to other tables. For example, if you want to drop a delinquent customer from your system and delete all of that customer's active orders, a trigger on the *customer* table could automatically delete all entries in the *orders* table.

Triggers automatically execute whenever a specified change to a data object is attempted. A trigger executes once per statement, even if multiple rows are affected. It has access to the before and after images of the data. (These before and after images are reconstructed from the transaction log into pseudo tables that can be accessed from within the trigger.) The trigger can then take further action, including rolling back the transaction. Although you can use triggers to enforce referential integrity, it is usually more convenient to establish these relationships when you create the tables by using declarative referential integrity.

Transaction Processing

Transaction processing guarantees the consistency and recoverability of SQL Server databases. A *transaction* is the basic unit of work under SQL Server. Typically, it consists of several SQL commands that read and update the database, but the update is executed only when a COMMIT command is issued. (Note that the example below is pseudocode and that error handling is required to achieve the behavior described.)

Transaction processing in SQL Server assures that all transactions are performed as a single unit of work—even in the presence of a hardware or general system

failure. Such transactions are referred to as having the *ACID properties*: atomicity, consistency, isolation, and durability. In addition to the explicit multistatement transactions such as those provided in the *DEBIT_CREDIT* example below, SQL Server guarantees that a single command that affects multiple rows maintains the ACID properties.

Here is an example in pseudocode of an ACID transaction, followed by an explanation of each of the ACID properties.

```
BEGIN TRANSACTION DEBIT_CREDIT
Debit Savings account $1000
Credit Checking account $1000
COMMIT TRANSACTION DEBIT_CREDIT
```

Atomicity

SQL Server guarantees the *atomicity* of its transactions. With atomicity, each transaction is treated as all-or-nothing—it either commits or aborts. If a transaction commits, all of its effects remain. If it aborts, all of its effects are undone. In the *DEBIT_CREDIT* example above, if the savings account debit is reflected in the database but the checking account credit is not, funds will essentially disappear from the database; that is, funds will be debited from the savings account but never credited to the checking account. If the reverse occurred (if the checking account were credited and the savings account were *not* debited), the customer's account would mysteriously increase in value without a corresponding customer cash deposit or account transfer. Because of SQL Server's atomicity feature, both the debit and credit must be completed or neither event is completed.

Consistency

The *consistency* property ensures that a transaction will not allow the system to enter an incorrect logical state—the data must always be logically correct. Constraints and rules are honored, even in the event of a system failure. For the *DEBIT_CREDIT* example, the logical rule is that money cannot be created or destroyed—a corresponding, counter-balancing entry must be made for each entry. (Consistency is implied by, and for most situations is redundant to, atomicity, isolation, and durability.)

Isolation

Isolation separates concurrent transactions from the updates of other incomplete transactions. In the *DEBIT_CREDIT* example, another transaction cannot see the "work in progress" while the transaction is being carried out. For example, if another transaction read the balance of the savings account after the debit occurred, and then the *DEBIT_CREDIT* transaction was aborted, the other transaction would be working from a balance that never logically existed.

Isolation among transactions is accomplished automatically by SQL Server. It locks data to allow multiple concurrent users to work with data, but it prevents side effects that could distort the results and make them different than would be expected if users serialized their requests (that is, if requests were queued and ran one at a time). This *serializability* feature is one of the isolation levels that SQL Server supports. SQL Server supports multiple degrees of isolation levels that allow you to make the appropriate trade-off between how much data to lock and how long locks must be held. This trade-off is known as *concurrency* versus *consistency*. Locking reduces concurrency (because locked data is unavailable to other users), but it provides the benefit of higher consistency. (I'll discuss locking in greater detail in Chapter 13.)

Durability

After a transaction commits, SQL Server's *durability* property ensures that its effects will persist even if a system failure occurs. Conversely, if a system failure occurs while a transaction is in progress, the transaction will be completely undone, leaving no partial effects on the data. For example, if a power outage occurs in the midst of a transaction before the transaction is committed, the entire transaction will be automatically rolled back when the system is restarted. If the power fails immediately after the acknowledgment of the commit is sent to the calling application, the transaction is guaranteed to exist in the database. Write-ahead logging and automatic rollback and rollforward of transactions during the recovery phase of starting SQL Server assure durability.

Symmetric Server Architecture

SQL Server uses a single-process, multithreaded architecture known as *Symmetric Server Architecture* that provides scalable high performance with efficient use of system resources. With Symmetric Server Architecture, only one memory address space is provided for the DBMS, eliminating the overhead of having to manage shared memory.

Traditional Process/Thread Model

To understand and contrast the architecture of Microsoft SQL Server, it is useful for you to first understand the traditional architectures that have been used by UNIX-based DBMS products. UNIX-based DBMS products are usually structured in one of two ways. In the first way, multiple processes (or *shadow processes*) are used, with one process per user, which makes the system quite resource intensive. The second type of architecture employs a single process that tries to simulate an operating system threading facility by moving in a round-robin way among multiple requests, maintaining a stack for each request and switching to that specific stack for whatever unit is being executed.

NOTE A *stack* is a LIFO (last-in, first-out) data structure kept in memory that basically serves as the control block for the executable unit to the operating system (a *thread* on Microsoft Windows NT, often called a *lightweight process* on other operating systems). A stack stores status data such as function call addresses, passed parameters, and some local variables.

In the first approach, because each process has its own address space, processes must resort to shared memory to communicate with one another. Unfortunately, shared memory is less efficient to use than the private memory of a process's own address space because of the weight of synchronization mechanisms (semaphores, mutexes, and so on) that are needed to avoid collisions while accessing shared memory. In addition, the implementation of stack switching and efficient access to shared memory adds overhead and complexity. Adding complexity to a system is never good. The best way to avoid bugs in software and maximize performance is to keep code simple and, better yet, to write no new code when an existing tried-and-true service exists.

In the second approach—simulated multithreading—the DBMS performs duties that should be performed by the operating system: at best, the DBMS can only simulate operating system behavior and give the illusion of providing threads. Typically, using such an architecture requires that the executing task be trusted to "yield" back to the system so another task can be run. If the task does not yield (because of software or hardware failure), *all* other tasks will be severely, perhaps fatally, affected. Trust is not a good basis upon which to build a multiuser system. Furthermore, if the schedulable unit in an operating system is the process, an instance of even a process that round-robins multiple requests can at most run only one request, no matter how many CPUs are available to service requests. Hence, multiple processes must be executed so that multiple CPUs can be used, with all the aforementioned drawbacks associated with multiple processes.

Microsoft SQL Server Process/Thread Model

A *thread* (more formally called a *thread of execution* and sometimes referred to as a *lightweight process*) is the executable unit on the Windows NT operating system. Threads, not processes, are scheduled for execution by Windows NT.

Rather than move a single thread among all user tasks, SQL Server employs a *pool* of threads. On a single CPU machine, a process using multiple threads is more efficient because even if one thread is not currently runnable (for example, it is waiting for an I/O to complete), another thread may well be runnable and will be executed. In a symmetric multiprocessor system, a process that has multiple threads (such as SQL Server) can use all the processors.

Windows NT provides *symmetric multiprocessor support* that allows execution of threads in parallel on multiple CPUs. Thus, SQL Server's process/thread model allows multiple SQL Server user connections to execute in parallel on multiprocessor hardware. In addition, certain discrete tasks, such as scanning data, can use multiple threads. For some tasks, a request by one user will run simultaneously across multiple CPUs. But more typically, a single user's task will run on one available CPU, another user's task will run on some other CPU, and so forth. (A more complete description of the Windows NT process and thread model is beyond the scope of this book. For more information, I suggest you read *Inside Windows NT* by Helen Custer [Microsoft Press, 1995].)

Because SQL Server uses native Windows NT threads, it automatically scales well to multiprocessor hardware with no special configuration or programming required. In addition, a relatively small amount of memory (about 55 KB) is required for each user connection. A large number of simultaneous users can be connected without consuming a lot of memory on the server.

Each user thread is maintained separately, so if one thread causes an access violation, only that thread will be affected; other threads will continue to operate unaffected. SQL Server's process/thread model greatly exceeds the reliability of typical UNIX-based database servers.

Multiuser Performance

The efficiency of the SQL Server threading model is borne out by its multiuser performance. SQL Server is able to efficiently handle hundreds, even thousands, of simultaneous active users. Built-in thread pooling allows workloads of this magnitude to be performed without the need for an external Transaction (TP) Monitor, which adds cost and complexity to a system.

NOTE There can, of course, never be a simple answer to questions such as "How many users can SQL Server handle?" or "How big a database can it handle?" The answers to these questions depend on the application and its design, required response times and throughput, and the hardware on which the system is running.

A majority of the systems that just a couple of years ago had required a mainframe or large minicomputer-based solution can now be efficiently built, deployed, and managed with SQL Server. Such industry-standard benchmarks as TPC-C can be illuminating. Today's SQL Server can perform workloads that surpass those submitted by the largest mainframe systems of a few years ago. As computer resources continue to grow, SQL Server will extend its reach into systems that traditionally would have required a mainframe solution.

Security

SQL Server provides numerous levels of security. At the outermost layer, SQL Server logon security is integrated directly with Windows NT security. With this *integrated security* in place, SQL Server can take advantage of the security features of Windows NT, such as password encryption, password aging, and maximum length restrictions on passwords.

Without integrated security, the administrator creates user accounts in the network/operating system and in SQL Server. A user logs on to the network and then must log on again to SQL Server. With integrated security, SQL Security Manager automatically copies Windows NT user accounts to SQL Server, providing an easy one-step process to implement integrated security. The user accounts created in Windows NT are automatically used to log a user on to SQL Server.

Integrated security relies on *trusted connections,* which makes use of the impersonation feature of Windows NT. Through impersonation, SQL Server can take on the security context of the Windows NT user account initiating the connection and test whether the Security Identifier (SID) has a valid privilege level. Windows NT impersonation and trusted connections are available with both the Named Pipes and Multi-Protocol network interfaces (Net-Libraries). Integrated security can be used with all the most popular network protocols, including TCP/IP, IPX/SPX, and NetBEUI.

For installations with a mix of named pipes and other clients (such as IPX/SPX or TPC/IC sockets), SQL Server can be installed in a *mixed security* model: named pipe clients will use integrated security and other clients will use standard SQL Server logon security. In addition, an application can request a trusted connection even if SQL Server has not been configured for integrated or mixed security. (You can also choose to disallow trusted connections.)

The Multi-Protocol Net-Library also allows all communications and data between the client application and the server to be optionally encrypted, which prevents even someone using a hardware "sniffer" from eavesdropping on the data. This is accomplished by using the Windows NT remote procedure call (RPC) services to encrypt the network traffic. This encryption uses a 40-bit key for versions of Windows NT Server sold outside of the United States and has an "RC4" designation by the U.S. National Bureau of Standards. The U.S. version of Windows NT Server uses a 128-bit key for much stronger security; however, it is not exportable due to U.S. government restrictions. Since SQL Server uses only the underlying services, the key length is transparent.

Monitoring and Managing Security

SQL Server makes it easy to monitor logon successes and failures. Administrators can simply check the appropriate box in the SQL Server Setup program. When logon monitoring is enabled in this way, each time a user successfully or

unsuccessfully attempts to log on to SQL Server, a message is written to the Windows NT event log indicating the time, date, and user who tried to log on.

SQL Server has a number of facilities for managing data security. Access privileges (select, insert, update, and delete) can be granted and revoked to users or groups of users on objects such as tables and views. Execute privilege can be granted on local and extended stored procedures. For example, to prevent a user from directly updating a specific table, you can write a stored procedure that updates the table and then cascades those updates to other tables as necessary. You can grant the user access to execute the stored procedure, thereby ensuring that all updates will take place through the stored procedure, eliminating the possibility of integrity problems arising from ad hoc updates to the base table.

High Availability

In many mission-critical environments, it is imperative that the application be available at all times—24 hours a day, seven days a week. SQL Server helps availability by providing online backup, online maintenance, automatic recovery, disk mirroring, and the ability to configure a fallback (or failover) server.

SQL Server's dynamic online backup allows databases to be backed up while users are actively querying and updating in the database. The SQL Executive service provides a built-in scheduling engine that enables backups to be scheduled to occur automatically, without involving the administrator. Other maintenance tasks, such as diagnostics, design changes (for example, adding a column to a table), and integrity changes can be accomplished without having to shut down SQL Server or restrict user access.

Only a few system-wide configuration changes, such as changing the amount of memory configured for use, require that SQL Server be restarted. Although these activities do not commonly occur in a well-planned and well-deployed production system, they can typically be completed with less than a minute of system downtime if and when they are necessary.

In the event of a system failure, such as a power outage, SQL Server ensures rapid database recovery when services are restored. By using the transaction logs associated with each database, SQL Server quickly recovers each database upon startup, rolling back transactions that had not yet completed and rolling forward transactions that had committed but were not yet written to disk. In addition, the SQL Executive service can be set to continually monitor the state of SQL Server. If an error occurs that causes SQL Server to stop unexpectedly, SQL Executive will detect this and can automatically restart SQL Server with minimal interruption.

In cooperation with shared-disk cluster hardware (such as the Compaq Online Recovery Server), SQL Server 6.5 provides fallback (also known as failover) capability. You can designate a SQL Server *standby server* to back up the *primary*

server. Should the primary server fail, the hardware support will signal the standby machine. SQL Server on the standby server will then mount and recover the databases of the primary server and take over its workload. The standby server does not need to be inactive while the primary server is working—it can also be functioning as a primary server. In fact, two primary servers can be configured as standbys for each other as well as for other servers.

Support is built into DB-Library and the SQL Server ODBC driver so that the primary-standby relationship is silently made known to the application when it initially connects. Should the primary server unexpectedly become unavailable, the application can automatically reconnect to the "hot backup" standby server and resume work with no user intervention and minimal disruption.

Distributed Data Processing

SQL Server provides features such as transactional remote stored procedure calls and two-phase commit for easily managing and using data in distributed environments. Although even Microsoft SQL Server version 1.1 supported a two-phase commit protocol, the new Microsoft Distributed Transaction Coordinator (MS DTC) that ships as part of SQL Server 6.5 has made those capabilities obsolete.

MS DTC was designed to be the "vote collector" and coordinator of transactions, and it allows many different types of systems to participate, laying the foundation for ACID transactions among heterogeneous systems. A system participating in a transaction coordinated by MS DTC manages its own work and is called a *resource manager*. This resource manager system communicates with MS DTC, which coordinates all the resource managers participating in the transaction to implement the two-phase commit protocol. Distributed transactions honoring the ACID properties are supported as a whole: the entire distributed transaction at all sites either commits or aborts.

SQL Server 6.5 is the first resource manager coordinated by MS DTC. However, by providing an X/Open DTP XA–compliant interface, MS DTC, and hence SQL Server, also interoperate with several transaction processing monitors, including Encina, Topend, and Tuxedo. MS DTC implements the OLE (Object Linking and Embedding) Transaction interfaces. Because all OLE Transaction interfaces are public, any database system can become an OLE Transaction resource manager and consequently participate in distributed transactions with SQL Server. In the future, Microsoft and other software companies will add other transactional resource managers, such as transactional behaviors, to the file system and workflow management systems. Transactional resource managers would, for example, allow a transaction with ACID properties to span Microsoft SQL Server, the Informix RDBMS, and the NTFS file system. (I can even imagine smart hardware devices participating in such transactions—such as a check-writing transaction in which

the transaction is complete only when and if the check is printed correctly.) So far, both Informix and Sybase have also pledged support for MS DTC.

In the first phase of the two-phase commit protocol, all participating resource managers (that is, those that have "enlisted" in the transaction) *prepare to commit*. This means that they have acquired all the locks and resources they need to complete the transaction. MS DTC then acts as a vote collector. If it gets confirmation that all participants are prepared to commit, it signals "go ahead and commit."

The actual *COMMIT* is the second phase of the protocol. If one or more participants notify the system that it cannot successfully prepare the transaction, MS DTC will automatically send a message to all participants indicating that they must abort the transaction. (In this case, an *abort,* rather than a commit, is the second phase of the protocol.) If one or more participants do not report back to MS DTC in phase one, the resource managers that have indicated that they are prepared to commit (but have not yet committed, since they have not received the instruction to do so yet), are said to be *in doubt*. Resource managers that have transactions in doubt will indefinitely hold the locks and resources necessary to ultimately commit or roll back the transaction, preserving the ACID properties. (SQL Server provides a way to force in doubt transactions to abort.)

Another important distributed capability is the ability for a SQL Server to issue remote procedure calls (RPCs) to other SQL Servers. *Remote procedure calls* are stored procedures that can be invoked from a remote server, allowing server-to-server communication. This communication can be accomplished transparently to the client application, since the client can execute a procedure on one server, and that procedure can then invoke a procedure located on a different server. Using RPCs can easily extend the capacity of an application without the added cost of reengineering the client application. And these RPCs can be coordinated by the MS DTC service to ensure that the transactions maintain their ACID properties.

Data Replication

Replication allows you to automatically distribute copies of data from one server to one or more destination servers at one or more remote locations. A key design point of the replication capabilities of Microsoft SQL Server is data integrity. The data at subscribing sites may be slightly out of date, but it will accurately reflect the state of the master copy of the data at some recent point in time. Because of this emphasis on the correctness of data, the replication metaphor used is *publish and subscribe*. For each piece of data participating in replication in the entire system, one site is the designated owner of that data, and that site *publishes* it to the other sites, which *subscribe* to that data. (A given site can publish some data and subscribe to other data.)

Distributed transactions using the two-phase commit protocol guarantee ACID properties, but replication does not. Replication is not strongly consistent (the *C* in ACID). Instead, replication provides *loosely consistent* data. Recall that with the two-phase commit protocol, a transaction is an all-or-nothing proposition and the data is assured to be *strongly consistent*. But inherent in the two-phase commit algorithm is the fact that a failure at any one site makes the entire transaction fail or can keep the transaction in doubt for long periods of time, during which all participants need to hold locks, crippling concurrency.

At first look, you may think a system should require that updates be made at all sites in *real time*. In fact, when the costs of two-phase commit are realized (chiefly, the vulnerability that can result due to a failure at just one node), the most pragmatic solution might be to make changes in *real enough time*.

For example, suppose you run a car rental agency, with 500 rental counters worldwide, and you maintain a customer profile table containing 500,000 renters who belong to your Gold Card program. You want to store this customer profile locally at all 500 rental counters so that even if a communication failure occurs, the profile will be available wherever a Gold Card member might walk up to do business. Although all sites should have up-to-date records of all customers, it would be disastrous to insist that an update of the Gold Card profile must occur as part of a two-phase transaction for all 500 sites or not at all. With this scenario, because of the realities of worldwide communications, or because at one site a storm might have knocked out the power, it is likely that you would seldom be able to perform a simple update to the customer profile.

Replication is a much better solution in such a case. The master customer profile table would be maintained at your corporate headquarters. Replication publishes this data, and the rental counters then subscribe to this information. When customer data is changed, or when a customer is added or removed, these changes (and *only* the changes) are propagated to all the subscribing sites. In a well-connected network, the time delay might be just a few seconds. If a particular site is unavailable, no other sites are affected—they still get their changes. When the unavailable site is back online, the changes are automatically propagated and the subscriber is brought up to date. At any time, a given rental counter may not have exactly the same information as the corporate site—it might be slightly out of date. The data at the rental counter is consistent with the state of the data at the corporate headquarters at some earlier time; it is not necessarily currently consistent with the corporate site data. This is what is meant by *loosely consistent,* as opposed to the *strongly consistent* model of two-phase commit in which all sites (or none) immediately reflect the change.

Although a time delay can occur in loosely consistent systems, maintaining transactional consistency is one of the chief design points of SQL Server replication. If multiple updates occur as a single atomic transaction to data being replicated,

the entire transaction will also be replicated. At the subscribing site, the transaction will either entirely commit or it will again be replicated until it commits.

With SQL Server, data can be replicated continuously or at specified intervals. It can be replicated in its entirety or as filtered subsets (known as *horizontal* and *vertical partitions*). In addition to replication to other SQL Servers, version 6.5 can replicate to Microsoft Access databases, ORACLE databases, or other ODBC subscribers (so long as they provide an appropriate ODBC driver).

Unlike SQL Server, some products on the market promote replication as an "Update Anywhere-Anytime-Anyway" model. However, this model has inherently unstable behavior if many nodes participate and update activity is moderate to heavy.[1] Updates made at multiple sites will conflict with one another and must be reconciled. A small system with few changes might appear to use the "Update Anywhere-Anytime-Anyway" model effectively, but a large system with a tenfold increase in nodes and traffic gives a thousandfold increase in deadlocks or reconciliations. Before long, even accurate reconciliation will be impossible because updates will have been made based on data that has not been accurately reconciled. The system will degrade into an inconsistent state, with no clear way to fix it or even to know what is the "correct" state.

Systems Management

The difficulty of systems management is probably the single biggest obstacle that has inhibited mass deployment of client/server solutions. Far from being a "downsized" version of the mainframe, today's distributed client/server system may be deployed on dozens or even hundreds of distributed servers, all of which must be controlled to the same exacting standards as mainframe production software systems. The issues here reside both inside and outside of the database environment. SQL Server provides a comprehensive architecture and tools for managing the database and related activities.

SQL Enterprise Manager

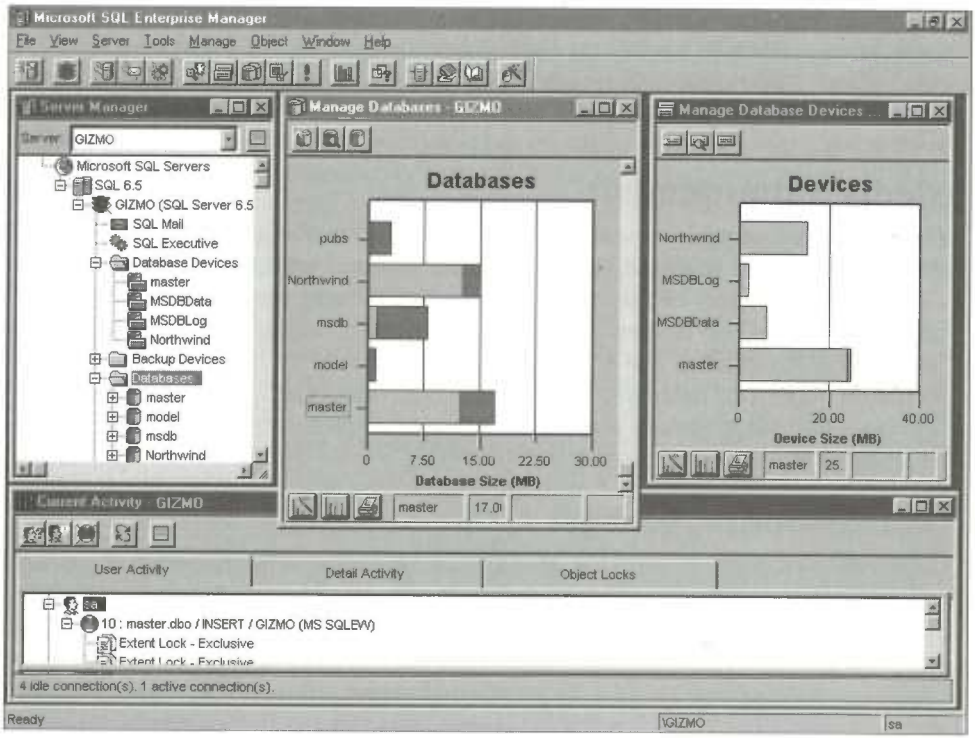
SQL Server's SQL Enterprise Manager is a major advancement in making client/server deployments manageable. Easy to use, SQL Enterprise Manager supports centralized management of all aspects of multiple SQL Servers, including managing security, events, alerts, scheduling, backup, server configuration, tuning, and replication. SQL Enterprise Manager allows SQL Server database schemas and objects such as tables, views, and triggers to be created, modified,

1. Gray, Helland, O'Neil, and Shasha, "The Dangers of Replication and a Solution," SIGMOD (1996). SIGMOD (Special Interest Group Management of Data) is a yearly database-oriented conference for developers. For more information, see http://bunny.cs.uiuc.edu.

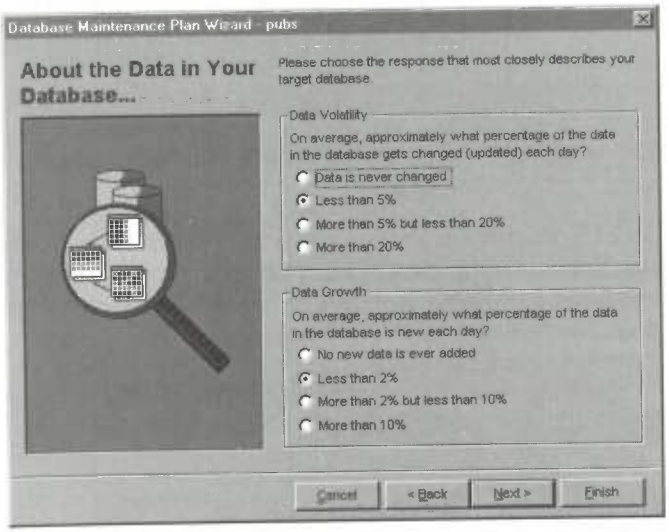
and copied. Because groups of servers can be associated, SQL Enterprise Manager can manage hundreds of servers simultaneously.

Although it can run on the same computer as the SQL Server engine, SQL Enterprise Manager offers the same management capabilities while running on any Windows NT workstation or Windows NT server in the environment. SQL Enterprise Manager also runs on Windows 95, although a few capabilities are not available in this environment (most notably the ability to use Service Control Manager, a feature of Windows NT, to remotely start and stop SQL Server). In addition, the efficient client/server architecture of SQL Server makes it practical to use the remote access (dial-up networking) capabilities of Windows NT and Windows 95 for administration and management.

SQL Enterprise Manager provides an easy-to-use interface, as shown in the illustration below. You can perform even complex tasks with just a few mouse clicks.



SQL Enterprise Manager relieves you from having to know the specific steps and syntax to complete a job. You can use the Database Maintenance Plan Wizard, shown below, to set up and schedule key maintenance tasks to help keep your system running properly.



Distributed Management Objects

In the Microsoft Windows 95 and Microsoft Windows NT operating systems, *Microsoft SQL Server Distributed Management Objects (SQL-DMO)* provides 32-bit Automation (formerly known as OLE Automation). These objects, properties, methods, and collections are used to write scripts and programs that can administer multiple SQL Servers distributed across a network. SQL Enterprise Manager is built entirely with SQL-DMO. You can customize your own specific management needs using SQL-DMO, or you can integrate management of SQL Server into other tools you use or provide.

All SQL Server functions are exposed in the form of objects, methods, and properties. The SQL-DMO model simplifies the management "surface" of SQL Server by organizing management functions in terms of the SQL Server object model. The primary object is SQLServer, which contains a collection of Database objects. The Database object contains a collection of Table, View, and StoredProcedure objects. Objects contain properties (SQLServer.Name = "MARKETING_SVR") and methods (SQLServer.Start or SQLServer.Shutdown).

Here are some examples of SQL-DMO objects and methods:

Object.Method	*Action*
SQLServer.Shutdown	Stops a SQL Server
SQLServer.Start	Starts a SQL Server
Database.Dump	Performs a database dump
Index.UpdateStatistics	Updates optimizer information for indexes
Database.Table.Add	Adds a table to a database

The SQL-DMO object model is comprehensive, consisting of more than 70 distinct objects and more than 1500 COM interfaces. The organization of these objects greatly reduces the task of learning and fully using SQL Server management components, as shown in Figure 2-2.

Any 32-bit Automation controlling application can harness the power and services of SQL-DMO. Probably the most common such Automation controller is Microsoft Visual Basic.

Automation and Visual Basic Scripting

The power of using an ActiveX interface (Automation) for SQL Server management becomes clear when you consider the potential of using a robust language such as Visual Basic as a scripting environment for administrative tasks. The following sample code lists the name and space available on all databases on a server. This code is simple, compact, easy to write and read, and yet very powerful. (Traditionally, programming such a task would have required several pages of much more complex C code.)

```
Dim MyServer as New SQLServer     'Declare the SQL Server Object
MyServer.Name = "MARKETING_SVR"
MyServer.Login = "sa"
MyServer.Connect          ' Connect to the SQL Server
' list the name, space available for all databases
For each MyDB in MyServer.Databases
    Print MyDB.Name, MyDB.SpaceAvailable
Next MyDB
MyServer.Disconnect          ' Disconnect
```

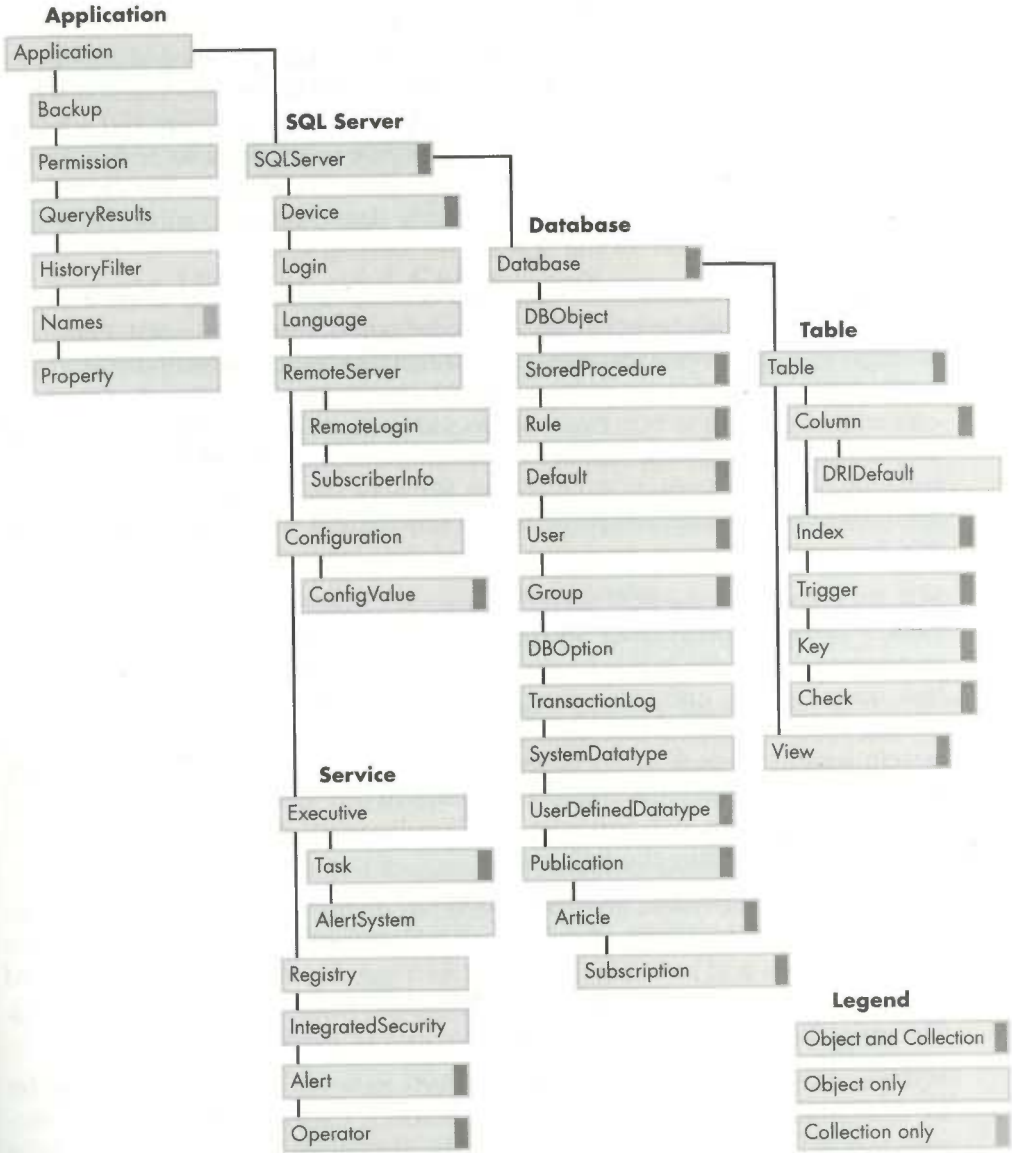


Figure 2-2. *The SQL-DMO model makes using the objects easy.*

SQL Executive

SQL Executive is an active, intelligent agent that plays an integral role in the management of the SQL Server environment. It provides a full-function scheduling engine designed to support regular tasks for the management and administration of SQL Server, and it allows you to schedule your own tasks and programs. SQL Executive plays a fundamental role in replication, acting as the mechanism that runs the distribution tasks that propagate data changes to subscribing sites. It is also the foundation for the SQL Server alerting system.

SQL Executive is a Windows NT–based service that can be started when Windows NT starts. It can also be controlled and configured from within SQL Enterprise Manager or SQL Service Manager. SQL Executive is entirely driven by entries in a SQL Server table that act as its control block. Clients never directly communicate with or connect to SQL Executive to add or modify scheduled tasks. Instead, they simply make the appropriate entries in the SQL Server table (although this typically occurs through SQL Enterprise Manager via a simple dialog box that's similar to a typical calendar program). At startup, SQL Executive connects to the SQL Server that contains its task table and then loads the list of tasks.

SQL Executive, like the SQL Server engine, is a single multithreaded process. It runs in its own process space and manages the creation of Windows NT threads to execute scheduled tasks. Its discrete managed subsystems (for replication, task management, and event alerting) are responsible for all aspects of processing specific tasks. When tasks are completed (successfully or not), the subsystem returns a result status (with optional messages) to SQL Executive. SQL Executive then records the completion status in the Windows NT event log and task history table in SQL Server and optionally sends e-mail to the designated administrator reporting the task status.

The event/alert subsystem gives SQL Server its ability to support proactive management. The primary role of the event/alert subsystem is to respond to events by raising alerts and invoking responses. As triggering activities (or user-defined activities) occur in the system, an event is posted to the Windows NT event log. The event log then notifies SQL Executive that an event has occurred.

SQL Executive determines whether any alerts have been defined for this event by examining the event's error number, severity, database of origin, and message text. If an alert has been defined (in the alert table), the administrator(s) can be alerted via e-mail, pager, or by raising a Simple Network Management Protocol (SNMP) trap (discussed later in this chapter). Or a SQL Executive on-demand task can be invoked and can take corrective action. (For example, SQL Executive might automatically expand a database that is almost full.)

If no alerts are defined locally, the event can be forwarded to another server for processing. This feature allows groups of servers to be monitored centrally so that alerts and administrators can be defined once and then applied to multiple servers. Beyond the database environment, Microsoft Systems Management Server (SMS)—a BackOffice component—is available to provide key services to manage the overall software configuration of all the desktops and servers in the environment.

SQL Server Utilities and Extensions

SQL Server also includes utilities and extensions that provide increased functionality, such as Internet enabling, monitoring capability, easy setup, and easy data importing.

SQL Server Web Assistant and Internet Enabling

SQL Server provides dynamic ways in which to work with the Internet: SQL Server Web Assistant and interoperability with Microsoft Internet Information Server (IIS). Although both the SQL Server Web Assistant and IIS enable SQL Server data to be used with Web pages, they satisfy different needs.

SQL Server Web Assistant generates HTML files from the result sets of SQL Server queries, making it simple to publish SQL Server data on the Internet. Let's say, for example, that a parts supplier keeps its inventory list in SQL Server. The supplier could publish its current parts inventory as a Web page (an HTML file) using SQL Server Web Assistant. SQL Server Web Assistant allows an ad hoc query or stored procedure to be submitted, provides some simple formatting capabilities, allows for the inclusion of links to other Web pages, and allows a template to be used for more advanced formatting. The output of the query is written as an HTML 2.0 table, and a Web page is created. The process to create or update the Web page can be automated by SQL Server Web Assistant to occur at a regular interval or whenever the data changes (via a trigger).

SQL Server Web Assistant is distinct from but complementary to the IIS in Microsoft BackOffice. With SQL Server Web Assistant, users browsing the Web page work separately from SQL Server, because the data on the Web page has been extracted. SQL Server Web Assistant does not use or require IIS, and a SQL Server Web Assistant page can be viewed using any Internet browser.

IIS uses SQL Server's high performance native ODBC interface to allow SQL queries to be fired from a Web page when a user accesses a particular region on the page. The results are then dynamically retrieved and combined with the HTML file for up-to-date viewing. In addition to SQL Server Web Assistant and the dynamic query capabilities enabled with IIS, SQL Server is Internet-enabled

in several other important ways. By minimizing network traffic and handshaking, SQL Server is inherently designed for efficient client/server computing. Extremely rich requests can be packaged via stored procedures or Transact-SQL batches for resolution entirely at the server with only the results sent back to the initiating client application. This capability has been a hallmark of SQL Server's client/server architecture from the outset, but nowhere is it more important than on the Internet, where network speed and bandwidth are often quite limited. In addition, SQL Server's networking architecture allows for ease of use and security on the Internet, including network name resolution. For example, Internet users can connect via a friendly name such as "sql.microsoft.com" instead of via an arcane IP address such as 200.154.54.678:1433. Secure encryption of data over the Internet is also possible.

SQL Trace

SQL Trace is a Win32-based graphical utility that allows database administrators and application developers to monitor and record database activity. SQL Trace can display all server activity in real time, or it can create filters that focus on the actions of particular users, applications, or types of commands. SQL Trace can display any SQL statement or stored procedure sent to any SQL Server (assuming your security privileges allow it) as well as the output or response sent back to the initiating client. The capabilities of SQL Trace provide an important tool for tuning and debugging applications and for auditing and profiling the use of the SQL Server.

SQL Service Manager

SQL Service Manager, shown below, manages the SQL Server, SQL Executive, and MS DTC services. It provides a simple way to start, stop, or check the state of any of these services. Many applications have "borrowed" its original intuitive traffic-light graphic as a way to provide a simple visual representation of a process's state.

Windows NT Performance Monitor Integration

SQL Server provides an extension DLL (SQLCTR60.DLL) that integrates with the Windows NT Performance Monitor and graphically displays important performance statistics, such as memory usage, number of users, transactions per second, and CPU use as well as many others (there are more than 75 such counters). Integrating with the Windows NT Performance Monitor is advantageous because it allows you to use a single tool to measure all aspects of a system's performance. If SQL Server simply provided its own performance-monitoring tool, it would still be necessary to check the performance of the operating system and network. Integration with the system performance monitor provides "one-stop shopping." A Performance Monitor graph is shown in the illustration below.

Using Performance Monitor, you can set an "alert" on any statistic that is being monitored; when a predefined threshold is reached, Windows NT will automatically execute a predefined command. For example, you can set an alert that is generated when a SQL Server database's transaction log becomes 90 percent full. The alert will execute a batch program to back up the log and purge it, freeing up space for new transactions.

SQL Security Manager

SQL Security Manager simplifies the management of user logons with SQL Server. It provides an interface to integrate Windows NT user accounts directly into SQL Server without having to redefine each user logon. SQL Server uses the Windows NT user accounts to validate database users and administrators.

SQL Client Configuration Utility

The SQL Client Configuration Utility is used with applications written with the DB-Library API. You use it to set up specific networking options for the application. In most cases, network name resolution is now automatic and the SQL Client Configuration Utility is usually not required.

SQL Server Setup

SQL Server's graphical Setup program allows SQL Server installation to be performed with a degree of ease and speed unprecedented for a full-feature DBMS. If the defaults are chosen, SQL Server can be installed in 5 to 10 minutes, depending on the speed of the computer. A custom installation typically takes well under 30 minutes. Traditional DBMS products usually require several days for installation and often require that you enroll in training classes before installing the product. SQL Server Setup can even install the product on a remote computer, a particularly useful feature if you manage a large number of servers. Setup initialization is easily specified so that it can be fully automated for numerous installations, or you can encapsulate the installation of SQL Server in the rest of an application's installation process.

After you've installed SQL Server, you use the Setup program to configure networking choices and server environment options (such as the security mode desired). The program also provides a simple uninstall process.

ISQL/w & ISQL

Having a simple interactive window in which to submit basic SQL commands and get results is essential to a database developer—as a hammer is to a carpenter. Even though other, more sophisticated, power tools are useful, there is always need for the basics. SQL Server provides the basics in two styles: ISQL/w and ISQL.

For interactive use, ISQL/w (ISQLW.EXE) provides a clean and simple Windows-based interface. It provides a graphical representation of SHOWPLAN, the steps chosen by the optimizer for query execution. ISQL/w allows for multiple windows so that simultaneous database connections (to one or more servers) can exist and be separately sized, tiled, or minimized.

ISQL (ISQL.EXE) is a character-based command-line utility. Every parameter, including the SQL statement or the name of the file containing the statement, can be passed to this character-based utility. Upon exit, it can return status values to Windows NT that can be checked within a command file (.CMD or .BAT). Consequently, programs commonly launch scripts of SQL commands by spawning the character-based ISQL utility and passing the appropriate parameters and filenames. SQL Server Setup itself spawns ISQL.EXE numerous times with scripts that install various database objects and permissions.

Bulk Copy Utility

SQL Server provides a character-based utility called **bcp**, or "bulk copy" (BCP.EXE), for flexible importing and exporting of SQL Server data. Similar to ISQL, **bcp** allows all parameters to be passed to it and is often called from command files. A special set of functions exists in DB-Library that lets you easily create a custom loader or unloader for your application; the **bcp** utility is a generalized wrapper application that calls these functions. SQL Enterprise Manager and SQL-DMO also provide facilities to simplify the transfer of data into or out of SQL Server, but their emphasis is on data migration between SQL Servers. The **bcp** utility provides an important capability by allowing you to specify the exact data format to be read or written, thus giving you the ability to exchange data with other data sources.

SNMP Integration

Support for Simple Network Management Protocol (SNMP, a standard protocol within TCP/IP environments) is provided via the SQL Server Management Information Block (MIB, another standard of the SNMP and TCP/IP environment). A group of database vendors, including Microsoft, cooperated in defining a standard MIB that would report certain status data about a database environment for monitoring purposes.

NOTE This group was a subcommittee of the IETF, the Internet Engineering Task Force, and the draft specification is known as "IETF SNMP RDBMS-MIB (RFC 1697)." The SQL Server MIB is generally based on this proposal but provides additional data beyond that called for in the specification.

For example, status information (such as whether SQL Server is currently running, when it was last started, and how many users are connected) is reported to SNMP via this MIB. A variety of SNMP management and monitoring tools exist and can access this data. If, for example, you use Hewlett-Packard's OpenView or Computer Associate's CA-Unicenter in managing your network, the SQL Server MIB enables those tools to also monitor multiple statistics regarding SQL Server.

SQL Server also supports the ability to raise *SNMP traps*. An SNMP trap sends notification to the *SNMP agent* that some type of change in status or condition has occurred. The SNMP agent can then forward that notification to prespecified workstations that are running SNMP monitoring application(s). This means that it is simple to configure alerts within SQL Enterprise Manager that will notify your network management application, such as OpenView or CA-Unicenter, of some situation that warrants attention (that the database should be expanded in size, for example).

SQL Server Books Online

In addition to being available in printed form, all SQL Server documentation is available online. A powerful viewer makes it simple to find and search for topics within seconds. (The viewer is the same one used by MSDN, the Microsoft Developer Network Library, which provides technical information to developers by subscription.)

Even if you favor printed books, you'll appreciate the speed and convenience of the search capabilities of SQL Server Books Online. In addition, because of the lead time required in the production of the printed manuals that ship with SQL Server, SQL Server Books Online is more complete and accurate than those manuals. Because this book cannot hope to and does not try to replace the complete documentation set, the CD-ROM included with this book also contains the complete SQL Server Books Online documentation.

Development Interfaces

SQL Server provides several development interfaces, supporting client *and* server application development. These interfaces—the DB-Library, ODBC, the Embedded SQL precompiler, and Microsoft Open Data Services—are described below.

DB-Library

DB-Library is a SQL Server–specific API that provides all the necessary macros and functions for an application to open connections, format queries, send them to the server, and process the results. It also includes the special purpose bulk copy interface used by the **bcp** utility. You can write custom DB-Library applications using either C/C++ or Microsoft Visual Basic (or any programming language that is capable of calling a C function).

DB-Library is the original programming interface to SQL Server. Libraries are provided for MS-DOS, Windows 3.1, Windows 95, and Windows NT. (The Windows NT library for Intel computers is the same library used for Windows 95, so you can write a single application that targets both environments.) Developers are granted licensed rights to redistribute the DB-Library runtimes royalty-free.

ODBC

ODBC (Open Database Connectivity) is an API for database access that is both a formal and de facto industry standard. Besides being the most popular database interface used by applications today, ODBC has gained status as the formal call-level interface standard by ANSI and ISO. Microsoft SQL Server provides a high-performance, *native* ODBC interface for all Windows-based programming environments, and like DB-Library, it can be distributed royalty-free with any application. The SQL Server ODBC driver implements every function in the

ODBC 2.0 specification. In "ODBC-speak," this makes it fully "Level 2" (the highest level) conformant.

ODBC drivers for other operating systems (Macintosh System 7, many flavors of UNIX, and OS/2) are available from Visigenic. (You can check out Visigenic's Web site at http://www.visigenic.com.) Microsoft licenses the source code for the SQL Server ODBC driver to Visigenic so that high-performance interfaces are available for those operating systems as well.

Most new application development is probably best targeted toward ODBC, rather than DB-Library. If you are already using DB-Library or have a lot invested in DB-Library in terms of knowledge and applications, then don't think you must abandon it. DB-Library will remain supported indefinitely, and that's why it was significantly enhanced for SQL Server version 6.5. In the future, though, you should expect fewer enhancements for DB-Library. It will likely continue to be maintained simply for backward compatibility. (You may also be steered to DB-Library if your application must include an MS-DOS version or if you will be directly using the **bcp** library, for which there is no current ODBC equivalent.)

If you are starting new development, however, most applications would be better served using ODBC. Having had the benefit of learning from DB-Library and going through a long design and specification phase, I've found that ODBC is a simpler, more elegant API than DB-Library. Despite some myths to the contrary, ODBC is as fast or faster than DB-Library. We call the ODBC driver for SQL Server a *native interface* to make it clear that it is not mapped onto DB-Library and does not incur more overhead than DB-Library. It directly reads and writes the SQL Server data stream protocol, Tabular Data Stream (TDS), just as DB-Library does. Perhaps the best proof of Microsoft's confidence in its performance is the fact that ODBC is used in SQL Server's published performance benchmarks. Since the name of the game in benchmarking is to eke out every last ounce of performance, if using the ODBC driver caused even a tiny slowdown, you can bet we'd use DB-Library instead.

ESQL for C

SQL Server provides an Embedded SQL precompiler (ESQL for C) that allows developers to write SQL Server applications by "embedding" the SQL queries directly in their C source code. Many minicomputer and mainframe developers are already accustomed to this style of programming, and ESQL might be a natural choice for that reason. In addition, Microsoft has licensed some of the Embedded SQL run-time environment to Microfocus, the leading provider of Cobol compilers and tools. Microfocus offers an embedded SQL interface for SQL Server directly in its Cobol development environment.

Open Data Services

SQL Server offers an open API for developing server-based gateway and connectivity applications that work in conjunction with SQL Server. Microsoft Open Data Services (ODS) is an event-driven API that provides a programmable gateway platform for server applications that can access any data source. ODS can be used to develop custom database gateways, data-driven event alerters, external program triggers, request auditing, extended stored procedure DLLs, and more. ODS is actually a core part of the SQL Server architecture and benefits from the high-performance architecture. It provides all the network, connection, and thread management that SQL Server uses.

ODS-based applications can function as stand-alone gateways, or as data-access servers, supporting connections from the same client platforms as SQL Server. They can also integrate with SQL Server directly through remote stored procedure calls.

Microsoft TransAccess uses ODS to provide data connectivity from SQL Server to the MVS CICS environment, including access to DB/2, VSAM, and IMS data. In addition, ODS is used by many other software vendors to provide SQL Server–compatible gateways to popular host computing platforms, including IBM DB2, IBM SQL/DS, CICS, IBM AS/400, and others.

SUMMARY

The Microsoft SQL Server component and product family, including the SQL Server RDBMS, visual systems management tools, distributed systems components, open client/server interfaces, and visual development tools, provides a complete and robust platform for developing and deploying large-scale applications.

The remainder of this book concentrates on the capabilities and uses of the SQL Server engine, which is the foundation of the product.

Naming Conventions

Many organizations and multiuser development projects adopt standard naming conventions, which are a good thing, in general. For example, assigning a standard moniker of *cust_id* to represent a customer number in every table makes it obvious that all the tables have data in common. If, instead, several monikers were used in the tables to represent a customer number, such as *cust_id, cust_num, customer_number,* and *customer_#,* it would not be so obvious that these monikers represented common data. One convention I see occasionally and recommend *against* using is Hungarian-style notation for column names. (Hungarian notation is a widely used practice in C programming, whereby variable names include information about their datatypes. Its name is attributed to its use by legendary Microsoft programmer Charles Simonyi, who is of Hungarian ancestry.) Hungarian-style notation uses names such as *sint_nn_custnum* to represent that the *custnum* column is a small integer (*smallint* of 2 bytes) and is NOT NULL (does not allow nulls). Although this practice makes good sense in C programming, it defeats the datatype independence that SQL Server provides.

Suppose it is discovered, for example, that the *custnum* column requires a 4-byte integer (*int*) instead of a 2-byte small integer. It is relatively simple to re-create the table with the column as an *int* instead of a *smallint*. In SQL Server, stored procedures will deal with the different datatype automatically. Applications using DB-Library or ODBC that bind the retrieved column to a character or integer datatype will be unaffected. The applications would need to change if they bound the column to a small integer variable, as the variable's type would need to be larger. For this reason, it is best to try not to be overly conservative with variable datatypes, especially in your client applications. You should be most concerned with the type on the server side; the type in the application can be larger and will automatically accommodate smaller values. By overloading the column name with datatype information, which is readily available from the system catalogs, the insulation from the underlying datatype is compromised. (You could, of course, change the datatype from a *smallint* to an *int,* but then the Hungarian-style name would no longer accurately reflect the column definition. Changing the column name would then result in the need to change application code or stored procedures or both.)

Datatypes

SQL Server provides a large number of datatypes, as shown in Table 6-1 on pages 202–03. Choosing the appropriate datatype is simply a matter of mapping the domain of values you need to store to the corresponding datatype. In choosing datatypes, you want to avoid wasting storage space while allowing enough space for a sufficient range of possible values over the life of your application.

Datatype synonyms

SQL Server syntactically accepts as datatypes both the words listed as synonyms and the base datatypes shown in Table 6-1, but it uses only the type listed as the datatype. For example, a column can be defined as *character(1), character,* or *char(1),* and SQL Server will accept all these as valid syntax. Internally, however, the expression is considered *char(1),* and subsequent querying of the SQL Server system catalogs for the datatype will show it as *char(1),* regardless of the syntax that was used when it was created.

Nullable columns are variable-length

Before deciding to use an ostensibly fixed-length datatype such as *char* instead of a variable-length one such as *varchar,* it is important that you understand *nullability*: all datatypes, with the exception of *bit,* can be declared either NULL or NOT NULL (that is, they can allow or disallow a null entry). Internally, declaring a column to allow a null entry makes that column a variable-length column. For example, a column declared as *char(5) NULL* is internally identical to one declared *varchar(5) NULL.* In both cases, if a null value is entered, no storage is consumed. If only 3 bytes are entered, then only 3 bytes of storage are used, even for the fixed-length type.

NOTE There is an exception to this. The command *SET ANSI_PADDING ON* instructs SQL Server to physically store spaces in the remaining 2 bytes of the *char(5)* type, in which case 5 bytes of storage would be used. This setting conforms to the ANSI SQL-92 standard.

Variable-length vs. fixed-length datatypes

Deciding to use a variable-length or a fixed-length datatype is not always straightforward or obvious. As a general rule, variable-length datatypes are most appropriate when you expect significant variance in the size of the data for a column and the data in the column will not be frequently changed.

Using variable-length datatypes can yield important storage savings. Choosing them can sometimes result in performance loss (as I will explain in a moment) and at other times can result in improved performance. A row with variable-length columns (including supposed fixed-length columns that allow NULLs) requires special offset and adjust entries to be internally maintained. These entries keep track of the actual length of the column. Calculating and maintaining the offsets requires slightly more overhead than a pure fixed-length row, which needs no such offsets at all. This is a CPU task of a few addition and subtraction operations to maintain the offset value. However, the extra overhead of maintaining these offsets is generally inconsequential, and I have not seen a system in which this alone made a significant difference. A more significant performance difference might arise from the method by which updates are processed.

Type of Data	Base Datatype	Synonyms	Range/Domain	Storage Size
Integer	*int*	*integer*	Whole numbers from –2,147,483,648 to 2,147,483,647	4 bytes
	smallint		Whole numbers from –32,768 to 32,767	2 bytes
	tinyint		Whole numbers from 0 to 255	1 byte
Packed decimal (exact numeric)	*numeric (p,s)* *decimal (p,s)*	*dec*	Whole or fractional numbers from -10^{38} to 10^{38}	2–17 bytes, depending on specified precision, p, which can range to 38 digits. On average, 1 byte of storage is required per every 2 digits of precision.
Floating point (approx numeric)	*float* (15-digit precision)	*float(n)*, where *n* is between 8 and 15 Double precision	Approximations of numbers from $-1.79E^{308}$ to $1.79E^{308}$ Positive range: $2.23E^{-308}$ to $1.79E^{308}$ Negative range: $-2.23E^{-308}$ to $-1.79E^{308}$	8 bytes
	real (7-digit precision)	*float(n)*, where *n* is between 1 and 7	Approximations of numbers from $-3.40E^{38}$ to $3.40E^{38}$ Positive range: $1.18E^{-38}$ to $3.40E^{38}$ Negative range: $-1.18E^{-38}$ to $-3.40E^{38}$	4 bytes
Character (fixed length)	*char(n)*	*character (n)* *character* (character without a specific size is synonymous to a 1-character field, char(1))	Up to 255 characters, as designated by *n*, of the installed character set	1 byte per character *n* declared, even if partially unused.
Character (variable length)	*varchar(n)*	*character varying (n)*, *char varying (n)*	Up to 255 characters, as designated by *n*, of the installed character set	1 byte per character stored. Declared but unused characters do not consume storage.
Monetary	*money*		Numbers with accuracy to one ten-thousandth of a unit (four decimal places), typically used to store currency values. From –922,337,203,685,477.5808 to 922,337,203,685,477.5807	8 bytes
	smallmoney		Numbers with accuracy to one ten-thousandth of a unit (four decimal places), typically used to store currency values. From –214,748.3648 to 214,748.3647	4 bytes

Table 6-1. *SQL Server supplies many datatypes.*

Type of Data	Base Datatype	Synonyms	Range/Domain	Storage Size
Date and Time	*datetime*		Combined date and time representation. (SQL Server does not have separate DATE and TIME datatypes.) Date part: 01-JAN-1753 to 31-DEC-9999 Time part: Number of milliseconds since midnight of the given date	8 bytes
	smalldatetime		Combined date and time representation. Date part: 01-JAN-1900 to 06-JUN-2079 Time part: Number of minutes since midnight of the given date	4 bytes
Binary (fixed length)	*binary(n)*		Any binary representation (bit patterns) up to 255 bytes	*n* bytes, even if *n* is partially unused.
Binary (variable length)	*varbinary(n)*	*binary varying*	Any binary representation (bit patterns) up to 255 bytes	The number of bytes actually stored. No storage for space declared but not used.
Long text/BLOB	*text* and *image*		Text: Character data up to 2 GB. Image: Binary data up to 2 GB. The text and image datatypes are always variable length.	If not null, a 16-byte pointer is used on the data page, plus however many 2-KB pages are required to store the actual length. Text and image pages cannot be shared. A single byte entered in a text or image column requires its own 2-KB page (most being unused).
Boolean	*bit*		0 or 1	Bit datatypes share a byte with other bit columns of the same table. Hence, 8-bit columns of the same table use 1 byte of storage. If the table has only 1-bit columns, it still uses 1 byte, although 7 more such columns could be added "for free."

10

Batches, Transactions, Stored Procedures, and Triggers

Introduction

In this chapter, I'll discuss using Transact-SQL for more than interactive queries. When you send a query to the server, you are sending a command batch to SQL Server. But you can do more! I'll show you how to wrap up commands in a module that can be stored and cached at the server for later reuse (stored procedures), and I'll demonstrate how you can create modules that will automatically execute when some event occurs (triggers). Although I've briefly discussed transaction boundaries earlier in the book, in this chapter I'll help you better understand them and the effects of changes by multiple users.

Batches

A *batch* is one or several SQL Server commands that are dispatched and executed together. Because every batch sent from the client to the server requires handshaking between the two, sending a batch instead of sending separate commands can prove to be more efficient. Even a batch that does not return a result set (for example, a single INSERT statement) requires at least an acknowledgment that the command was processed and offers a status code for its level of success. At the server, the batch must be received, queued for execution, and so on.

Although commands are grouped and dispatched together for execution, each command is distinct from the others. Let's look at an example. Suppose that you need to execute 150 INSERT statements. Executing all 150 statements in one batch requires the processing overhead once, rather than the overhead incurred 150

times. I know of a real-life situation in which an application that took 5 to 6 seconds to complete 150 individual INSERT statements was changed so that all 150 statements were sent as one batch. The processing time decreased to well under 0.5 second, more than a tenfold improvement. And this was on a LAN, not on a slow network like a WAN or the Internet where the improvement would have been even more pronounced. (Try running an application with 150 batches to insert 150 rows over the Internet, and you'll be really sorry!)

Using a batch is a huge win. By using the Transact-SQL constructs that I presented in Chapter 9, such as conditional logic and looping, you can often perform sophisticated operations within a single batch and eliminate the need to carry on an extensive conversation between the client and the server. Those operations can also be saved on the server as a stored procedure, which allows them to execute even more efficiently. Using batches and stored procedures to minimize the client/server conversations is crucial for achieving high-performing applications. And now that more applications are being deployed on slower networks—such as WANs, the Internet, and dial-up systems—instead of on LANs only, using batches and stored procedures is crucial.

Every SELECT statement (except for those used for assigning a value to a variable) generates a result set. Even a SELECT statement that finds zero rows returns a result set that describes the columns that were selected. Every time the server sends a result set back to the client application, it must send *metadata* as well as the actual data. The metadata describes the result set to the client. You can think of metadata in this way: "Here is a result set with eight columns. The first column is named *last_name* and is of type *char(30)*. The second column is...."

Obviously, then, executing a single SELECT statement with a WHERE clause formulated to find (in one fell swoop) all 247 rows that meet your criteria is much more efficient than separately executing 247 SELECT statements that each returns one row of data. In the former case, one result set is returned. In the latter case, 247 result sets are returned—the performance difference is striking. Using batches might seem like a painfully obvious necessity, yet many programmers still write applications that perform poorly because they do not use batches. The problem is especially common for developers who have worked on ISAM or similar sequential files doing row-at-a-time processing. Unlike ISAM, SQL Server works best with *sets* of data, not individual rows of data, so that you can minimize conversations between the server and the client application.

Following is a simple batch, issued from ISQL. Even though three unrelated operations are being performed, I can package them in a single batch to conserve bandwidth.

```
INSERT authors VALUES(etc.)
SELECT * FROM authors
UPDATE publishers SET pub_id= (etc.)
GO
```

NOTE GO is not an SQL command. It is the end-of-batch signal that tells ISQL and ISQL/w that everything since the last GO should be sent to the server for execution. All commands between GOs are sent together in a batch for execution. There is nothing special about the word GO—the end-of-batch signal is specific to the front-end tool (ISQL and ISQL/w use GO), and the server is not aware of it. With a custom application, a batch is executed with a single **dbsqlexec** from DB-Library or **SQLExecute** from ODBC.

Transactions

Like a batch, a user-declared *transaction* typically consists of several SQL commands that read and update the database. But unlike a batch, a transaction doesn't make any permanent changes until a COMMIT statement is issued, or a transaction can undo its changes when a ROLLBACK statement is issued. With a batch, each command is separately and automatically committed. When a transaction is declared or SQL Server is configured for *implicit transactions,* commands are performed as a unit. Usually, both batches and transactions contain multiple commands dispatched together. A single transaction can span across batches (although that's a bad thing to do from a performance perspective), and a batch can contain multiple transactions.

Following is a simple transaction. The BEGIN TRAN and COMMIT TRAN statements cause the commands between them to be performed as a unit.

```
BEGIN TRAN
INSERT authors VALUES(etc.)
SELECT * FROM authors
UPDATE publishers SET pub_id= (etc.)
COMMIT TRAN
GO
```

Transaction processing in SQL Server assures that all commands within a transaction are performed as a unit—even in the presence of a hardware or general system failure. Such transactions are referred to as having the ACID properties (atomicity, consistency, isolation, and durability). (For more information about the ACID properties, refer to pages 38 and 39 in Chapter 2, "A Tour of Microsoft SQL Server.")

Explicit and Implicit Transactions

By default, SQL Server treats each statement, whether dispatched individually or as part of a batch, as independent and immediately commits it. If you want multiple statements to be part of a transaction, you must wrap the group of statements within BEGIN TRANSACTION and COMMIT or ROLLBACK TRANSACTION statements. You can also configure SQL Server to implicitly start a transaction

by using *SET IMPLICIT_TRANSACTIONS ON* or by turning the option on globally using **sp_configure 'user options', 2**. More precisely, take the previous value for the *'user options'* setting and OR it with (decimal) 2, which is the mask for IMPLICIT_TRANSACTIONS. For example, if the previous value was (decimal) 8, you'd set it to 10, since 8|2 is 10. If bit operations like 8|2 are somewhat foreign to you, let SQL Server do the work. You can issue a *SELECT 8|2* in ISQL/w and the row returned will be 10. (But be careful not to assume that you just add 2 to whatever is already there—for example, 10|2 is 10, not 12.)

If implicit transactions are enabled, all statements are considered part of a transaction and no work is committed until and unless an explicit COMMIT TRAN (or synonymously, COMMIT WORK or simply COMMIT) is issued. This is true even if all the statements in the batch have executed: you must issue a COMMIT in a subsequent batch before any work is made permanent.

Error Checking in Transactions

One of the most common mistakes that developers make with SQL Server is to assume that *any* error within a transaction will cause the transaction to automatically roll back. If conditional action should result from a possible error, your multistatement transactions should check for errors by selecting the value of @@ERROR after each statement. If a nonfatal error is encountered and you do not take action on it, processing moves on to the next statement. Only fatal errors cause the batch to be automatically aborted.

TIP Neither a query that finds no rows meeting the criteria of the WHERE clause nor a searched UPDATE statement that affects no rows is an error, and @@ERROR is not set to a nonzero value for either case. If you want to check for "no rows affected," use @@ROWCOUNT, not @@ERROR.

Syntax errors will always cause the entire batch to be aborted, as will references to objects that don't exist (for example, a SELECT from a table that doesn't exist). Typically, you'll work out syntax errors and correct references to objects before you put an application into production, so these won't be much of an issue. However, a syntax error on a statement dynamically built and executed using EXECUTE('*string*') cannot be caught until execution, so this type of syntax error does not abort the batch and processing proceeds to the next statement.

Errors that occur in a production environment are typically resource errors that should be encountered infrequently, especially if you have sufficiently tested to make sure that your configuration settings and environment are appropriate. Out-of-resource errors occur when the system runs out of locks, when there is not enough memory to run a procedure, and so on. For these types of fatal errors, conditional action based on @@ERROR is moot since the batch will automatically be aborted.

The following errors are common:

- Lack of permissions on an object
- Constraint violations
- Duplicates encountered while trying to update or insert a row
- Deadlocks with another user
- NOT NULL violations
- Illegal values for the current datatype

Following is a SQL Server chat group posting from a user who encountered a nonfatal execution error and assumed that the entire transaction should have been automatically aborted. When that didn't happen, the user assumed that SQL Server must have a grievous bug. Here's the posting:

> Using SQL 6.5 in the example below, the empty tables *b* and *c* each get an insert within one tran. The insert into table *c* is fine, but the insert to *b* fails a DRI reference, but the table *c* still has a row in it. Isn't this a major bug?! Should the tran not have been rolled back implicitly!?
>
> —*Keith*

```
create table a (
a char(1) primary key)

create table b (
b char(1) references a)

create table c (
c char(1))
go

create proc test as
begin transaction
insert c values ( 'X' )
Insert b values ( 'X' )  --Fails reference
commit transaction
go

exec test
go

select *
from c  --Returns  'X'  !!
```

The statements, however, are performing *as expected;* the bug is in the way the user wrote the transaction. The transaction has not checked for errors for each statement and *unconditionally* commits at the end. So even if one statement fails with a nonfatal execution error like a constraint or permissions violation, execution proceeds to the next statement. Ultimately the COMMIT is executed, so all the statements without errors are committed. *That's exactly what the procedure has been told to do.* If you want to roll back a transaction if *any* error occurs, you must check @@ERROR or use SET XACT_ABORT.

Here is an example of the procedure rewritten to perform error checking, with a branch to perform a rollback if any error is encountered:

```
CREATE PROC test as
BEGIN TRANSACTION
INSERT c VALUES ('X')
    IF (@@ERROR <> 0) GOTO on_error
INSERT b VALUES ('X')  -- Fails reference
    IF (@@ERROR <> 0) GOTO on_error
COMMIT TRANSACTION
RETURN(0)

on_error:
ROLLBACK TRANSACTION
RETURN(1)
```

This simple procedure illustrates the power of Transact-SQL. The global variable @@ERROR is set for the connection after each statement. A value of 0 for @@ERROR means no error occurred. Given the data the user provided, the INSERT statement on table *b* will fail with a foreign key violation. The error message for that type of failure is error 547, with text such as this:

```
INSERT statement conflicted with COLUMN FOREIGN KEY constraint
'FK__b__b__723BFC65'. The conflict occurred in database 'pubs',
table 'a', column 'a'
```

Consequently, @@ERROR would be set to 547 following that INSERT statement. Therefore, the *IF (@@ERROR <> 0)* statement evaluates as TRUE, and execution follows the *GOTO* to the *on_error:* label. Here, the transaction is rolled back. The procedure terminates with a return code of 1 because the *RETURN(1)* statement was used. Had the branch to *on_error:* not been followed (by virtue of @@ERROR not being 0), the procedure would have continued line-by-line execution. It would have reached *COMMIT TRANSACTION,* and then it would have returned value 0 and never made it all the way to the *on_error:* section.

As you do with most programming languages, you should make sure that a status is returned from a procedure to indicate success or failure (and/or other possible outcomes) and that those return status codes are checked from the calling rou-

tines. In Keith's original code, an "EXEC test" invoked the procedure. This approach is perfectly legal and could be done equally well with a procedure that returns 0 for SUCCESS and 1 for FAILURE. However, simply using "EXEC test" will not directly provide information about whether the procedure performed as expected. A better method is to use a local variable to examine the return code from that procedure:

```
DECLARE @retcode int
EXEC @retcode=test
```

Following execution of the test procedure, the local variable *@retcode* will have the value 0 (if no errors occurred in the procedure) or 1 (if execution branched to the *on_error:* section).

We added the SET XACT_ABORT option in version 6.5 to help users like Keith. If this option is set, any error, not just a fatal error (equivalent to checking *@@ERROR <> 0* after every statement), will terminate the batch. Here is another way to ensure that *nothing* is committed if *any* error is encountered:

```
CREATE PROC test AS
SET XACT_ABORT ON
BEGIN TRANSACTION
INSERT c VALUES ('X')
INSERT b VALUES ('X')  -- Fails reference
COMMIT TRANSACTION
GO

EXEC test
GO

SELECT * FROM c
```

The output:

```
(0 rows affected)
```

Note that the name of the XACT_ABORT option is a bit of a misnomer because the current batch, not simply the transaction, will be immediately aborted if an error occurs, just as it is when a fatal resource error is encountered. This has consequences that might not be immediately apparent. For example, if you issued two transactions within one batch, the second transaction would never be executed because the batch would be aborted before the second transaction got a chance to execute. More subtly, suppose that I wanted to use good programming practice and check the return status of the procedure above. (Note that even though it does not explicitly do a RETURN(), every procedure has a return status by default, with 0 indicating SUCCESS.)

I'd write a batch like this:

```
DECLARE @retcode int
EXEC @retcode=test
SELECT @retcode
```

Yet there is a subtle but important problem here. If the procedure has an error, the SELECT @RETCODE statement will never be executed: the entire batch will be aborted by virtue of the SET XACT_ABORT statement. This is why I recommend checking @@ERROR instead of using SET XACT_ABORT. Checking @@ERROR after each statement is a bit more tedious, but it gives you finer control of execution in your procedures.

No doubt, error handling will be improved in future releases. Unfortunately, error handling in SQL Server 6.5 can be somewhat messy and inconsistent. For example, there is no way to install a routine that means "Do this on any error" (other than SET XACT_ABORT, which aborts but does not let you specify the actions to be performed). Instead, you must use something similar to the preceding examples that check @@ERROR and then do a GOTO. In addition, there is currently no easy way to determine in advance which errors might be considered fatal so that the batch can be aborted, versus which errors are nonfatal so that the next statement can be executed. In most cases, an error with severity level of 16 or higher is fatal and the batch will be aborted. But syntax that refers to nonexistent functions are level 15 errors, yet the batch is still aborted. Although you can use an @@ERROR to return the specific error number, no global variable such as @@SEVERITY is available to indicate the error's severity level. Instead, you must subsequently select from the *sysmessages* table to see the severity level of the last error. To further complicate matters, some level 16 errors are not fatal. Following is a list of the most common nonfatal level 16 errors:

Error	*Error Message*
515	Attempt to insert the value NULL into column *'%.*s'*, *table '%.*s'*; column does not allow nulls. *%s* fails.
544	Attempt to insert explicit value for identity column in table *'%.*s'* when IDENTITY_INSERT is set to OFF.
547	*%s* statement conflicted with *%s %s* constraint *'%.*s'*. The conflict occurred in database *'%.*s'*, table *'%.*s'%s%.*s%s*.
550	The attempted insert or update failed because the target view either specifies WITH CHECK OPTION or spans a view which specifies WITH CHECK OPTION and one or more rows resulting from the operation did not qualify under the CHECK OPTION constraint.

Admittedly, the rules are hardly consistent, and this area is ripe for some attention in future releases. As you write your procedures, you should keep in mind that they could be automatically aborted due to an unexpected fatal error or you can cause them to abort with a nonfatal error. In many applications, you should add retry logic to your error handler to attempt to reexecute a command if it is aborted due to a deadlock (error 1205).

Transaction Isolation Levels

The isolation level at which your transaction runs determines your application's sensitivity to changes made by others, and consequently it also determines how long your transaction will need to hold locks to potentially protect against changes made by others. SQL Server 6.5 offers three isolation-level behaviors (although syntactically four options are available, if REPEATABLE READ is included):

- READ UNCOMMITTED (dirty read)
- READ COMMITTED (default—READ COMMITTED is equivalent to the term CURSOR STABILITY, which is used by several other products such as IBM DB/2)
- SERIALIZABLE

Your transactions will behave differently depending on which isolation level is set. The saying "Not to decide *is* to decide" applies here, because *every transaction has an isolation level whether you've specified it or not.* It makes sense for you to understand the levels and choose the one that best fits your needs.

NOTE The syntactical options listed above correspond to the SQL standard isolation levels, which are discussed in detail in the Chapter 3 section entitled "The Transaction Manager." In this section, I will discuss only information that wasn't covered earlier. See Chapter 3 if you need to refresh your memory.

When using the READ UNCOMMITTED option, keep in mind that you should deal with inconsistencies that might result from dirty reads. Because share locks are not issued and exclusive locks of other connections are not honored (that is, the data pages can be read, even though they are supposedly locked), it is possible to get some spurious errors during execution when using the READ UNCOMMITTED option. A moment later, reexecuting the same command is likely to work without error. To shield your end users from such errors, your applications using isolation level 0 (dirty read) should be prepared to retry due to spurious 605, 606, 624, or 625 errors. One of these errors might get raised to falsely indicate

that the database is inconsistent. In such a case, what frequently happens is that an update or insert has been rolled back along with its page allocations, so you've read data pages that no longer exist and logically never did. Ordinarily, locking will prevent such inconsistencies, but READ UNCOMMITTED takes some shortcuts and the errors can still occur. The retry logic should be identical to the logic that would be used to retry on a deadlock condition (error 1205).

NOTE Of course, you don't *have* to add retry logic, but without it the command will be aborted and your end user might see a scary and confusing message. Adding a good error handler can make your application much better behaved by enabling an automatic retry, and the user will never know when such an error is raised.

Phantoms

SQL Server allows you to specify REPEATABLE READ, but this is currently simply a synonym for SERIALIZABLE. As currently implemented in SQL Server, both the SERIALIZABLE and REPEATABLE READ options prevent phantoms from occurring. (As you'll recall from Chapter 3, phantoms occur when updated rows suddenly appear in data when it is revisited during a query or another transaction.) I wouldn't use SERIALIZABLE and REPEATABLE READ as interchangeable terms, however. A future SQL Server release will likely loosen the consistency characteristics of REPEATABLE READ and phantoms will be possible. I recommend instead that you specify the isolation level that best fits your current needs. In some operations, REPEATABLE READ might be all you require, and protecting against phantoms might incur an additional and unnecessary penalty on concurrency in the future.

The Future...

Suppose that you're working with Jane Doe's records, and when you issue a requery you need to ensure that only her records haven't been changed. In future releases, it might be possible to lock the range of records with the Jane Doe key (called a *key range*). This would be a preferable situation if SQL Server had to scan data to solve a join, for example. The scan wouldn't have to place locks that would prevent other transactions from inserting new rows into the ranges they scanned indirectly.

In version 6.5, *all* the scanned data would have to be kept locked and new rows could not be inserted. The current locking is more coarse than you might require, and thus concurrency is reduced more than is necessary. Because this situation is likely to change in a future release, you should be aware that the semantics of REPEATABLE READ might allow phantoms in the future.

Nothing illustrates the differences and effects of isolation levels like seeing them for yourself with some simple examples. I urge you to run the following examples. To do so, you'll need to establish two connections to SQL Server. Remember that each connection is *logically* a different user, even if both connections use the same login ID. (Locks made by one connection affect the other connection, even if both are logged in as the same user.) In case you can't run these examples, I'll show the SQL scripts and output here.

EXAMPLE 1

This example shows a dirty read. A simple command file on the accompanying CD-ROM, RUN_ISOLATION1.CMD spawns two simultaneous ISQL.EXE sessions, running the scripts ISOLATION_CNX1.SQL and ISOLATION_CNX2.SQL. Following are the scripts and the output (in bold) as contained in the output files ISOLATION_CNX1.OUT and ISOLATION_CNX2.OUT.

NOTE The procedures **sem_set** and **sem_wait** provide a simple synchronization mechanism between the two connections. They do nothing more than set a value and then poll for a value—a handy technique. You can create these procedures by running the SEMAPHORE.SQL script, which is included on the companion CD-ROM.

But this mechanism won't always do the job as it does in this case. For example, if I put **set_sem** within a transaction, it could be aborted or rolled back. Also, for illustration purposes, I sometimes use simple delays here to coordinate the two scripts. Relying on timing like this creates the potential for a race condition, and it's not how you'd want to program a production application.

```
-- ISOLATION_CNX2.SQL
USE pubs
GO

EXEC sem_wait 1     -- Wait until other connection says can start
BEGIN TRAN
    UPDATE authors SET au_lname='Smith'
    -- Give other connection chance to read uncommitted data
    WAITFOR DELAY "000:00:20"
ROLLBACK TRAN

EXEC sem_set 0      -- Tell other connection done

(23 rows affected)
```

```
-- ISOLATION_CNX1.SQL
-- Illustrate Read Uncommitted (aka "Dirty Read")
--
USE pubs
GO
EXEC sem_set 0     -- Clear semaphore to start
GO

-- First verify there's only one author named 'Smith'
SELECT au_lname FROM authors WHERE au_lname='Smith'
GO

au_lname
--------------------------
Smith

(1 row affected)

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

-- Signal other connection that it's ok to update, and then wait
-- for it to proceed. (Obvious possible race condition here
-- this is just for illustration.)

EXEC sem_set 1
-- Wait 10 secs for other connection to update
WAITFOR DELAY "000:00:10"
-- Check again for authors of name 'Smith'.  Now find 23 of them,
-- even though the other connection doesn't COMMIT the changes.
SELECT au_lname FROM authors WHERE au_lname='Smith'

au_lname
--------------------------
Smith
Smith
Smith
Smith
Smith
Smith
Smith
Smith
Smith
Smith
Smith
Smith
Smith
```

```
Smith
Smith
Smith
Smith
Smith
Smith
Smith
Smith
Smith
Smith

(23 rows affected)

IF (@@ROWCOUNT > 0)
    PRINT 'Just read uncommitted data !!'

Just read uncommitted data !!

-- Now the other connection will roll back its changes:
EXEC sem_wait 0

-- Now check again for authors of name 'Smith'.
-- Find only one now, because other connection did a rollback.
SELECT au_lname FROM authors WHERE au_lname='Smith'

au_lname
--------------------------
Smith

(1 row affected)
```

The scripts above, running simultaneously, illustrate that by setting the isolation level to READ UNCOMMITTED, you can read data that logically never existed (a dirty read). The update to *Smith* was never committed by the ISOLATION_CNX2.SQL script, yet the other connection read it. Not only did the connection read the update, it did so immediately and did not have to wait for the exclusive lock of the second connection updating the data to be released. In this example, concurrency is very high, but consistency of the data is not maintained. By changing the isolation level back to the default (READ COMMITTED), the same scripts would never see more than one row containing *Smith*. But with READ COMMITTED, the connection reading the data must wait until the updating transaction is done, and this is the tradeoff: higher consistency is achieved (the uncommitted rows are not seen by others), but concurrency is reduced.

EXAMPLE 2

This example illustrates the semantic differences between selecting data under the default isolation level of READ COMMITTED (cursor stability) and under the isolation level of SERIALIZABLE. Use the command file RUN_ISOLATION3.CMD to run these two scripts simultaneously. Output will be written to ISOLATION3-_CNX1.OUT and ISOLATION3_CNX2.OUT.

```
-- ISOLATION3_CNX2.SQL

USE pubs
GO

-- Wait until other connection says can start, then sleep 10 secs
EXEC sem_wait 1

WAITFOR DELAY "000:00:10"
GO
UPDATE authors SET au_lname='Smith'
GO
(23 rows affected)

EXEC sem_set 0     -- Tell other connection done with first part
EXEC sem_wait 1    -- Wait until other connection says can start,
                   -- then sleep 10 secs
GO

WAITFOR DELAY "000:00:10"
GO

UPDATE authors SET au_lname='Jones'
GO
(23 rows affected)

-- ISOLATION3_CNX1.SQL
-- Illustrate Read Repeatable/Serializable (aka level 2 & 3)
USE pubs
GO

UPDATE authors SET au_lname='Doe' -- Make sure no Smith or Jones
(23 rows affected)

EXEC sem_SET 0    -- Clear semaphore to start
GO

-- First verify there are no authors named 'Smith'
SELECT au_lname FROM authors WHERE au_lname='Smith'
GO
```

```
au_lname
--------------------

(0 rows affected)

SET TRANSACTION ISOLATION LEVEL READ COMMITTED
GO

-- Signal other connection it's ok to update, then wait for it to
-- proceed. (Obvious possible race condition here - this is just
-- for illustration.)
EXEC sem_SET 1
GO

BEGIN TRAN
SELECT au_lname FROM authors WHERE au_lname='Smith'
GO

au_lname
--------------------

(0 rows affected)

WAITFOR DELAY "000:00:15"

SELECT au_lname FROM authors WHERE au_lname='Smith'

au_lname
--------------------
 Smith
 Smith
 Smith
 Smith
 Smith
 Smith
 Smith
 Smith
 Smith
 Smith
 Smith
 Smith
 Smith
 Smith
 Smith
 Smith
 Smith
 Smith
 Smith
```

```
Smith
Smith
Smith
Smith
Smith

(23 rows affected)

COMMIT TRAN
GO

EXEC sem_wait 0
EXEC sem_SET 1
GO

-- Now do the same thing, but with SERIALIZABLE isolation

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
GO

BEGIN TRAN
SELECT au_lname FROM authors WHERE au_lname='Jones'
GO

au_lname
--------------------

(0 rows affected)

-- Wait for other connection to have a chance to make and commit
-- its changes
WAITFOR DELAY "000:00:15"

SELECT au_lname FROM authors WHERE au_lname='Jones'

au_lname
--------------------

(0 rows affected)

COMMIT TRAN
GO

-- Now notice that Jones updates have been done
SELECT au_lname FROM authors WHERE au_lname='Jones'
GO
```

```
au_lname
----------------------
 Jones
 Jones
 Jones
 Jones
 Jones
 Jones
 Jones
 Jones
 Jones
 Jones
 Jones
 Jones
 Jones
 Jones
 Jones
 Jones
 Jones
 Jones
 Jones
 Jones
 Jones
 Jones
 Jones

(23 rows affected)

EXEC sem_SET 0     -- Tell other connection done
```

As you can see, when the isolation level was set to READ COMMITTED, doing the same SELECT twice within a transaction yielded totally different results: first no Smiths were found, and then 23 of them appeared. If this were a banking transaction, these results would be unacceptable. For applications that are less sensitive to minor changes, or when business functions guard against such inconsistencies in other ways, this behavior might be acceptable.

After changing the isolation level to SERIALIZABLE, I got the same result (no Joneses) with both SELECT statements. Immediately after the transaction has committed, the other connection updated all the names to Jones. So when I again selected for Jones (immediately after the transaction was committed), the update took place—but at some cost. In the first case, the other connection was able to do the update as soon as the first SELECT was processed. It did not have to wait for the second query and subsequent transaction to complete. In the second case, that second connection had to wait until the first connection completed the transaction in its entirety. In this example, concurrency was significantly reduced but consistency was perfect.

Additional Characteristics of Transactions

In addition to isolation levels, it is important to understand the following characteristics of transactions.

First, a single UPDATE, DELETE, or INSERT/SELECT statement that affects multiple rows is always an atomic operation and must complete without error or it is automatically rolled back. For example, if you did a single UPDATE statement that updated all rows, but one row failed a constraint, the operation would be terminated with no rows updated. Because there is no way to do error checking for each row within such a statement, any error rolls back the statement. However, the batch is not aborted. Execution proceeds to the next available statement in the batch. This will occur even in INSERT operations that are based on a SELECT statement (INSERT/SELECT or SELECT INTO).

Like the statements mentioned above, modifications made by a trigger are always atomic with the underlying data modification statement. For example, if an update trigger attempts to update data but fails, the underlying data modification operation is rolled back. Both operations must succeed or neither does.

It might not be obvious, but Example 2 (the SERIALIZABLE example) shows that a transaction can affect pure SELECT operations in which no data is modified. The transaction boundaries define the statements between which a specified isolation level will be assured. Two identical SELECT statements might behave differently if they were executed while wrapped by BEGIN TRAN/COMMIT TRAN with an isolation level of REPEATABLE READ than they would behave if they were executed together in the same batch, but not within a transaction.

Finally, another point that might not seem obvious: the SET TRANSACTION ISOLATION LEVEL setting applies to the entire batch, no matter where or in what order it appears in the batch. For example, if you have been operating with REPEATABLE READ, but you want to change the setting to READ COMMITTED, it would seem natural to put the READ COMMITTED statement at the end of your batch. However, doing so would change the entire batch to READ COMMITTED—including the statements that appeared before it.

Stored Procedures

Now that you've learned about batches and transactions, it's time to discuss another important capability of Transact-SQL. *Stored procedures* enable you to cache commands at the server for later use. To create a stored procedure, you take a batch and wrap it inside a CREATE PROCEDURE *proc_name* AS statement. After that, you use Transact-SQL with nothing special except for declaring what parameters will be passed to the procedure.

To demonstrate how easy it is to create a stored procedure, I've written a simple procedure called **get_author** that takes one parameter, the *author_id,* and returns the names of any authors that have IDs equal to whatever character string is passed to the procedure.

```
CREATE PROC get_author @au_id varchar(11)
AS
SELECT au_lname, au_fname
FROM authors
WHERE au_id=@au_id
```

This procedure can be subsequently executed with syntax like this:

```
EXEC get_author '172-32-1176'
EXEC get_author @au_id='172-32-1176'
```

You can see that the parameters can be passed anonymously by including values in order, or the parameters can be explicitly named so that the order in which they are passed is not important.

If the procedure is the first statement in the batch, using EXEC is optional. However, I think it's always best to use EXEC so that later you won't wind up scratching your head with confusion, wondering why your procedure won't execute (only to realize later that it is no longer the first statement of the batch).

In practice, you'll probably want a procedure like the one above to be a bit more sophisticated. Perhaps you'd want to search for author names that begin with a partial ID that you pass. If you choose not to pass anything, rather than give you an error message stating that the parameter is missing, the procedure should show all authors. And maybe you'd like to return some value as a variable, distinct from the result set returned and from the return code, that you would use to check for successful execution.

You can do this by passing an *output* parameter, which has a *pass-by reference* capability. Passing an output parameter to a stored procedure is similar to passing a pointer when calling a function in C. Rather than passing a value, you pass the address of a storage area in which the procedure will cache a value. That value is subsequently available to the SQL batch after the stored procedure has executed. For example, I might want an output parameter to tell me the number of rows returned by a SELECT statement. While the @@ROWCOUNT global variable has this information, it is maintained only for the last statement executed. If the stored procedure executed many statements, I'd need to put this value away for safekeeping. An output parameter provides an easy way to do this.

Here's a simple procedure that selects all the rows from the *authors* table and all the rows from the *titles* table. It also sets an output parameter for each table based on @@ROWCOUNT. This value is subsequently available to the calling batch by checking the variables passed as the output parameters.

```
CREATE PROC count_tables @authorcount int OUTPUT,
@titlecount int OUTPUT
AS
SELECT * FROM authors
SELECT @authorcount=@@ROWCOUNT
SELECT * FROM titles
SELECT @titlecount=@@ROWCOUNT
RETURN(0)
```

The procedure would then be executed like this:

```
DECLARE @a_count int, @t_count int
EXEC count_tables @a_count OUTPUT, @t_count OUTPUT
```

TIP Variables are always local, so I could have used the same names for both the variables in the procedure and those passed by the batch as output parameters. In fact, this is probably the most common way to invoke them. I chose not to do this here to make it clear that the variables don't need to have the same name. Even with the same name, they are in fact different variables because their scoping is different.

This procedure will return all the rows from both tables. In addition, the variables *@a_count* and *@t_count* will retain the row counts from the *authors* and *titles* tables, respectively.

```
SELECT authorcount=@a_count, titlecount=@t_count
```

Here's the output:

```
authorcount    titlecount
-----------    ----------
23             18
```

When creating a stored procedure, you can reference a table, a view, or another stored procedure that does not currently exist. (In the latter case, you'll get a warning message informing you that a referenced object does not exist. As long as the object exists at the time the procedure is executed, all will be fine.)

Nested Stored Procedures

Stored procedures can be nested and can call other procedures. A procedure invoked from another procedure can also then invoke yet another procedure. In such a transaction, the top-level procedure has a nesting level of 1. The first subordinate procedure has a nesting level of 2. If that subordinate procedure subsequently invokes another stored procedure, the nesting level will be 3, and so on, to a limit of 16 nesting levels. If the 16-level limit is reached, a fatal error will result, the batch will be aborted, and any open transactions will be rolled back. The nesting-level limit prevents stack overflows that can result from procedures recursively calling themselves infinitely. The limit allows a procedure to recursively call itself only 15 subsequent times (for a total of 16 procedure calls). To determine how deeply a procedure is nested at runtime, you can select the global variable @@NESTLEVEL.

Unlike nesting levels, SQL Server has no practical limit on the number of stored procedures that can be invoked from a given stored procedure. For example, a main stored procedure could invoke hundreds or more subordinate stored procedures. If the subordinate procedures don't invoke other subordinate procedures, the nesting level never reaches a depth greater than 2.

An error in a nested (subordinate) stored procedure is not necessarily fatal to the calling stored procedure. When invoking a stored procedure from another stored procedure, it's a good idea to use a RETURN statement and check the return value in the calling procedure. In this way, you can conditionally work with error situations (as shown in the factorial example below).

Recursion in Stored Procedures

Stored procedures can perform nested calls to themselves, a technique known as *recursion*. Only powerful programming languages such as C—and Transact-SQL—support recursion. Recursion is a technique by which the solution to a problem can be expressed by applying the solution to subsets of the problem. Programming instructors usually demonstrate recursion by having students write a factorial program using recursion to display a table of factorial values for *0!* through *10!*. Recall that a factorial of a positive integer *n*, written as *n!*, is the multiplication of all integers from 1 through *n*. For example:

```
8!    = 8 x 7 x 6 x 5 x 4 x 3 x 2 x 1 = 40320
```

(Zero is a special case—*0!* is defined as equal to 1.)

I can write a stored procedure that computes factorials, and I can do the recursive programming assignment in Transact-SQL.

NOTE I'm not saying recursion is the best way to solve this problem. An iterative (looping) approach is probably better. This example is simply for illustration purposes.

```
-- Use Transact-SQL to recursively calculate factorial
-- of numbers between 0 and 12
-- Parameters greater than 12 are disallowed as result
-- overflows the bounds of an int

CREATE PROC factorial @param1 int
AS
DECLARE @one_less int, @answer int
IF (@param1 < 0 OR @param1 > 12)
    BEGIN
       -- Illegal parameter value. Must be between 0 and 12.
        RETURN -1
    END

IF (@param1=0 or @param1=1)
    SELECT @answer=1
ELSE
    BEGIN
    SELECT @one_less=@param1 - 1
    EXEC @answer=factorial @one_less -- Recursively call itself
    IF (@answer= -1)
        BEGIN
            RETURN -1
        END

    SELECT @answer=@answer * @param1
        IF (@@ERROR <> 0)
            RETURN -1
    END

RETURN(@answer)
```

Note that when the procedure is initially created, a warning message like the one shown below will indicate that the procedure is referencing a procedure that doesn't currently exist (which is itself in this case):

```
Cannot add rows to Sysdepends for the current stored procedure
because it depends on the missing object 'factorial'. The stored
procedure will still be created.
```

Once the procedure exists, I can use it to display the standard factorial table that students generate in C programming classes:

```
DECLARE @answer numeric, @param int
SELECT @param=0
WHILE (@param <= 12)
BEGIN EXEC @answer=factorial @param IF (@answer= -1) BEGIN
RAISERROR('Error executing factorial procedure.', 16, -1) RETURN END
SELECT CONVERT(varchar, @param) + '! = ' + CONVERT(varchar, @answer)
SELECT @param=@param + 1
END
```

Here's the return table:

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
```

I stopped at *12!* in the **factorial** procedure because *13!* is 6,227,020,800, which exceeds the range of a 32-bit (4-byte) integer. However, even without the range limit, I could have gone only to *15!* because I'd have reached the maximum nesting depth of 16, which includes recursive calls. The procedure would have then terminated with error 217:

```
Maximum stored procedure nesting level exceeded (limit 16)
```

Notice too that I return –1 if the procedure results in an error or is passed an illegal parameter. This return code can then be checked for an error.

In C, you need to be sure that you don't overflow your stack when you use recursion. Using Transact-SQL shields you from that concern, but it does so by steadfastly refusing to nest calls more than 16 levels deep. You can also watch @@NESTLEVEL and take appropriate action before reaching the hard limit. As is often the case with a recursion problem, an iterative solution could be performed without the restriction of nesting or worries about the stack.

Here is an iterative approach. To illustrate that there is no restriction of 16 levels since it is simple iteration, I take this exercise further, to *33!*. The value from a RETURN statement, however, is always an *int*. So this version uses an output parameter declared as *numeric(38,0)* instead of *int*. (This example also illustrates that a numeric datatype with scale of 0 can be used as an alternative to *int* for

integer operations that require values larger than the 4-byte *int* can handle.) I stop at *33!* because *34!* would overflow the precision of a *numeric(38,0)* variable. (Note that SQL Server must be started with the **-p38** flag to increase the maximum precision to 38 digits instead of the default of 28.) Here's the iterative solution:

```
-- Alternative iterative solution does not have the restriction
-- of 16 nesting levels
CREATE PROC factorial2 @param1 int, @answer NUMERIC(38,0) OUTPUT
AS
DECLARE @counter int
IF (@param1 < 0 OR @param1 > 33)
    BEGIN
    RAISERROR ('Illegal Parameter Value. Must be between 0 and 33',
        16, -1)
    RETURN -1
    END

SELECT @counter=1, @answer=1

WHILE (@counter < @param1 AND @param1 <> 0 )
    SELECT @answer=@answer * (@counter + 1), @counter=@counter + 1

RETURN
GO

DECLARE @answer numeric(38, 0), @param int
SELECT @param=0
WHILE (@param <= 33)
BEGIN
    EXEC factorial2 @param, @answer OUTPUT
    SELECT CONVERT(varchar(45), @param) + '! = '
        + CONVERT(varchar(45), @answer)
    SELECT @param=@param + 1
END
```

And here's the output table:

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
```

```
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
21! = 51090942171709440000
22! = 1124000727777607680000
23! = 25852016738884976640000
24! = 620448401733239439360000
25! = 15511210043330985984000000
26! = 403291461126605635584000000
27! = 10888869450418352160768000000
28! = 304888344611713860501504000000
29! = 8841761993739701954543616000000
30! = 265252859812191058636308480000000
31! = 8222838654177922817725562880000000
32! = 263130836933693530167218012160000000
33! = 8683317618811886495518194401280000000
```

Nested Transaction Blocks

It is syntactically acceptable for blocks of BEGIN TRANSACTION followed by COMMIT or ROLLBACK to be nested within other such blocks. This kind of nesting can also be done with calls to nested stored procedures. However, the semantics of such a formulation might not be what you would expect if you think that the transactions are truly nested: they are not. But the behavior is reasonable and predictable. A ROLLBACK rolls back all levels of the transaction, not only its inner block. A COMMIT TRAN does nothing to commit the transaction if the statement is not part of the outermost block, in which case it commits all levels of the transaction. So the behavior of a COMMIT or a ROLLBACK is not too orthogonal when the transaction blocks are nested. Only the outermost COMMIT is able to commit the transaction, but any ROLLBACK will roll back the entire transaction at all levels. If this were not true, a ROLLBACK in an outer transaction would not be able to perform its job, because the data would have already been committed. This behavior allows stored procedures (and triggers) to be executed automatically and in a predictable way without your needing to check the transaction state. A nested stored procedure that does a ROLLBACK will roll back the entire transaction, including work done by the top-level procedure. At this point, you will get a message similar to the one on the following page that warns of a mismatch of transaction blocks.

```
Msg 266, Level 16, State 1
Transaction count after EXECUTE indicates that a COMMIT or ROLLBACK
TRAN is missing. Previous count = 1, Current count = 0.
```

In addition, the blocks of BEGIN TRAN and COMMIT or ROLLBACK are determined only by what actually executes, not by what is present in the batch. If conditional branching occurs (via IF statements, for example) and one of the statements doesn't execute, the statement is not part of a block.

A common misconception about ROLLBACK is that it changes the flow of control, causing, for example, an immediate return from a stored procedure or a batch. However, flow of control continues to the next statement, which, of course, could be an explicit RETURN. ROLLBACK affects only the actual data; it does not affect local variables or SET statements. If local variables are changed during a transaction or if SET statements are issued, those variables and options *do not* revert to the values they had before the transaction started. Variables and SET options are not part of transaction control.

The global variable @@TRANCOUNT keeps count of the depth of executed BEGIN TRAN blocks. You can think of the behavior of COMMIT and ROLLBACK in this way: ROLLBACK performs its job for all levels of transaction blocks whenever @@TRANCOUNT is 1 or greater. A COMMIT commits changes *only* when @@TRANCOUNT is 1.

NOTE Beginning on page 476, I'll show you several examples of nesting transaction blocks. For illustration, I've noted the value of @@TRANCOUNT in the comments. If this discussion isn't totally clear to you, I suggest that you work through these examples by hand to understand why the value of @@TRANCOUNT is what it is, why it changes, and why the behavior of COMMIT or ROLLBACK acts in a way that depends on the @@TRANCOUNT value.

Executing a BEGIN TRAN statement always increments the value of @@TRANCOUNT. If no transaction is active, @@TRANCOUNT is 0. Executing a COMMIT TRAN decrements the value of @@TRANCOUNT. Executing a ROLLBACK TRAN rolls back the entire transaction and sets @@TRANCOUNT to 0. Executing either a COMMIT or a ROLLBACK when there is no open transaction (@@TRANCOUNT is 0) results in error 3902 or 3903, which states that the COMMIT or ROLLBACK request has no corresponding BEGIN TRANSACTION. In this case, @@TRANCOUNT is not decremented, so it can never go below 0.

Other errors, not of your making, are fatal as well—for example, out of memory, out of locks, or termination due to a deadlock. Such errors cause an open transaction to automatically roll back. If that occurs, @@TRANCOUNT is set to 0, signaling that no open transaction exists. The following incidents cause fatal errors:

- The system runs out of resources such as locks.
- The log runs out of space.
- Deadlock conditions exist.
- Protection exceptions occur (that is, sufficient permissions are unavailable on an object).
- A stored procedure cannot run (for example, it is not present or you don't have privileges).
- Reading or writing a stored procedure or trigger from the *sysprocedures* table or the cache is not possible because the system is out of memory or a procedure cache is too small.
- The maximum nesting level of stored procedure executions has been reached.

You can't plan precisely for all of these situations, and even if you could, a new release could likely add another one. In a production environment, fatal errors should be few and far between. But as is true in nearly all programming, it's up to you to decide whether you want to try to plan for every conceivable problem that could happen and then deal with each specifically, or whether you will accept the system default behavior. In SQL Server, the default behavior would typically be to raise an error for the condition encountered and then, when the COMMIT or ROLLBACK is executed, raise another that says it has no corresponding transaction. For my program, I would probably consider writing some retry logic in my ODBC or DB-Library application to deal with a possible deadlock condition. For the other error conditions, I'd probably go with the default error behavior. If I deploy my application with proper system management, such errors should be nonexistent or rare. I could also check @@TRANCOUNT easily enough before executing a ROLLBACK or COMMIT to ensure that a transaction on which to operate is open.

The nesting of transaction blocks provides a good reason for you *not* to name the transaction in a ROLLBACK statement. If, in a rollback, any transaction other than the top-level transaction is named, error 6401 will result:

```
Cannot rollback XXX - no transaction or savepoint of that name found.
```

It's fine to name the transaction in the BEGIN TRAN block, however. A COMMIT can also be named, and it won't prompt an error if it is not paired in the top-level branch since it basically is a NOOP in that case anyway (except that it decrements the value of @@TRANCOUNT). You might choose sometimes to name the transaction so that you can see which transactions were automatically rolled back and rolled forward at system startup if you set *recovery flags* to 1 using **sp_configure**. The error log and event log will then include useful messages similar to those shown on the next page.

```
96/11/06 10:52:10.04 spid1     Recovering database 'pubs'.
96/11/06 10:52:10.20 spid1     Recovery dbid 4 ckpt (1039,20) oldest
                               tran=(1044,12).
96/11/06 10:52:10.20 spid1     Roll forward transaction 'TRAN_B' in dbid 4.
96/11/06 10:52:10.44 spid1     Roll back transaction 'TRAN_A' - was
                               aborted in dbid 4.
96/11/06 10:52:10.53 spid1     1 transactions rolled forward in dbid 4.
96/11/06 10:52:10.53 spid1     1 transactions rolled back in dbid 4.
```

The following batch will result in an error. The statement *ROLLBACK TRAN B* will fail, with error 6401. The subsequent *ROLLBACK TRAN A* will succeed because it is the top-level transaction block:

```
-- To start with, verify @@TRANCOUNT is 0
SELECT @@TRANCOUNT
BEGIN TRAN A
    -- Verify @@TRANCOUNT is 1
    SELECT @@TRANCOUNT
    -- Assume some real work happens here
    BEGIN TRAN B
        -- Verify @@TRANCOUNT is 2
        SELECT @@TRANCOUNT
        -- Assume some real work happens here
    ROLLBACK TRAN B
-- @@TRANCOUNT is still 2, because the previous ROLLBACK
-- failed due to error 6401
SELECT @@TRANCOUNT -- Assume some real work happens here
ROLLBACK TRAN A
-- This ROLLBACK succeeds, so @@TRANCOUNT is back to 0
SELECT @@TRANCOUNT
```

The following example is arguably an improvement over the previous attempt. The first ROLLBACK will execute but will still result in an error. The second ROLLBACK will fail, with error 3903:

```
-- The ROLLBACK transaction request has no corresponding BEGIN
-- TRANSACTION. The first ROLLBACK did its job and there is no open
-- transaction.
-- To start with, verify @@TRANCOUNT is 0
SELECT @@TRANCOUNT
BEGIN TRAN A
    -- Verify @@TRANCOUNT is 1
    SELECT @@TRANCOUNT
    -- Assume some real work happens here
    BEGIN TRAN B
        -- Verify @@TRANCOUNT is 2
        SELECT @@TRANCOUNT
   -- Assume some real work happens here
```

```
    ROLLBACK TRAN -- Notice the tran is unnamed but works
   -- That ROLLBACK terminates transaction. @@TRANCOUNT is now 0.
    SELECT @@TRANCOUNT
-- The following ROLLBACK will fail because there is no open
-- transaction (that is, @@TRANCOUNT is 0)
ROLLBACK TRAN
-- @@TRANCOUNT does not go negative. It remains at 0.
SELECT @@TRANCOUNT
```

In the following example, the second ROLLBACK will not execute because @@TRANCOUNT is 0 following the prior ROLLBACK. If you syntactically nest transactions, be careful not to nest multiple ROLLBACK statements in a way that would allow more than one to execute. To illustrate that COMMIT behaves differently than ROLLBACK, note that the following example will run without error. However, the statement *COMMIT TRAN B* does not commit any changes. It does have the effect of decrementing @@TRANCOUNT, however.

```
-- To start with, verify @@TRANCOUNT is 0
SELECT @@TRANCOUNT
BEGIN TRAN A
    -- Verify @@TRANCOUNT is 1
    SELECT @@TRANCOUNT
    -- Assume some real work happens here
        BEGIN TRAN B
            -- Verify @@TRANCOUNT is 2
            SELECT @@TRANCOUNT
            -- Assume some real work happens here
        COMMIT TRAN B
-- The COMMIT didn't COMMIT anything, but does decrement
-- @@TRANCOUNT
    -- Verify @@TRANCOUNT is back down to 1:
    SELECT @@TRANCOUNT
    -- Assume some real work happens here
COMMIT TRAN A
-- The COMMIT on previous line does commit the changes and
-- closes the transaction
-- Since there's no open transaction, @@TRANCOUNT is again 0
SELECT @@TRANCOUNT
```

In summary, nesting transaction blocks is perfectly valid, but you must understand the semantics as I've discussed and shown above. If you understand when @@TRANCOUNT is incremented, decremented, and set to 0, and you know the simple rule for COMMIT and ROLLBACK, you can pretty simply produce the effect you want.

Savepoints

Often, users will nest transaction blocks, only to find that the behavior is not what they want. What they *really* want is for a *savepoint* to occur in a transaction. A savepoint provides a point up to which a transaction can be undone—it might have been more accurately named a "rollback point." A savepoint does not commit any changes to the database—only a COMMIT statement can do that.

SQL Server allows you to use savepoints via the SAVE TRAN statement, which doesn't affect the @@TRANCOUNT value. A rollback to a savepoint (not a transaction) does not affect the @@TRANCOUNT either. However, the rollback must explicitly name the savepoint: using ROLLBACK without a specific name will roll back the entire transaction.

In the first nested transaction block example shown on the preceding pages, a rollback to a transaction name failed with error 6401 because the name was not the top-level transaction. Had the name been a savepoint instead of a transaction, no error would result, as this example shows:

```
--To start with, verify @@TRANCOUNT is 0
SELECT @@TRANCOUNT
BEGIN TRAN A
    -- Verify @@TRANCOUNT is 1
    SELECT @@trancount
    -- Assume some real work happens here
    SAVE TRAN B
-- Verify @@TRANCOUNT is still 1. A savepoint does not affect it.
        SELECT @@TRANCOUNT
        -- Assume some real work happens here
    ROLLBACK TRAN B
-- @@TRANCOUNT is still 1, because the previous ROLLBACK
-- affects just the savepoint, not the transaction
SELECT @@TRANCOUNT
-- Assume some real work happens here
ROLLBACK TRAN A
-- This ROLLBACK succeeds, so @@TRANCOUNT is back to 0
SELECT @@TRANCOUNT
```

Stored Procedure Parameters

Stored procedures take parameters, and you can give parameters default values. If you do not supply a default value, a specific parameter will be required. If you do not pass a required parameter, an error like this will result:

```
Msg 201, Level 16, State 2
Procedure sp_passit expects parameter @param1, which was not supplied.
```

You can pass values by explicitly naming the parameters or by furnishing all the parameter values anonymously but in correct positional order. You can also use the keyword DEFAULT as a placeholder in passing parameters. NULL can also be passed as a parameter (or defined to be the default). Here's a simple example with results in bold:

```
CREATE PROCEDURE pass_params
@param0 int=NULL,    -- Defaults to NULL
@param1 int=1 ,      -- Defaults to 1
@param2 int=2        -- Defaults to 2
AS
SELECT @param0, @param1, @param2
GO

EXEC pass_params           -- PASS NOTHING - ALL Defaults
(null)    1    2

EXEC pass_params 0, 10, 20     -- PASS ALL, IN ORDER
0    10    20

EXEC pass_params @param2=200, @param1=NULL
-- Explicitly identify last two params (out of order)
(null)    (null)    200

EXEC pass_params 0, DEFAULT, 20
-- Let param1 default. Others by place.
0    1    20
```

Executing Batches, or What's Stored About a Stored Procedure?

Typically, when a batch of Transact-SQL commands is received from a client connection, the following high-level actions are performed:

Step 1: Parse commands and create the sequence tree. The command parser checks for proper syntax and translates the Transact-SQL commands into an internal format that can be operated on. The internal format is known as a *sequence tree* or *query tree*. The command parser handles these language events.

Step 2: Compile the batch. An execution plan is generated from the sequence tree. The entire batch is compiled, queries are optimized, and security is checked. The execution plan contains the necessary steps to check any constraints that exist. If a trigger exists, the call to that procedure is appended to the execution plan. (Recall that a trigger is really a specialized type of stored procedure. Its plan is cached, and the trigger does not need to be recompiled every time data is modified).

The execution plan includes:

- The complete set of necessary steps to carry out the commands in the batch or stored procedure
- The steps needed to enforce constraints (for example, for a foreign key, this would involve checking values in another table)
- A branch to the stored procedure plan for a trigger, if one exists

Step 3: Execute. During execution, each step of the execution plan is dispatched serially to a "manager" that is responsible for carrying out that type of command. For example, a data definition command (in DDL), such as CREATE TABLE, is dispatched to the DDL Manager. DML statements, such as SELECT, UPDATE, INSERT, and DELETE, go to the DML Manager. Miscellaneous commands, such as DBCC and WAITFOR, go to the Utility Manager. Calls to stored procedures (for example, EXEC **sp_who**) are dispatched to the Stored Procedure Manager. A statement with an explicit BEGIN TRAN interacts directly with the Transaction Manager.

Contrary to what many people think, stored procedures do not permanently store the execution plan of the procedure. (*This is a feature*—the execution plan is relatively dynamic.) Think for a moment about why it is important for the execution plan to be dynamic. As new indexes are added, preexisting indexes are dropped, constraints are added or changed, and triggers are added or changed; or as the amount of data changes, the plan can easily become obsolete.

So, what's stored about a stored procedure?

The sequence tree and the SQL statements that were used to create the procedure are stored, eliminating the need to reparse the Transact-SQL statements and re-create the sequence tree. The first time a stored procedure is executed after SQL Server was last restarted, the sequence tree is retrieved and an execution plan is compiled. The execution plan resides in the portion of memory known as the *procedure cache*. The execution plan is then cached, and it remains in the procedure cache for possible reuse until it is forced out in a least recently used (LRU) manner. Hence, a subsequent execution of the stored procedure can skip not only Step 1, *parsing,* but also Step 2, *compiling,* and go directly to Step 3, *execution*. Steps 1 and 2 always add some overhead and sometimes can be as costly as actually executing the commands. Obviously, if you can eliminate the first two steps in a three-step process, you've done well. That's what stored procedures let you do.

As I mentioned above, the execution plan of a stored procedure resides in the portion of SQL Server's cache known as the procedure cache. The procedure cache must be big enough to store at least one copy of the biggest execution plan you will execute. If it is not big enough, you will get an error stating that insufficient memory is available to run the procedure.

NOTE The SQL Server cache is composed of the procedure cache, which keeps execution plans of stored procedures, and the data cache. The amount of space that is allocated to each area is a function of the **sp_configure 'procedure cache'** setting, which is expressed as a percentage of available cache rather than as a specific amount. The remaining memory is then made available for data caching.

When you execute a stored procedure, if a valid execution plan exists in the procedure cache, it will be used (eliminating the parsing and sequencing steps). When the server is restarted, no execution plans will be in the cache, so the first time the server is restarted a stored procedure will be compiled.

TIP You can preload your procedure cache with execution plans for stored procedures by defining a startup stored procedure that executes the procedures you want to have compiled and cached.

After a procedure executes, its plan remains in the procedure cache and is reused the next time any connection executes the same procedure. However, an execution plan is not *reentrant*: that is, a copy of the plan must be available for every connection attempting to execute the procedure. So if you expect 10 different connections to simultaneously execute a stored procedure, you should expect 10 copies of the execution plan to be in memory (provided that you have sufficient space in the procedure cache).

Figures 10-1 and 10-2 show execution with and without a stored procedure:

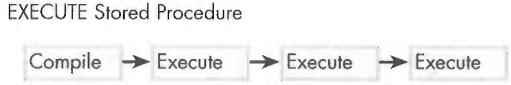


Figure 10-1. *Execution is more efficient with a stored procedure.*

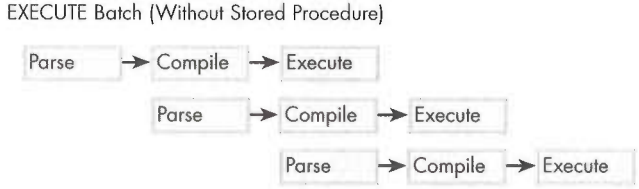


Figure 10-2. *Execution is less efficient without a stored procedure.*

Step 4: Recompile execution plans. By now it should be clear that sequence trees persist in the database but execution plans do not. Execution plans are cached in memory (in the procedure cache). But sometimes they can be invalidated and a new plan generated.

So, when is a new execution plan compiled?

- When a copy of the execution plan is not available in memory.
- When an index on a referenced table is dropped.
- When a table referenced in the procedure is altered using ALTER TABLE. This includes adding datatypes or adding, changing, or dropping constraints. Any changes to the table results in the schema column of the *sysobjects* table being incremented. If the schema value changes after the execution plan is created, the plan will be invalidated at runtime and then re-created.
- When a rule or default is bound to the table or column. This is basically the same as altering a table. Binding a rule or default increments the schema column of *sysobjects* for the given table, invalidating the plans of any procedures that reference the table.
- When the table has been specifically identified, using **sp_recompile**, to force recompilation of any stored procedures referencing it. The system procedure **sp_recompile** increments the schema column of *sysobjects* for a given table. This invalidates any plans that reference the table, as described in the examples above. You do not specify a specific procedure to be recompiled—instead, you simply supply the name of a table to **sp_recompile** and all execution plans or procedures referencing the table will be invalidated. If you add a new index or update statistics on the data, it is a good idea to execute **sp_recompile** if you want the new options or information to be considered. (If it's practical to do so, you can simply restart SQL Server. All stored procedures will then get new execution plans the next time they are run.)
- When the stored procedure was created using the WITH RECOMPILE option. A stored procedure can be created using WITH RECOMPILE to ensure that its execution plan will be recompiled for every call and will never be reused. Using WITH RECOMPILE is not common, but it can be useful if the procedures take parameters and the values of the parameters differ widely, resulting in a need for different execution plans to be formulated. For example, if a procedure is passed a value to be matched in the WHERE clause of a query, the best way to carry out that query can depend on the value passed. SQL Server keeps a page of sample data for each index as a histogram to help it decide whether the index is selective enough to be useful.

For a given value, while an index might be highly selective, distribution statistics might indicate that only 5 percent of the rows have that value. Although the index would need to be visited, it would exclude many pages of data from having to be visited. Using the index would still be a good strategy. Using another value, the index might not be so selective. Rather than use only the index to visit most of the data pages (ultimately doing more I/O, since you're reading both the index and the data), you'd be better off simply scanning the data and not reading the index.

For example, suppose that I have an index on the *color* column of my *automobile* table. Forty percent of the cars are blue, 40 percent are red, and 5 percent each are yellow, orange, green, and purple. It is likely that a query based on color should table scan if the color being searched on is blue or red; but it should use the index for the other colors. Without using the WITH RECOMPILE option, the execution plan created and saved would be based on the color value the first time the procedure was executed. So if I passed *blue* to the procedure the first time it executed, I would get a plan that table scanned. Subsequently, if I pass *yellow,* I might be able to use the previous plan that was created for *blue.* In this case, however, I'd be doing a table scan, when I'd be better off using the index. In such a case, when there is a lot of variance in the distribution of data and execution plans are based on the parameters passed, it makes sense to use the WITH RECOMPILE option.

This example should also make it clear that two execution plans for the same procedure can be different. Suppose that I am not using WITH RECOMPILE, and I execute the procedure for both *blue* and *green* simultaneously from two different connections. Assume for a moment that no plan is cached (each will generate a new plan, but one plan will use the index and the other will not). When a subsequent request for *red* arrives, the plan it would use is a matter of chance. And if two simultaneous calls come in for *red* and each plan is available, the two equivalent requests will execute differently because they'll use different plans. If, as you're processing queries, you see significant deviations in the execution times of apparently identical procedures, think back to this example.

- When the stored procedure is executed using the WITH RECOMPILE option. This case is similar to the preceding one, except that here the procedure isn't created with the option, but rather the option is specified when the procedure is called. The WITH RECOMPILE option can always be added on execution, forcing a new execution plan to be generated. The new plan is then available for subsequent executions (not using WITH RECOMPILE). Executing using the WITH RECOMPILE option can be appropriate if there are significant deviations in parameters being passed or if you want a new plan to be generated because the histogram of distribution was updated (UPDATE STATISTICS) or new indexes that you think might help were added.

Storage of Stored Procedures

With each new stored procedure, a row is created in the *sysobjects* table, as occurs for all database objects. Information about the sequence tree that is used internally by SQL Server is stored in the *sysprocedures* table. The actual binary representation of the sequence tree is not exposed as part of the table—think of it as a hidden column of *sysprocedures*. The text of a stored procedure (including comments) is stored in *syscomments,* which is typically useful. This allows procedures like **sp_helptext** to display the source code of a stored procedure so that you can understand what's going on, and it allows stored procedure editors and debuggers to exist.

For most users and developers, the fact that the full text of a procedure is stored in English text is clearly a feature of SQL Server. But this text storage can also become the factor limiting how big a procedure can be. Prior to version 6.0, if you hit a size limit for your procedure, the execution plan was likely the cause. The execution plan was limited to 64 pages (128 KB, given 2-KB pages). Version 6.0 eliminated that limit, and as long as the procedure cache is big enough to hold one copy of the plan, it can execute. However, the size of a stored procedure might still be constrained due to the amount of text that can be stored in *syscomments* for a given stored procedure.

In the *syscomments* table, a throwback to the days before the *text* datatype existed, the text of procedures is kept in a column named *text*—but the *text* column is defined as *varchar(255)*. Any given procedure can have many rows in *syscomments,* with comment chunks in lengths of up to 255 characters, and those chunks are sequenced by the *colid* field. But *colid* was rather shortsightedly defined as a *tinyint* to ostensibly save a byte. Because the maximum value a *tinyint* can take is 255, up to 255 chunks of text can be included, each of which can be up to 255 bytes in size. Hence, the maximum size of the text used to create a stored procedure is 255 × 255, or 65,025 bytes (roughly 64 KB).

Remember that comments are also stored—so in a pinch, you can strip the comments from your procedures. A better solution, though, is to break up your stored procedure into multiple stored procedures, since procedures can call other procedures. Not only does this get you past the 64-KB text limit, but it makes your code more maintainable since smaller, more discreet modules are preferable to a single huge module.

Encrypting Stored Procedures

With the rollout of version 6.0, we learned somewhat painfully that some users did not appreciate as a feature the ability to store the text of stored procedures. Several independent software vendors (ISVs) had already built integrated solu-

tions or tools that created stored procedures to use with earlier versions of SQL Server. In most cases, these solutions were sophisticated applications, and the ISVs viewed the source code as their proprietary intellectual property. They had correctly noticed that the text of the procedure in *syscomments* didn't seem to do anything. If they set the text field to NULL, the procedure still ran fine—so that's what these ISVs did. In this way, they wouldn't be publishing their procedure source code with their applications. Unfortunately, when it came time to upgrade a database to version 6.0, this approach exposed a significant problem for their applications. The internal data structures for the sequence plans had changed between versions 4.2 and 6.0. ISVs had to re-create procedures, triggers, and views to generate the new structure. Although our Setup program was designed to do this automatically, it accomplished the tasks by simply extracting the text of the procedure from *syscomments* and then dropping and re-creating the procedure using the extracted text. It's no surprise that this automatic approach failed for procedures in which the creator had deleted the text.

The truth is that we were simply not aware of the number of ISV application designers who had deleted the text in this way. When we learned of this problem after we released our beta version, we immediately understood why developers had felt compelled to delete the text. Nonetheless, we could not undo the work that had already been done. Developers with these ISVs had to dig out their original DDL scripts and manually drop and re-create all their procedures. While perhaps possible, it wasn't really practical for us to create a converter program that would operate purely on the internal data structures used to represent procedures and views. Attempting this would have been like developing a utility to run against an executable program and have it backward engineer the precise source code (more than a disassembly) that was used to create the binary. The SQL source code compilation process is designed to be a descriptive process that produces the executable, not an equation that can be solved for either side.

Following the 6.0 beta release, but before the final release of version 6.0, we added the ability to *encrypt* the text stored in *syscomments* for stored procedures, triggers, and views. This allowed programmers to protect their source code, without making it impossible for our upgrade process to re-create stored procedures, triggers, and views in the future. You can now protect your source by simply adding the modifier WITH ENCRYPTION to CREATE. No decrypt function is exposed (which would defeat the purpose of hiding the textlike source code). Internally, SQL Server can read this encrypted text and upgrade the sequence trees when necessary. Because the text is not used at runtime, no performance penalty is associated with executing procedures created using WITH ENCRYPTION.

NOTE You give up some capabilities when you use WITH ENCRYPTION. For example, you can no longer use the **sp_helptext** stored procedure or object editors that display and edit the text of the stored procedure, and you cannot use a source-level debugger for Transact-SQL, like the one available in Microsoft Visual C++ Enterprise Edition. Unless you are concerned about someone seeing your procedures, you shouldn't use the WITH ENCRYPTION option.

If a procedure, trigger, or view is created using WITH ENCRYPTION, the *texttype* column of *syscomments* will have its third bit set to ON. (It will be OR'ed with decimal number 4.) For now, this simply means that the decimal value of *texttype* would be 6—the only other bit to be set would be the second one (decimal 2), indicating that the text in that procedure resulted from a CREATE statement and not a user-supplied comment. If you want to programmatically determine whether a procedure is encrypted, it is safer to check the value of the third bit by AND'ing it with 4 than it is to look for the value of 6. New bits could get added in the future, and the value of 6 might no longer be accurate.

To illustrate their effects on *syscomments*, two procedures—one encrypted and one not—are created in the following example:

```
CREATE PROCEDURE cleartext
AS
SELECT * FROM authors
GO

CREATE PROCEDURE hidetext WITH ENCRYPTION
AS
SELECT * FROM authors
GO

SELECT sysobjects.name, syscomments.* FROM syscomments, sysobjects
WHERE sysobjects.id=syscomments.id AND
(
sysobjects.id=OBJECT_ID('hidetext') OR
sysobjects.id=OBJECT_ID('cleartext')
)
```

Here's the output:

```
name       id         number  colid  texttype  language  text
---------  ---------  ------  -----  --------  --------  ----------------
cleartext  364528332  1       1      2         0         CREATE PROCEDURE
                                                        cleartext
                                                        AS
                                                        SELECT * FROM
                                                        authors
```

```
hidetext    380528389  1        1        6          0          Lýsåvù0Oéf:3j"ØË/
                                                                @âËAx+"Z%ÙX3ÔÏ2Då
                                                                &UÓ4ÏtRh]ÅÑn² ^
                                                                ·2Zò{'õÌÛvv,D"
```

To find created objects that have encrypted text, you can use a simple query:

```
-- Find the names and types of objects that have encrypted text
SELECT name, type FROM syscomments, sysobjects
WHERE sysobjects.id=syscomments.id
AND texttype & 4 > 0
```

Here's the output:

```
name       type
--------   ----
hidetext   P6
```

If you try to run **sp_helptext** against an encrypted procedure, it will return a message stating that the text is encrypted and cannot be displayed:

```
EXEC sp_helptext 'hidetext'
The object's comments have been encrypted.
```

Temporary Stored Procedures

Temporary stored procedures allow a sequence tree to be set up and an execution plan to be cached, but the object's existence, the sequence tree, and the text of the procedure are stored in the temporary database (*tempdb*) system tables—in *sysobjects, sysprocedures,* and *syscomments*. Recall that *tempdb* is re-created every time the server is restarted, so these objects do not exist after SQL Server is shut down. During a given SQL Server session, you can reuse the procedure without permanently storing it. If you are familiar with the PREPARE/EXECUTE model used by several other products, especially with the Embedded SQL programming paradigm, you'll know that temporary procedures use a similar model. The SQL Server ODBC driver, in fact, creates and executes temporary stored procedures when **SQLPrepare** and **SQLExecute** are performed.

Typically, you'll use a temporary stored procedure when you want to regularly execute the same task several times in a session, although you might use different parameter values, and you don't want to permanently store the task. You could conceivably use a permanent stored procedure and drop it when you are finished, but you'd inevitably run into cleanup issues if a stored procedure was still hanging around and the client application terminated without dropping the procedure. Because temporary stored procedures are deleted automatically when SQL server is shut down (and *tempdb* is created anew at startup), cleanup is not an issue. (And if you explicitly drop your temporary objects when you're finished with them, you might be able to keep *tempdb* at a smaller size.)

Just as SQL Server has three types of temporary tables, it also has three types of temporary stored procedures: *private, global,* and those created from direct use of *tempdb.*

Private temporary stored procedures

By adding a single pound sign (#) at the beginning of the stored procedure name (for example, *CREATE PROC #get_author AS...*), you can create the procedure from within any database as a private temporary stored procedure. Only the connection that created the procedure can execute it, and you cannot grant privileges on it to another connection. The procedure exists for the life of the creating connection only; that connection can explicitly use DROP PROCEDURE on it to clean up sooner. Because the scoping of a private temporary table is specific only to the connection that created it, you will not encounter a name collision should you choose a procedure name that's used by another connection. As with local variables, you use your private version and what occurs in other connections is irrelevant.

Global temporary stored procedures

By prefixing two pound signs (##) to the stored procedure name (for example, *CREATE PROC ##get_author AS...*), you can create the procedure from within any database as a global temporary stored procedure. Any connection can subsequently execute that procedure without EXECUTE permission being specifically granted. Unlike private temporary stored procedures, only one copy of a global temporary stored procedure exists for all connections. If another connection created a procedure with the same name, the two names will collide and the CREATE PROCEDURE statement will fail. A global temporary stored procedure exists until the creating connection terminates and all current execution of the procedure completes. Once the creating connection terminates, however, no further execution is allowed. Only those connections that have already started executing are allowed to finish.

Procedures created from direct use of *tempdb*

Realizing that *tempdb* is re-created every time SQL Server is started, you can create a procedure in *tempdb* that fully qualifies objects in other databases. Procedures created in *tempdb* in this way can exist even after the creating connection is terminated, and the creator can specifically grant and revoke execute permissions to specific users. To do this, the creator of the procedure must have CREATE PROCEDURE privileges in *tempdb*. Privileges in *tempdb* can be set up in one of two ways: you can set your privileges in *model* (the template database) so that they will be copied to *tempdb* when it is created at system restart, or you can set up an *autostart* procedure to set the *tempdb* privileges every time SQL Server is started. Here's an example of creating a procedure in *tempdb* and then executing it in the *pubs* database:

```
USE tempdb
GO

CREATE PROC testit AS
SELECT * FROM pubs.dbo.authors
GO

-- Executing the procedure created above from the pubs database
USE pubs
EXEC tempdb..testit
```

While we're on the subject of temporary objects, keep in mind that a private temporary table created within a stored procedure is not visible to the connection after the creating procedure completes. It is possible, however, to create a local temporary table before executing a stored procedure and make the table visible to the stored procedure. The scoping of the temporary table extends to the current statement block and all subordinate levels.

NOTE You can use the @@NESTLEVEL global variable to check for the visibility of temporary tables. A temporary table created at nest level 0 will be visible to all further levels on that connection. A table created within a procedure at nest level 1, for example, will not be visible when execution returns to the calling block at nest level 0. A global temporary table, or a table directly created in *tempdb* without using either # or ##, will be visible no matter what the nesting level.

System Stored Procedures and the Special *sp_* Prefix

SQL Server installs a large number of system stored procedures that are used mostly for administrative and informational purposes. In many cases, these are called behind the scenes by the SQL-DMO objects used by SQL Enterprise Manager and other applications. But the system stored procedures can also be called directly, and only a few years ago, doing so was the primary mechanism by which SQL Server was administered. Old-time SQL Server users (like me) were indoctrinated into using system stored procedures, and I confess that my primary administration tool for SQL Server remains using these system stored procedures directly, even though I recognize that the administration tools of today make things much easier. (I have made some headway into the modern era though. I now use the more graphical ISQLW.EXE instead of the character-based ISQL.EXE!) With the great tools and interfaces that are a core part of SQL Server today, there is not much reason to work with these system stored procedures directly anymore. But it's good to be familiar with them—understanding them can help you understand the operations that occur on the system tables and can take much of the mystery out of what's going on behind the scenes with the graphical tools.

All of the system stored procedure names begin with ***sp_***, and most exist in the *master* database. This is more than just a convention. A procedure created in the *master* database that begins with ***sp_*** is uniquely able to be called from any other database without the necessity of fully referencing the procedure with the database name. This can be useful for procedures you create as well. The ***sp_*** magic works even for *extended stored procedures,* which are user-written calls to dynamic link libraries (DLLs). By convention, extended stored procedure names begin with ***xp_***, but the ***sp_*** prefix and its special property can be applied to them as well (but only when added to the *master* database). In fact, some extended procedures that are supplied as part of the product, such as those used to create Automation objects (for example, **sp_OACreate**), use the ***sp_*** prefix so that they can be called from anywhere, although they are actually functions in a DLL, not a Transact-SQL stored procedure.

If you look carefully through the SQL Server system tables, you will find procedures beginning with ***sp_*** that are not among the documented system stored procedures. Typically, these procedures exist to be called by some other system stored procedure that is exposed; to support some SQL Server utility, such as SQL Enterprise Manager; or to provide statistics to the Windows NT Performance Monitor. These procedures are not documented for direct use because they exist only to support functionality exposed elsewhere—they do not provide that functionality independently. There is nothing secret about these procedures, and their text is exposed clearly in *syscomments*. You are welcome to explore them to see what they do and use them if you want. But unlike the documented stored procedures, maintaining system stored procedures or striving to make them exhibit exactly consistent behavior is not a commitment in future releases. Of course, if your applications were to become dependent on one of these procedures, you could certainly maintain your own version of it to perform exactly as you specify, or you could use one of them as a starting point and customize it to suit your needs (under a different name).

The *SQL Server Transact-SQL Reference* explains the specifics of each system stored procedure, so there is no need to restate those specifics here. I'll just categorize and enumerate most of them to give you a general understanding of the types and number of procedures that exist. The name of the procedure usually reveals its purpose. But first, I'll show you how to autostart stored procedures.

Autostart Stored Procedures

Version 6.0 introduced the handy ability to mark a stored procedure as *autostart*. Autostart stored procedures are useful if you regularly want to perform housekeeping functions or if you have a background daemon procedure that is expected always to be running. Another handy use for an autostart procedure is to have it assign some privileges in *tempdb*. Or the procedure can create a global tem-

porary table and then sleep indefinitely using WAITFOR. This will ensure that such a temporary table will always exist, because the calling process is the first thing executed and it never terminates.

It is simple to make a stored procedure start automatically—you use the system stored procedure **sp_makestartup** *procname* and pass the name of the procedure you want to start. You can remove the *autostart* attribute using **sp_unmakestartup**, or you can use **sp_helpstartup** to enumerate which procedures have been so marked. A procedure that is autostarted runs in the context of the SA (system administrator) account. (The procedure can use SETUSER to impersonate another account.) An autostarted procedure is launched asynchronously, and it can execute in a loop for the entire duration of the SQL Server process. This allows several such procedures to be launched simultaneously at startup. While a startup procedure is active, it consumes one of the configured user connections (from **sp_configure**).

A single startup procedure can nest calls to other stored procedures, consuming only a single user connection. Such execution of the nested procedures is synchronous, as would normally be the case. (That is, execution in the calling procedure does not proceed until the procedure being called completes.) Typically, a stored procedure that is autostarted will not generate a lot of output. Errors, including those raised with RAISERROR, will be written to the SQL Server error log, and any result sets generated will seemingly vanish. If you need the stored procedure to return result sets, you should use a stored procedure that calls the main stored procedure with INSERT/EXEC to insert the results into a table.

If you want to prevent a procedure marked as autostart from executing, you can start the server using trace flag 4022 or as a minimally configured server using the **-f** switch to SQLSERVR.EXE. (Add **-T4022** or **-f** as a parameter to SQL Server using the Setup program's Server Options dialog box.) These safeguards allow you to recover from problems. (Consider the perhaps absurd but illustrative example of someone including a procedure that executes the SHUTDOWN command. If such a procedure were marked for autostart, SQL Server would immediately shut itself down before you could do anything about it!)

The following sections discuss the broad categories for grouping stored procedures: System, Catalog, SQL Executive, Replication, and Extended.

System Stored Procedures

System stored procedures aid in the administration of your system, and they sometimes modify the system tables. You should not configure the system to allow direct modification of the system tables, since a mistake can render your database useless. That's why direct modification of system tables is prohibited by default. If modification is necessary, a system stored procedure is provided that

is known to do the job correctly. Below are the SQL Server system stored procedures. Each procedure's name gives you a clue as to its function.

sp_addalias
sp_addextendedproc
sp_addgroup
sp_addlanguage
sp_addlogin
sp_addmessage
sp_addremotelogin
sp_addsegment
sp_addserver
sp_addumpdevice
sp_adduser
sp_altermessage
sp_bindefault
sp_bindrule
sp_certify_removable
sp_change_users_login
sp_changedbowner
sp_changegroup
sp_check_removable
sp_coalesce_fragments
sp_configure
sp_create_removable
sp_dbinstall
sp_dboption
sp_dbremove
sp_db_upgrade
sp_defaultdb
sp_defaultlanguage
sp_depends
sp_devoption
sp_diskdefault
sp_dropalias
sp_droparticle
sp_dropdevice
sp_dropextendedproc
sp_dropgroup
sp_droplanguage
sp_droplogin
sp_dropremotelogin
sp_dropsegment
sp_dropserver
sp_droptype
sp_dropuser
sp_dropwebtask
sp_extendsegment
sp_fallback_activate_svr_db
sp_fallback_deactivate_svr_db
sp_fallback_enroll_svr_db
sp_fallback_help
sp_fallback_permanent_svr
sp_fallback_upd_dev_drive
sp_fallback_withdraw_svr_db
sp_help
sp_helpconstraint
sp_helpdb
sp_helpdevice
sp_helpextendedproc
sp_helpgroup
sp_helpindex
sp_helplanguage
sp_helplog
sp_helplogins
sp_helpremotelogin
sp_helprotect
sp_helpsegment
sp_helpserver
sp_helpsort
sp_helpsql
sp_helpstartup
sp_helptext
sp_helpuser
sp_lock
sp_lock2
sp_lockinfo
sp_logdevice
sp_makestartup
sp_makewebtask
sp_monitor
sp_objcheck
sp_objectsegment
sp_password
sp_placeobject
sp_recompile
sp_remoteoption
sp_rename
sp_renamedb
sp_runwebtask
sp_server_info
sp_serveroption
sp_setlangalias
sp_setnetname
sp_spaceused
sp_special_columns
sp_sproc_columns
sp_unbindefault
sp_unbindrule
sp_unmakestartup
sp_who
sp_who2

Catalog Stored Procedures

Applications and development tools commonly need access to information about table names, column types, datatypes, constraints, privileges, and configuration options. All this information is stored in the system tables (system catalogs). But system tables might require changes between releases to support new features, so your directly accessing the system tables could result in your application breaking from a new SQL Server release. For this reason, SQL Server provides catalog stored procedures, a series of stored procedures that extract the information from the system tables, providing an abstraction layer that insulates your application. If the system tables are changed, the stored procedures that extract and provide the information will also be changed to ensure that they operate consistently from an external perspective from one release to another. Many of these procedures also map nearly identically with an ODBC call. The SQL Server ODBC driver calls these procedures in response to those function calls. While it is fine to directly query the system catalogs for ad hoc use, if you are deploying an application that needs to get information from the system tables, use these catalog stored procedures:

sp_column_privileges
sp_columns
sp_databases
sp_datatype_info
sp_fkeys
sp_pkeys
sp_server_info
sp_special_columns
sp_sproc_columns
sp_statistics
sp_stored_procedures
sp_table_privileges
sp_tables

SQL Executive Stored Procedures

SQL Executive stored procedures are used by SQL Enterprise Manager to set up alerts and to schedule tasks for execution. If your application needs to carry out tasks like these, the following procedures can be called directly. They must be called from the *msdb* database.

Here are the alert stored procedures:

sp_addalert
sp_addnotification
sp_addoperator
sp_dropalert
sp_dropnotification
sp_dropoperator
sp_helpalert
sp_helpnotification
sp_helpoperator
sp_updatealert
sp_updatenotification
sp_updateoperator

And here are the scheduling stored procedures:

sp_addtask
sp_droptask
sp_helphistory
sp_helptask
sp_purgehistory
sp_updatetask

Replication Stored Procedures

Replication stored procedures are used to set up and manage publication and subscription tasks. SQL Enterprise Manager typically provides a front-end to these, but you can also call them directly. SQL Server has many replication stored procedures, and frankly it is hard to manually use replication with these procedures. (It *can* be done though, if you're bound and determined.) Everything SQL Enterprise Manager does (and makes it easy to do) ultimately uses these system stored procedures. Especially for replication, I urge you to use SQL Enterprise Manager or SQL-DMO if you need to customize replication administration into your application. Following are the replication stored procedures, by function. First, here are the server configuration and replication monitoring stored procedures:

sp_addpublisher
sp_addsubscriber
sp_changesubscriber
sp_dboption
sp_distcounters
sp_dropsubscriber
sp_helpdistributor
sp_helpserver
sp_helpsubscriberinfo
sp_MSkill_job
sp_replcleanup
sp_replcmds
sp_replcounters
sp_repldone
sp_replica
sp_replsync
sp_repltrans

Here are the publication stored procedures:

sp_articlecolumn
sp_changearticle
sp_changepublication
sp_droparticle
sp_droppublication
sp_enumfullsubscribers
sp_helppublication
sp_helppublicationsync
sp_helpreplicationdb
sp_replflush
sp_replstatus

And these are the subscription stored procedures:

sp_addsubscription
sp_changesubscription
sp_changesubstatus
sp_dropsubscription
sp_helpreplicationdb
sp_helpsubscription
sp_subscribe
sp_unsubscribe

Extended Stored Procedures

Extended stored procedures allow you to create your own external routines in a language such as C and have SQL Server automatically load and execute those routines just like a regular stored procedure. As you can with stored procedures, you can pass parameters to extended stored procedures and they can return results and/or return status. This allows you to extend the capabilities of SQL Server in powerful ways. Many features in the SQL Server product that have been introduced in the last couple of years have been implemented using extended stored procedures. These features include additions to SQL Enterprise Manager, the ability to send or receive e-mail messages, login integration with Windows NT domain security, and the ability to create a Web page based on a query.

Extended stored procedures are DLLs that SQL Server can dynamically load and execute. Extended stored procedures are not separate processes spawned by SQL Server—they run directly in the address space of SQL Server. The DLLs are created using the Open Data Services API, which SQL Server also uses.

Writing an extended stored procedure sounds harder than it really is, which is probably why these procedures are somewhat underused. But writing one can be as simple as writing a wrapper around a C function. For example, consider the formatting capabilities in SQL Server's PRINT statement, which are limited and do not allow parameter substitution. The C language provides the **sprintf** function, which is powerful for formatting a string buffer and includes parameter substitution. It is easy to wrap the C **sprintf** function and create an extended stored procedure that calls it, resulting in the procedure **xp_sprintf**. To show you how easy this is, below is the entire source code for the procedure **xp_sprintf**. Note that most of this code is setup code, and at the heart is the call to the C run-time function **sprintf()**:

```
// XP_SPRINTF
//
// Format and store a series of characters and values into an
// output string using sprintf
//
// Parameters:
//    srvproc - the handle to the client connection
//
// Returns:
//    XP_NOERROR or XP_ERROR
//
// Side Effects:
//
//
```

```
SRVRETCODE xp_sprintf( SRV_PROC * srvproc )
{
    int numparams;
    int paramtype;
    int i;
    char string [MAXSTRLEN];
    char format[MAXSTRLEN];
    char values[MAXARGUMENTS][MAXSTRLEN];
    char szBuffer[MAXSTRLEN];

    // Get number of parameters
    //
    numparams=srv_rpcparams(srvproc);

    // Check number of parameters
    //
    if (numparams < 3)
    {
        // Send error message and return


        //
        LoadString(hModule, IDS_ERROR_PARAM, szBuffer,
            sizeof(szBuffer));
        goto ErrorExit;
    }

    paramtype=srv_paramtype(srvproc, 1);
    if (paramtype != SRVVARCHAR)
    {
        // Send error message and return
        //
        LoadString(hModule, IDS_ERROR_PARAM_TYPE, szBuffer,
            sizeof(szBuffer));
        goto ErrorExit;
    }

    if (!srv_paramstatus(srvproc, 1))
    {
        // Send error message and return
        //
        LoadString(hModule, IDS_ERROR_PARAM_STATUS, szBuffer,
            sizeof(szBuffer));
        goto ErrorExit;
    }
```

```
for (i = 2; i <= numparams; i++)
{
    paramtype=srv_paramtype(srvproc, i);

    if (paramtype != SRVVARCHAR)
    {
        // Send error message and return
        //
        LoadString(hModule, IDS_ERROR_PARAM_TYPE, szBuffer,
            sizeof(szBuffer));
        goto ErrorExit;
    }
}

for (i=0; i < MAXARGUMENTS; i++)
{
    memset(values[i], 0, MAXSTRLEN);


    srv_bmove(srv_paramdata(srvproc, i + 3),
              values[i],
              srv_paramlen(srvproc, i + 3));
}

memset(string, 0, MAXSTRLEN);
srv_bmove(srv_paramdata(srvproc, 2), format,
    srv_paramlen(srvproc, 2));
format[srv_paramlen(srvproc, 2)]='\0';

// This is the heart of the function -- it simply wraps sprintf
// and passes back the string
sprintf(string, format,
    values[0],  values[1],  values[2],  values[3],  values[4],
    values[5],  values[6],  values[7],  values[8],  values[9],
    values[10], values[11], values[12], values[13], values[14],
    values[15], values[16], values[17], values[18], values[19],
    values[20], values[21], values[22], values[23], values[24],
    values[25], values[26], values[27], values[28], values[29],
    values[30], values[31], values[32], values[33], values[34],
    values[35], values[36], values[37], values[38], values[39],
    values[40], values[41], values[42], values[43], values[44],
    values[45], values[46], values[47], values[48], values[49]);

srv_paramset(srvproc, 1, string, strlen(string));

return XP_NOERROR;
```

```
ErrorExit:
    srv_sendmsg(srvproc,
                SRV_MSG_ERROR,
                SPRINTF_ERROR,
                SRV_INFO,
                (DBTINYINT) 0,
                NULL,
                0,
                0,
                szBuffer,
                SRV_NULLTERM);

    return XP_ERROR;
}
```

Because extended stored procedures run in the same address space as SQL Server, they can be efficient; however, although unlikely, a badly behaved extended stored procedure could theoretically crash SQL Server. A server crash would more likely result from someone's maliciousness rather than carelessness. But this is a definite area for concern, and you should understand the issues that I'll discuss in the rest of this section.

An extended stored procedure runs on the thread that called it. Each calling thread executes using the Windows NT structured exception handling constructs (most notably *try-except*). When a thread is badly written and does a bad thing, such as trying to reference memory outside its address space, it is terminated. But only that single connection is terminated, and SQL Server remains unaffected. Any resources held by the thread, such as locks, are automatically released.

In actual usage, I've seen that extended stored procedures do not introduce significant stability issues into the environment. Nonetheless, it certainly is theoretically possible for an extended stored procedure to twiddle some data structure within SQL Server (which it would have access to since the procedure is part of the SQL Server's address space) that could disrupt SQL Server's operation or conceivably even corrupt data. This could happen as a chance occurrence as a result of a bug in the extended stored procedure if you're unlucky, but it is more likely that the procedure would cause an access violation and have its thread terminated with no ill effects. A malicious procedure could conceivably cause data corruption, but such data structures are not exposed publicly so it would not be easy to write a malicious procedure. It is possible, however, and given the propensity of some social misfits to create viruses, I wouldn't rule out this problem (although I don't know of a single instance of this happening). The ultimate responsibility for protecting your data has to rest with your SA, who has control over which, if any, extended stored procedures can be added to the system.

Only the SA can register an extended stored procedure with the system (using **sp_addextendedproc**), and only the SA can grant others permission to execute the procedure. Extended stored procedures can be added only to the *master* database (eliminating their ability to be transferred simply to other systems via dump/load of databases, for example). The SA should allow use of only the procedures that have been thoroughly tested and proven to be safe and nondestructive. Ideally, the SA could also have access to the source code and build environment of the extended stored procedure to verify that it bears no malicious intent. (Some people have told me that they don't even want their SA to be able to do this—because the SA might not be trustworthy. If that's the case, you have bigger problems. If you can't trust your SA, you'd better get a new one.)

Even without extended stored procedures, the SA can disrupt a SQL Server environment in many ways. (Munging the system tables would be a good start.) Of course, you can decide that extended stored procedures will never be added to your system. That's certainly a safe approach, but you give up powerful capability by taking this route. (It's kind of like deciding never to ride in a car to avoid having an accident.) Even if you prohibit foreign extended stored procedures from your system, you should not go overboard and make this a sweeping rule that would prevent use of even the procedures provided by Microsoft to implement new features. Could one of these procedures have a bug that could disrupt SQL Server? Sure, but it's no more likely to occur than if the code for them had simply been statically linked into the SQLSERVR.EXE file rather than implemented as a DLL and loaded on demand. (Of course, the Microsoft procedures are thoroughly tested before their release. The chance of a catastrophic bug is pretty low.) The fact that these are *extended* stored procedures in no way increases the risk of bugs. It's an engineering decision, and a smart one, that allows additional features to be added to the product in a way that doesn't require extra change to the core product nor additional resource use by environments that don't call these features.

By convention, most of the extended stored procedures provided as part of the product begin with **xp_**. Unlike the **sp_** prefix, no special properties are associated with **xp_**. In fact, several extended stored procedures begin with ***sp_*** (for example, *EXEC* **sp_name**), which allows them to be called from any database without being fully qualified (for example, *EXEC master.dbo.***xp_name**). To ascertain whether a procedure is a regular stored procedure or an extended stored procedure, you shouldn't rely on the name's prefix. Instead, check the *type* column of *sysobjects,* which will show *P* for stored procedures or *X* for extended stored procedures.

As was the case with stored procedures, some extended stored procedures that are installed are not documented for direct use. These procedures exist to support functionality elsewhere, especially for SQL Enterprise Manager, SQL-DMO, and replication, rather than to provide features directly themselves.

Following are the extended stored procedures that are provided and documented for direct use. First, here are the general extended stored procedures:

xp_cmdshell

xp_sprintf

xp_sscanf

Here are the administration and monitoring extended stored procedures:

xp_logevent

xp_msver

xp_snmp_getstate

xp_snmp_raisetrap

xp_sqlinventory

xp_sqltrace

These are the integrated security related extended stored procedures:

xp_enumgroups

xp_grantlogin

xp_loginconfig

xp_logininfo

xp_revokelogin

And finally, the SQL mail-related extended stored procedures:

xp_deletemail

xp_findnextmsg

xp_readmail

xp_sendmail

xp_startmail

xp_stopmail

Triggers

A *trigger* is a special type of stored procedure that is fired on an event-driven basis rather than by a direct call. Here are some common uses for triggers:

- To maintain data integrity rules that extend beyond simple referential integrity
- To keep running totals updated
- To keep a computed column updated

- To implement a referential action, such as cascading deletes
- To maintain an audit record of changes
- To invoke an external action, such as begin a reorder process if inventory falls below a certain level or send e-mail or a pager notification to someone who needs to perform an action because of data changes

A trigger can be set up to fire when data is changed in some way—that is, via an INSERT, an UPDATE, or a DELETE statement. Only one trigger can be defined for each event, although a trigger can invoke many stored procedures and different actions can be specified by evaluating the values of a given column of data. While it is true that a table is "limited" to three triggers, because of the ability to call an almost unlimited number of stored procedures, this should not really be a limitation in any way. (The nesting depth limit is still 16, however.) In fact, some might consider having the three triggers a feature because it forces you to properly think through and specify the ordering of the actions. Rather than simply defining, say, five stored procedures that should fire given a data change, this "limitation" also ensures that the order of trigger firing is exactly specified, which, of course, can greatly affect the results of the trigger.

A single trigger can be created to execute for any or all of the INSERT, UPDATE, and DELETE actions, which modify data. Currently, SQL Server offers no trigger on a SELECT statement, since SELECT does not modify data. In addition, triggers can exist only on base tables, not on views. (Of course, data modified on a view does cause a trigger on the underlying base table to fire.)

A trigger is executed once for each UPDATE, INSERT, or DELETE statement, regardless of the number of rows it affects. Although it is sometimes thought that a trigger is executed once per row or once per transaction, neither of these assumptions is correct, strictly speaking. However, if a statement affects only one row or is a transaction unto itself, the trigger will exhibit the *characteristics* of per-row or per-transaction execution. For example, if a WHILE loop were set up to perform an UPDATE statement repeatedly, an update trigger would execute each time the UPDATE statement was executed in the loop.

A trigger fires after the data modification statement has performed its work but before that work is committed to the database. Both the statement and any modifications made in the trigger are implicitly a transaction (whether or not an explicit BEGIN TRANSACTION was declared). Therefore, the trigger can roll back the work. A trigger has access to the before image and after image of the data via the special pseudotables *inserted* and *deleted*. These two tables have the same set of columns as the underlying table being changed. You can check the before and after values of specific columns and take action depending on what you encounter. These tables are not physical structures—SQL Server constructs them

from the transaction log. This is why an unlogged operation such as a bulk copy or SELECT INTO does not cause triggers to fire. For regular logged operations, a trigger will always fire if it exists. A trigger cannot be circumvented (short of dropping it).

The *inserted* and *deleted* pseudotables cannot be modified directly because they don't actually exist. As I mentioned earlier, the data from these tables can be queried only. The data they appear to contain is based entirely on modifications made to data in an actual, underlying base table. The *inserted* and *deleted* pseudotables will contain as many rows as the INSERT, UPDATE, or DELETE statement affected. Sometimes it is necessary to work on a row-by-row basis within the pseudotables, although, as usual, a set-based operation is generally preferable to row-by-row operations. You can perform row-by-row operations by executing the underlying INSERT, UPDATE, or DELETE in a loop so that any single execution affects only one row, or you can perform the operations by opening a cursor on one of the *inserted* or *deleted* tables within the trigger. The need to reconstruct the *inserted* and *deleted* pseudotables from the log is the primary reason why an update on a table having an update trigger always needs to generate delete and insert log records—and why update-in-place is not possible when an update trigger exists.

Rolling Back a Trigger

Executing a ROLLBACK from within a trigger is different from executing a ROLLBACK from within a nested stored procedure. In a nested stored procedure, a ROLLBACK will cause the outermost transaction to abort, but the flow of control continues. However, if a trigger results in a ROLLBACK (whether because of a fatal error or from an explicit ROLLBACK command), the entire batch is aborted.

NOTE In the final release of version 6.5, a ROLLBACK in a trigger did not abort the batch and flow of control continued. This was due to a bug, not an intentional change. You could argue that the "buggy" behavior is more desirable than the expected behavior, but the expected behavior was added back in Service Pack 1 to provide backward compatibility. In Service Pack 1 and all subsequent service packs, a rollback in the transaction aborts the batch as it always did in releases earlier than version 6.5.

Suppose that the following pseudocode batch is issued from ISQL.EXE:

```
begin tran
delete....
update....
insert.... -- This starts some chain of events that fires a trigger
           -- that rolls back the current transaction
```

```
update.... -- Execution never gets to here - entire batch is
           -- aborted because of the rollback in the trigger
if....commit  -- Neither this statement nor any of the following
               -- will be executed
else....rollback
begin tran....
insert....
if....commit
else....rollback

GO            -- isql batch terminator only

select ...    -- Next statement that will be executed is here
```

As you can see, once the trigger in the first INSERT statement aborts the batch, SQL Server not only rolls back the first transaction but skips the second transaction completely and continues execution following the GO.

Misconceptions about triggers include the belief that the trigger cannot do a SELECT statement that returns rows and that it cannot execute a PRINT statement. Although you can use SELECT and PRINT in a trigger, doing these operations is usually dangerous practice unless you control all the applications that will work with the table that includes the trigger. Otherwise, applications not written to expect a result set or a print message following a change in data might fail because that unexpected behavior occurs anyway.

Be aware that if a trigger modifies data on the same table on which the trigger exists, that trigger does not fire again. (This could easily lead to an infinite loop.) However, if separate triggers exist for INSERT, UPDATE, and DELETE statements, one trigger on a table could cause a different trigger on the same table to fire (but only if **sp_configuration 'nested triggers'** is set to 1, as I'll discuss in a moment).

A trigger can also modify data on some other table. If that other table has a trigger, whether or not that trigger also fires depends on the current **sp_configuration** value for the **nested triggers** option. If that option is set to 1 (TRUE), which is the default setting, triggers will cascade to a maximum chain of 16. If an operation would cause more than 16 triggers to fire, the batch will be aborted and any transaction will be rolled back. This prevents an infinite cycle from being encountered. If your operation is hitting the limit of 16 firing triggers, you should probably look at your design—you've reached a point at which there are no longer any simple operations, so you're probably not going to be ecstatic with the performance of your system. If your operation truly is so complex that you need to perform further operations on 16 or more tables to modify any data, you could call stored procedures to perform the actions directly rather than enabling and using cascading triggers. Although valuable, overused cascading triggers can make your system a nightmare to maintain.

Debugging Stored Procedures and Triggers

By now, it should be obvious that stored procedures and triggers can represent a significant portion of your application's code. Even so, for a long time, no decent debugger support existed for stored procedures. (The typical debugging tool was the liberal use of PRINT statements.) Some ISVs jumped in and helped by adding some "pseudo-debugging" products—but these products didn't have access to the actual SQL Server execution environment. They typically added debugging support by doing tricks behind the scenes, such as adding additional PRINT statements or adding SELECT statements to get the current values of variables. Although some of these products were helpful, they had significant limitations: typically, they couldn't step into nested stored procedures or into triggers. A Transact-SQL debugger was always prominent on the wish list of SQL Server customers. I know firsthand how frustrating it was not to have a proper debugger, and I've spent many late nights putting PRINT statements into stored procedures.

Still, it didn't make sense for us to write a new SQL Server–specific debugger because there were already too many debuggers on the market. If you're like most programmers, you want to do your work in one development environment. For example, if you are a C programmer, you probably want to use the same debugger on your SQL code that you use on your C code. Or if you program in Microsoft Visual Basic, you probably want to use the Visual Basic development environment for debugging. Fortunately, this environment-specific debugging capability now exists and its availability is rapidly expanding. If you are a developer in Microsoft Visual C++, Microsoft Visual J++, or Visual Basic, you can now debug Transact-SQL using the same debugger you use in those environments.

To accomplish this, we defined a DLL and a set of callbacks that SQL Server 6.5 would load and call at the beginning of each SQL statement. In essence, we defined a set of debug events that would allow us to control the execution on a statement-by-statement basis within SQL Server. This has come to be known as the SQL Server Debug Interface, or SDI. The interface that shipped with the version 6.5 release was a work in progress. SDI's first customer was Visual C++ version 4.2. SQL Server 6.5 shipped several months before Visual C++ 4.2, and, of course, we didn't get the interface quite right. Specs are never perfect, and they nearly always get tweaked—at a minimum—during implementation. This was certainly true for SDI, so to use it with Visual C++ version 4.2 and later you need SQL Server version 6.5 with Service Pack 1 or later. We debugged the debugging interface using the Visual C++ team as guinea pigs, and now other development tools are also adding support for SDI. By the time you read this, I expect SDI to be available with Visual C++, Visual Basic, Visual J++, and other development tools from Microsoft. (The exact packaging is always subject to change, but in general, the debugger support for Transact-SQL is available only in each product's Enterprise Edition.)

Although the interface is quite likely to change in future releases, it is also made available via a technical note to ISVs that want to add SQL Server debugger support. SDI is a specialized interface that is of interest only to those writing debuggers, so it is not considered a general feature of SQL Server.

With the existence of SDI, and using the Microsoft Developer Studio debugging environment of Visual C++ Enterprise Edition, you now have a real debugging environment for Transact-SQL. For example, as a C/C++ developer using Developer Studio (Visual C++ Enterprise Edition), you can:

- Do line-by-line debugging of all your Transact-SQL code.
- Step directly from your C code executing on your client machine into the Transact-SQL code executing remotely at the SQL Server machine.
- Remotely debug your procedures, with the actual execution of the procedures happening at the SQL Server machine and your debugging environment happening locally. (Or if you prefer, you can do it all from one machine.)
- Set breakpoints anywhere in your SQL code.
- Watch the contents of SQL local and global variables. You can even watch global variables that are not used in your SQL code: for example, you can watch current status codes using @@ERROR or the number of rows selected using @@ROWCOUNT.
- Modify the values of most variables in a watch window, testing conditional logic in your code more easily. (Note that variables with datatypes for which there is no direct mapping in C cannot be edited in a watch window.)
- Examine the values of parameters passed to stored procedures.
- Step into or over nested procedures. And if a statement causes a trigger to fire, you can even step into the trigger.
- Use Microsoft Developer Studio to edit your procedures and save them to the server. This makes it easy to fix bugs on the fly. SQL keywords and comments in your code are color coded, as they would be in C, to make them easier to spot.
- Optionally send results of your SQL statements to the result window directly in Developer Studio.

The SDI is implemented via the pseudo–extended stored procedure **sp_sdidebug**. (The *sp_* convention was used so that the procedure could be called from any database without being fully qualified.) By *pseudo,* I mean that, like a normal

extended stored procedure, you will see an entry in the *sysobjects* table of type *X* for **sp_sdidebug**. But unlike a normal extended stored procedure, the code for **sp_sdidebug** is internal to the SQL Server and does not reside in a separate DLL. This is true for a few other procedures as well, such as the remote cursor calls made by ODBC and DB-Library cursor functions. This was done so that we could add a new capability to the server without having to change the tabular data stream (TDS) protocol that describes result sets back to the client application. It also eliminates the need for new keywords (potentially breaking a few applications) when the commands are of the sort that would not be executed directly by an application anyway.

You should never call **sp_sdidebug** directly. The procedure exists to load a DLL that the provider of the debugger would write and to toggle debugging on and off for the specific SQL Server connection being debugged. The debug DLL for Visual C++ is SQLSDI.DLL. When debugging is enabled, SQLSDI.DLL is given access to internal state information for the SQL Server connection being debugged. All of the APIs defined in the interface are synchronous calls, and they are called in the context of the thread associated with the connection, which allows for callbacks to SQL Server to occur in the context of the client's thread. The internal Process Status Structure (PSS) holds status information for the connection being debugged, and the DLL is then able to read this structure to determine local variable, parameter, global variable, and symbol information.

The debugging support in Visual C++ Enterprise Edition seems pretty normal if you are already familiar with the environment. Typically, the biggest problem people have with debugging Transact-SQL is getting it configured in the first place. Here are some tips that might help to you in debugging Transact-SQL from Developer Studio:

- You must use the Enterprise Edition of Visual C++ 4.2 or later, not the Professional or Standard edition. You must have run Setup from Visual C++ to install the SQL Server debugging components. Setup also installs SQL Server version 6.5 Service Pack 1 (SP1), and this version or a later one is also required for debugging support.
- You must use the SP1 or later components of DB-Library and/or the SQL Server ODBC driver to be able to step from your C++ or Java source code to stored procedure code and back. For DB-Library, you need version 6.50.212 or later; for the ODBC driver, you need version 2.50.0212 or later.
- I recommend that you use Windows NT 4.0. If you use Windows NT 3.51, you must apply Windows NT 3.51 Service Pack 4. You can debug from Windows 95 as well, but you must use remote debugging because SQL Server does not currently run on Windows 95.

- Run SQL Server under a user account, not as Local System. (You can change this in the Services applet of the Windows NT Control Panel.) If SQL Server is running under the local account, breakpoints are ignored. When debugging on a machine also running SQL Server, you should run SQL Server under the same user context used for running the debugger. (Make sure that you can run SQL Server from the command line rather than as a service—for example, *C:\MSSQL\BINN\SQLSERVER.EXE -c.*)
- Extended error information regarding debugging can be written to the Windows NT event log. The events are written to the application log under MSDEVSDI. For example, *Event ID 11,* which relates directly to the previous tip (running SQL Server under a user account), will be written there:

```
Event ID #11: SQL Server when started as service must not log
on as System Account. Reset to logon as user account using
Control Panel.
```

- SQL Server Debugging must be enabled in Developer Studio. To enable this option in Developer Studio 97, from the Tools menu, select Options. In the Options dialog box, click on the Data View tab, and then click the SQL Server Debugging check box.
- Text, numeric/decimal, and float datatypes cannot be edited in a watch window.
- Do not debug on a production server. Due to the added overhead and break-in nature of the debugging product, you could adversely affect other users.
- String and text values larger than 255 bytes are shown as NULL in the watch window.
- Right-click the mouse, and then click the Refresh option to obtain object changes in DDL from other clients.
- If you delete all characters from a string in the watch window, the value will show NULL. If the variable does not allow NULL, the next step operation will reset the value to its previous value.
- Only the first 64 bytes of a text column are displayed in the output window, even if more data is contained in the actual column.

For simple debugging, you might find yourself still using PRINT; for tougher problems, you might come to regard the new debugging capability as a lifesaver. I demonstrated this capability at the SQL Server Professional Developer Conference

in September 1996, and this 5-minute sidebar of my 2-hour presentation seemed to generate more interest than anything else I covered!

Execute("*any string*")

The ability to formulate and then execute a string dynamically in SQL Server is a subtle but powerful capability. Using this capability, you can avoid additional round-trips to the client application by formulating a new statement to be executed directly on the server. You can pass a string directly, or you can pass a local variable of type *char* or *varchar*. This capability is especially useful if you need to pass an object name to a procedure or if you want to build an SQL statement from the result of another SQL statement. For example, suppose that I had partitioned my database to have multiple tables similar to the *authors* table. I could write a procedure like the one shown below to pass the name of the table I want to insert into. The procedure would then formulate the INSERT statement by concatenating strings, and then it would execute the string it formulated:

```
CREATE PROC add_author
@au_id char(11),
@au_lname varchar(20),
@au_fname varchar(20),
@tabname varchar(30) AS

BEGIN
DECLARE @insert_stmt varchar(255)
SELECT @insert_stmt="INSERT " + @tabname + " (au_id,
    au_lname, au_fname, contract) VALUES ('" + @au_id +
    "','" + @au_lname + "','" + @au_fname + "', 1)"
EXECUTE (@insert_stmt)
END

EXEC add_author '999-99-1234', 'Soukup', 'Ron', 'authors'
```

Working with Text and Image Data

SQL Server provides binary large object (BLOB) support via the *text* and *image* datatypes. If you work with these datatypes, you might want to use the additional statements provided by SQL Server along with the standard SELECT, INSERT, UPDATE, and DELETE statements. Because a single text column can be as large as 2 GB, you frequently need to work with text data in chunks, and these additional statements (which I'll discuss in a moment) can help. (I might have discussed this topic earlier, when I discussed Transact-SQL programming. However, because you need some knowledge of isolation levels, transactions, and consistency issues to understand this topic, I decided to wait until after I had covered those issues.)

For simplicity's sake, I'll frame this discussion mostly in terms of the *text* datatype. But everything here is also relevant to the *image* datatype. These two datatypes are essentially the same internally. Recall that *text* and *image* datatypes are unique in that they are not stored on the same data page as the rest of the row. Instead, a pointer to a separate chain of pages for the text/image data is stored in the row. A separate chain of pages exists for each *text* (or *image*) column, and these pages are not shared when several such columns are present. This means that an entire 2-KB page must be used to store the first single byte of data plus the 16-byte text pointer that is written on the data page. If another row with 1 byte of text were added, another entire 2-KB page would be needed to store the data. An initially NULL text column (occurring either by omission of the column in the INSERT statement or by specifying NULL in the VALUES clause) does not require an entire page for storage, so until data is written to the *text* column, no storage for it is consumed. Although you can think of *text* and *image* as variable-length datatypes, their storage size is a step function. The effective storage size can be 0 bytes if the value is implicitly NULL, but then storage increases in 2-KB increments as each new page is required. (Plus, when the column is not NULL, an additional 16 bytes is required for the text pointer. About 1800 bytes of data can actually be stored per page.)

Clearly, the space required by *text* and *image* for small amounts of data is inefficient. But there are also functional drawbacks. Although you can indeed use standard INSERT, UPDATE, DELETE, and SELECT statements with a *text* or *image* column, some significant restrictions apply. In a WHERE clause, you can search on the *text* column only with the LIKE operator or with a function such as PATINDEX(). *Text* and *image* variables cannot be manipulated. You can declare a parameter in a stored procedure to be of type *text* or *image,* but you can't do much besides pass a value to the procedure initially. For example, you cannot subsequently assign different values to the parameter. Because of the space usage and functional drawbacks, you'll want to use *text* or *image* only when another datatype isn't a reasonable option. If a *varchar(255)* column can work for you, you can use it and avoid *text* altogether. But if you absolutely need a memo field, for example, and 255 characters are not enough, you'll need to use *text* (or denormalize and use multiple *varchar* columns).

If a *text* column makes the most sense for you despite its drawbacks, you need to understand how to work effectively with text. When you can, it is easiest to work with *text/image* datatypes using standard SELECT, INSERT, UPDATE, and DELETE statements. But if your text data gets large, you're going to run into issues, such as how big a string your application can pass, that might make it necessary for you to deal with chunks of data at a time instead of the entire column.

The special statements for working with text data are WRITETEXT, READTEXT, and UPDATETEXT. Both READTEXT and UPDATETEXT let you work with chunks of a *text* column at a time. The WRITETEXT statement does not let you deal with

chunks but rather with the entire column only. WRITETEXT and UPDATETEXT will not log the text operations by default, although they can be instructed to log them. (The database must have the **select into/bulkcopy** option enabled for nonlogged operations.) An INSERT, UPDATE, or DELETE statement will always be logged, but these special text statements can be run without logging.

I don't encourage nonlogged operations for general use because they can compromise your database backup strategy. The situation is similar to nonlogged bulk copy. Nonlogged operations cannot be recovered at startup. A terminated nonlogged operation will leave the database in the state it was in before the operation began because logging (and hence rollback) of extent allocations still occurs. But the biggest downside to this is that in the face of a failure after a nonlogged operation, your database is only as good as your last full backup and transaction dumps up to the issuance of the nonlogged operation. You can't do further transaction dumps after a nonlogged operation is performed. So think carefully about the appropriateness of nonlogged text and image operations. Also, if you use SQL Server replication to replicate *text* or *image* columns, the operations *must* be logged because the replication process looks for changes based on the transaction log.

The WRITETEXT, READTEXT, and UPDATETEXT statements all work with a *text pointer*. A text pointer is a unique *varbinary(16)* value for each *text* or *image* column of each row.

WRITETEXT

WRITETEXT completely overwrites an existing *text* or *image* column. You provide the column name (qualified by the table name), the text pointer for the specific column of a specific row, and the actual data to be written. The WITH LOG clause is optional, although I will always use it in the examples presented here. It might seem like a catch-22 when using WRITETEXT immediately, because you need to pass it a text pointer—but if the column is initially NULL, there *is* no text pointer. So how do you get one? You SELECT it with the TEXTPTR() function. But if the *text* column has not been initialized, the TEXTPTR() function returns NULL. To initialize a text pointer for a column of a row with *text* or *image* data, you can do some variation of the following:

- Explicitly insert a non-null value in the *text* column when you use an INSERT statement. Recognize that WRITETEXT will completely overwrite the column anyway, so the value can always be something like *A* or a blank space.
- Define a default on the column with a non-null value like *A*. Then when you do the insert, you can specify DEFAULT or omit the column, which will result in the default value being inserted and the text pointer being initialized.

- Explicitly update the row after inserting it, and then set the column to NULL (or to anything else).

No matter how you initialize the text pointer, as soon as you do it, at least one data page will be consumed for the column of that row, even if you initialize the value to NULL or to a single character.

You then select the text pointer into a variable declared as *varbinary(16)* and pass that to WRITETEXT. You can't use SELECT statements or expressions in the WRITETEXT statement. This means that the statement is limited to being executed one row at a time (although it can be done from within a cursor). The SELECT statement that gets the text pointer should be known to return only one row, preferably by using an exact match on the primary key value in the WHERE clause, because that will ensure that at most one row can meet the criteria. You can, of course, use @@ROWCOUNT to check this if you are not absolutely sure that the SELECT statement can return only one row. Before using the WRITETEXT statement, you should also ensure that you have a valid text pointer. If you find a row with the criteria you specified and the text pointer for that row was initialized, it will be valid. You can check it as a separate statement using the TEXTVALID() function. Or you can check that you do not have a NULL value in your variable that was assigned the text pointer, as I'll show in the following example. Make sure that you don't have an old text pointer value from a previous use, which would make the IS NOT NULL check be TRUE. In this example, I do one variable assignment and the variable starts out NULL, so I am sure that a non-null value means I have selected a valid text pointer:

```
-- WRITETEXT with an unprotected text pointer
DECLARE @mytextptr varbinary(16)
SELECT @mytextptr=TEXTPTR(pr_info)
    FROM pub_info WHERE pub_id='9999'
IF @mytextptr IS NOT NULL
    WRITETEXT pub_info.pr_info @mytextptr WITH LOG 'Hello Again'
```

In this example, the text pointer is not protected from changes made by others. Therefore, it is possible that the text pointer will no longer be valid by the time the WRITETEXT operation is performed. Suppose that you get a text pointer for the row with *pub_id='9999'*. But before you use it with WRITETEXT, another user deletes and reinserts the row for publisher 9999. In that case, the text pointer you are holding will no longer be valid. In the example above, the window for this happening is small, since I do the WRITETEXT immediately after getting the text pointer. *But there is still a window.* In your application, the window may be wider. If the text pointer is not valid when you do the WRITETEXT operation, you will get an error message like this:

```
Msg 7123, Level 16, State 1
Invalid text pointer value 000000000253f380.
```

You can easily see this for yourself if you add a delay (for example, *WAITFOR DELAY "00:00:15"*) after getting the text pointer and then delete the row from another connection. You'll get error 7123 when the WRITETEXT operation executes. If you think the chances of getting this error are slim, you can choose to simply deal with the error when and if it occurs. Frankly, because this seems to be what most applications that use text do, *text* columns are used in mostly low-concurrency environments. But even so, I don't think it's good practice. (More likely, this is the general usage because we haven't sufficiently explained the concurrency issue.)

I recommend that you instead use transaction protection to ensure that the text pointer will not change from the time you read it until you use it, and to serialize access for updates so that you do not encounter frequent deadlocks. Many applications use TEXTVALID() to check right before operating—that's the right idea, but it's hardly foolproof. There is still a window between the TEXTVALID() operation and the use of the text pointer, during which the text pointer may be invalidated. The only way to close the window is to make both operations part of an atomic operation. This means using a transaction and having SQL Server protect the transaction with a lock. (For more about locking, see Chapter 13. You might want to read that chapter and then return to this section.)

By default, SQL Server will operate with Read Committed isolation and release a share (READ) lock after the page has been read. So simply putting the pointer in a transaction with the READ COMMITTED isolation level, which is SQL Server's default, is not enough. You need to ensure that the lock is held until the text pointer is used. You could change the isolation level to Repeatable Read, which is not a bad solution, but this changes the isolation behavior for all operations on that connection and so it might have a more widespread effect than you intend. (Although you could, of course, then change it right back.) But even this is not ideal. This approach doesn't guarantee that you will subsequently be able to get the exclusive lock required to do the WRITETEXT operation; it ensures only that when you get to the WRITETEXT operation, the text pointer will still be valid. You won't be sure that you're not in the lock queue behind another connection waiting to update the same row and column. In that case, your transaction and the competing one would both hold a share lock on the same page and would both need to acquire an exclusive lock. Since both transactions are holding a share lock, neither can get the exclusive lock, and a *deadlock* results in one of the connections having its transaction automatically aborted. (In Chapter 13, you'll see that this is an example of a *conversion deadlock*.) If multiple processes are intending to modify the text and all transactions first request an update lock in a transaction when selecting the text pointer, conversion deadlocks will be avoided because only one process will get the update lock and the others will queue for it. But those users' transactions that need only to read the page will not be affected, since an update lock and a share lock are compatible.

Using the update lock on the text pointer is good for serializing access to the actual text pages, even though the lock on the text page is distinct from the lock or the text pointer. In this case, you essentially use the update lock on a text pointer as you'd use an intent lock for the text page. This is conceptually similar to the intent locks that SQL Server uses on a table when a page-locking operation for that table will take place. That operation recognizes that pages and tables have an implicit hierarchy. You can think of text pointers and text pages as having a similar hierarchy and use the update lock on the text pointer to protect access to the associated text pages. Following is the improved version that protects the text pointer from getting invalidated and also reserves my transaction's spot in the queue so that it will get the exclusive lock that's necessary to change the column. This approach will avoid conversion deadlocks on the text pages:

```
-- WRITETEXT with a properly protected text pointer
BEGIN TRAN
DECLARE @mytextptr varbinary(16)
SELECT @mytextptr=TEXTPTR(pr_info)
    FROM pub_info (UPDLOCK) WHERE pub_id='9999'
IF @mytextptr IS NOT NULL
    WRITETEXT pub_info.pr_info @mytextptr WITH LOG 'Hello Again'
COMMIT TRAN
```

READTEXT

READTEXT is used in a similar way to WRITETEXT, except that READTEXT allows you to specify a starting position and the number of bytes to read. Here is its basic syntax:

```
READTEXT [[database.]owner.]table_name.column_name
    text_ptr offset size [HOLDLOCK]
```

Unlike with WRITETEXT, with READTEXT I do not need to work with the entire contents of the data. I can specify the starting position (*offset*) and the number of bytes to read (*size*). READTEXT is often used with the PATINDEX() function to find the offset at which some string or pattern exists, and it's also used with DATALENGTH() to determine the total size of the text column. But these functions cannot be used as the offset parameter directly. Instead, you must execute them beforehand and keep their values in a local variable, which you then pass. As mentioned in the discussion of WRITETEXT, you'll want to protect your text pointer from becoming invalidated. In the next example, you'll read text without updating it. So you can use the HOLDLOCK lock hint on the SELECT statement for the text pointer (or set the isolation level to Reapeatable Read).

Sometimes people think that transactions are used only for data modifications, but notice that in this case you use a transaction to ensure read repeatability (of the text pointer) even though you are not updating anything. You can optionally

add HOLDLOCK to the READTEXT statement to ensure that the text doesn't change until the transaction has completed. But in the example below, I read the entire contents with just one read and I will not be rereading the contents, so there is no point in using HOLDLOCK here. This example finds the pattern *Washington* in the *pr_info* column for *pub_id* 0877 and returns the contents of that column from that point on:

```
-- READTEXT with a protected text pointer
BEGIN TRAN
DECLARE @mytextptr varbinary(16), @sizeneeded int, @pat_offset int
SELECT @mytextptr=TEXTPTR(pr_info),
    @pat_offset=PATINDEX('%Washington%',pr_info) - 1,
    @sizeneeded=DATALENGTH(pr_info) -
        PATINDEX('%Washington%',pr_info) - 1
    FROM pub_info (HOLDLOCK) WHERE pub_id='0877'

IF @mytextptr IS NOT NULL AND @pat_offset >= 0 AND
    @sizeneeded IS NOT NULL
    READTEXT  pub_info.pr_info @mytextptr @pat_offset @sizeneeded

COMMIT TRAN
```

The offset returned by PATINDEX() and the offset used by READTEXT unfortunately are not consistent. READINDEX treats the first character as offset *0*. (This makes sense to me since I think of an offset as how many characters you have to move to get to the desired position—to get to the first character, you don't need to move at all.) But PATINDEX() returns the value in terms of position, not really as an offset, and so the first character for it would be *1*. (I'd call this a minor bug. But people have adapted to it, so changing it would cause more problems than it would solve at this point. You should assume that this acts as intended and adjust for it.) You need to fix this discrepancy by taking the result of PATINDEX() and subtracting *1* from it. PATINDEX() returns *–1* if the pattern is not found. Since I subtract *1* from the value returned by patindex(), if the pattern were not found, the variable @pat_offset would be *–2*. I simply check that @pat_offset is not negative.

You also need to specify how many bytes you want to read. If you want to read from that point to the end of the column, for example, you can take the total length as returned from DATALENGTH() and subtract the starting position, as shown in the example above. You cannot simply specify a buffer that you know is large enough to read the rest of the column into; that will result in error 7124:

```
The offset and length specified in the READTEXT command is greater
than the actual data length of %d.
```

If you could always perform a single READTEXT to handle your data, you'd probably not use it; instead, you could use SELECT. You need to use READTEXT

when a *text* column is too long to reasonably bring back with just one statement. For example, the text size of the *pr_info* field for publisher 1622 is 18,518 bytes. I can't select this value in a program like ISQL/w because it's longer than the maximum expected row length for a result set, so it would be truncated. But I can set up a simple loop to show the text in pieces. To understand this process, you need to be aware of the global variable @@TEXTSIZE, which is the maximum amount of text or image data that you can retrieve in a single statement. (Of course, you need to make sure that the buffer in your application will also be large enough to accept the text.) You can read chunks smaller than the @@TEXTSIZE limit, but not larger.

You can change the value of @@TEXTSIZE for your connection by using SET TEXTSIZE *n*. The default value for @@TEXTSIZE is 64 KB. I suggest that you read chunks whose size is based on the amount of space available in your application buffer and based on the network packet size so that the text will fit in one packet, with an allowance for some additional space for metadata. For example, with the default network packet size of 4192 bytes (4 KB), a good read size would be about 4100 bytes, assuming that your application could deal with that size. You should also be sure that @@TEXTSIZE is at least equal to your read size. You can either check to determine its size or explicitly set it as I do in the example below. Also notice the handy use of the CASE statement for a variable assignment to initialize the @readsize variable to the smaller of the total length of the column and the value of @@TEXTSIZE. (In this example, I make my read size only 100 characters so that it displays easily in an ISQL/w or in a similar query window. But this is too small for most applications and is used here for illustration only.)

```
-- READTEXT in a loop to read chunks of text
-- Instead of using HOLDLOCK, use SET TRANSACTION ISOLATION LEVEL
-- REPEATABLE READ (equivalent). Then SET it back when done but
-- be sure to do so in a separate batch.
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
SET TEXTSIZE 100     -- Just for illustration.  Too small for
                     -- real world.  4000 would be a better value.
BEGIN TRAN

DECLARE @mytextptr varbinary(16), @totalsize int,
    @lastread int, @readsize int

SELECT
    @mytextptr=TEXTPTR(pr_info), @totalsize=DATALENGTH(pr_info),
    @lastread=0,
    -- Set the readsize to the smaller of the @@TEXTSIZE setting
    -- and the total length of the column
    @readsize=CASE WHEN (@@TEXTSIZE < DATALENGTH(pr_info)) THEN
        @@TEXTSIZE ELSE DATALENGTH(pr_info) END
    FROM pub_info WHERE pub_id='1622'
```

```
IF @mytextptr IS NOT NULL AND @readsize > 0
    WHILE (@lastread < @totalsize)
    BEGIN
        READTEXT pub_info.pr_info @mytextptr @lastread @readsize
        IF (@@error <> 0)
            BREAK    -- Break out of loop if an error on read
        -- Change offset to last char read
        SELECT @lastread=@lastread + @readsize
        -- If read size would go beyond end, adjust read size
        IF ((@readsize + @lastread) > @totalsize)
            SELECT @readsize=@totalsize - @lastread
    END

COMMIT TRAN
GO
-- Set it back, but in a separate batch
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
```

Notice that in this example I need to ensure not only that the text pointer is still valid when I get to READTEXT but also that the column did not get changed *between* iterations of READTEXT. (If another connection simply updated the text in place, the text pointer would still be valid, although my read would be messed up since the contents and length were changed.) I could use HOLDLOCK both on the READTEXT statement as well as for protecting the text pointer. But for illustration, I instead chose to change the isolation level to REPEATABLE READ.

UPDATETEXT

UPDATETEXT, added in version 6.0, is a big improvement to text processing. In earlier versions, you were stuck with only WRITETEXT, which meant that to make even a minor change, you needed to completely rewrite the entire column. UPDATETEXT lets you work with text in pieces to insert, overwrite, or append data. Or you can copy data from another *text* column and append it or overwrite the column with it. Because of its additional capability and flexibility, the syntax for UPDATETEXT is a bit more complex:

```
UPDATETEXT table_name.dest_column_name dest_text_ptr
    offset delete_length [WITH LOG] [inserted_data |
    table_name.src_column_name src_text_ptr ]
```

The destination column name and text pointer parameters point to the column that you will be updating; these parameters are always used. Like you would with WRITETEXT, you should use the UPDLOCK hint to protect the text pointer from becoming invalid and to serialize access to the text pages to prevent a conversion deadlock. The source column name parameters are used only when you are copying data from another *text* column. Otherwise, you directly include in that spot the data you'll be adding or you omit the parameter if you are deleting data.

The *offset* is the position at which you start your data modification. It should be *NULL* if you are appending to the current contents and *0* if you are starting from the beginning of the column. The *delete_length* parameter tells you how many bytes to delete (if any) starting from the offset parameter. Use *NULL* for this parameter if you will delete all contents from the offset up to the end of the column, and use *0* if you will delete no bytes. As with READTEXT, the first character of the column is considered to have a 0 offset.

UPDATETEXT can do everything, and it can do much more than WRITETEXT can do. So you might choose to use only READTEXT and UPDATETEXT and forget about WRITETEXT. (WRITETEXT existed in versions before UPDATETEXT appeared, so the former is maintained for backward compatibility, but there isn't much need for it now.)

Following are some examples that will illustrate the use of UPDATETEXT better than further explanation.

EXAMPLE 1

Use UPDATETEXT to completely replace the contents of a column:

```
-- Use UPDATETEXT to completely overwrite a text column.
-- Alternative to WRITETEXT.
DECLARE @mytextptr varbinary(16)
BEGIN TRAN

SELECT @mytextptr=TEXTPTR(pr_info) FROM pub_info (UPDLOCK) WHERE
    pub_id='9999'
IF @mytextptr IS NOT NULL
    UPDATETEXT pub_info.pr_info @mytextptr 0 NULL WITH LOG
        "New text for 9999"

COMMIT TRAN
```

EXAMPLE 2

Use UPDATETEXT to delete characters off the end; first notice that publisher 0877, Binnet, has the following contents in the text column *pr_info*:

```
This is sample text data for Binnet & Hardley, publisher 0877 in
the pubs database. Binnet & Hardley is located in Washington,
D.C.
This is sample text data for Binnet & Hardley, publisher 0877 in
the pubs database. Binnet & Hardley is located in Washington,
D.C.
```

```
This is sample text data for Binnet & Hardley, publisher 0877 in
the pubs database. Binnet & Hardley is located in Washington,
D.C.
This is sample text data for Binnet & Hardley, publisher 0877 in
the pubs database. Binnet & Hardley is located in Washington,
D.C.
This is sample text data for Binnet & Hardley, publisher 0877 in
the pubs database. Binnet & Hardley is located in Washington,
D.C.
```

Because the text is repeated several times, I want to delete all characters that follow the first occurrence of *D.C.* Here's how:

```
DECLARE @mytextptr varbinary(16), @pat_offset int
BEGIN TRAN
SELECT @mytextptr=TEXTPTR(pr_info),
    @pat_offset=PATINDEX('%D.C.%', pr_info)-1+4
    -- For offset, subtract 1 for offset adjust but add 4 for
    -- length of "D.C."
    FROM pub_info (UPDLOCK) WHERE pub_id='0877'

IF @mytextptr IS NOT NULL AND @pat_offset >= 0
    UPDATETEXT pub_info.pr_info @mytextptr @pat_offset NULL WITH LOG

COMMIT TRAN
```

The column now has these contents (only):

```
This is sample text data for Binnet & Hardley, publisher 0877 in
the pubs database. Binnet & Hardley is located in Washington, D.C.
```

EXAMPLE 3

With the small amount of text here, it wouldn't be bad to simply rewrite the column with new text. But if this were a large *text* column (you could literally store the contents of *War and Peace* in a single *text* column), it would be extremely inefficient to rewrite the entire column just to make a minor change. In this example, I want to add the text "Mary Doe is president of the company." to the current contents. I'll use UPDATETEXT to append text to the column:

```
DECLARE @mytextptr varbinary(16)
BEGIN TRAN
SELECT @mytextptr=TEXTPTR(pr_info) FROM pub_info (UPDLOCK)
    WHERE pub_id='0877'
```

```
IF @mytextptr IS NOT NULL
    UPDATETEXT pub_info.pr_info @mytextptr NULL NULL WITH LOG
        "Mary Doe is president of the company."

COMMIT TRAN
```

And the result:

```
This is sample text data for Binnet & Hardley, publisher 0877 in
the pubs database. Binnet & Hardley is located in Washington,
D.C.Mary Doe is president of the company.
```

That worked exactly as I specified, but I really wish I had skipped a line and then included a tab before adding the new sentence. I can easily add both a vertical and a horizontal tab, as you can see in Example 4.

EXAMPLE 4

Use UPDATETEXT to insert some characters:

```
DECLARE @mytextptr varbinary(16), @pat_offset int,
    @mystring char(2)
BEGIN TRAN
SELECT
@mystring=char(13) + CHAR(9),    -- Vertical tab is code point 13.
                                 -- Tab is 9.
@pat_offset=PATINDEX('%Mary%', pr_info)-1,
@mytextptr=TEXTPTR(pr_info) FROM pub_info (UPDLOCK)
    WHERE pub_id='0877'

IF @mytextptr IS NOT NULL AND @pat_offset >= 0
    UPDATETEXT pub_info.pr_info @mytextptr @pat_offset 0 WITH LOG
        @mystring

COMMIT TRAN
```

And the result:

```
This is sample text data for Binnet & Hardley, publisher 0877 in
the pubs database. Binnet & Hardley is located in Washington,
D.C.
    Mary Doe is president of the company.
```

Oops! I just learned that the president is *Marie Dow,* not *Mary Doe.* I need to fix that.

EXAMPLE 5

Use UPDATETEXT for search and replace:

```
-- UPDATETEXT for Search and Replace
DECLARE @mytextptr varbinary(16), @pat_offset int,
    @oldstring varchar(255), @newstring varchar(255),
    @sizeold int

BEGIN TRAN
SELECT @oldstring="Mary Doe", @newstring="Marie Dow"

SELECT @sizeold=DATALENGTH(@oldstring),
@pat_offset=PATINDEX('%' + @oldstring + '%', pr_info)-1,
@mytextptr=TEXTPTR(pr_info)
FROM pub_info (UPDLOCK) WHERE pub_id='0877'

IF @mytextptr IS NOT NULL AND @pat_offset >= 0
    UPDATETEXT pub_info.pr_info @mytextptr @pat_offset @sizeold
        WITH LOG @newstring

COMMIT TRAN
```

And the result:

```
This is sample text data for Binnet & Hardley, publisher 0877 in
the pubs database. Binnet & Hardley is located in Washington,
D.C.
    Marie Dow is president of the company.
```

I used variables above and figured lengths and offsets using SQL Server's built-in functions. By doing this, I ensured that the procedure is pretty generic and that it can deal with changing the string to another string that is either longer or shorter than the original.

EXAMPLE 6

Suppose that I want to append the contents of the text for publisher Scootney (*pub_id* 9952) to the text for Binnet (*pub_id* 0877). If I did not have this option in UPDATETEXT, it would be necessary to bring all that text back to the client application, append it, and then send it back to the server. Over a slow network like the Internet, this would not be practical if the *text* columns were large. But with UPDATETEXT, the whole operation is done on the server. In this example, notice that I protect the text pointer for the target with UPDLOCK, since I'll be updating that row, but I use HOLDLOCK for the source row since I am reading it only and I want to ensure that it hasn't changed.

I'll use UPDATETEXT to copy and append one text column to another:

```
-- UPDATETEXT to copy and append another text column
DECLARE @target_textptr varbinary(16),
    @source_textptr varbinary(16)
BEGIN TRAN

SELECT @target_textptr=TEXTPTR(pr_info) FROM pub_info (UPDLOCK)
WHERE pub_id='0877'
SELECT @source_textptr=TEXTPTR(pr_info) FROM pub_info (HOLDLOCK)
WHERE pub_id='9952'

IF @target_textptr IS NOT NULL AND @source_textptr IS NOT NULL
    UPDATETEXT pub_info.pr_info @target_textptr NULL NULL
        WITH LOG pub_info.pr_info @source_textptr

COMMIT TRAN
```

Environmental Concerns

To finish this discussion of Transact-SQL programming, I'll introduce some of the environmental concerns that you need to be aware of in your programming—for example, case sensitivity, which can greatly affect your applications. I'll also discuss nullability issues and ANSI compatibility.

Case Sensitivity

Various options and settings affect the semantics of your Transact-SQL statements. You must be sure that your Transact-SQL code can work regardless of the setting, or you must control the environment so that you know what the setting is.

Case sensitivity is by far the most common environmental problem, and it is simple to avoid. I recommend that you do most of your development in a case-sensitive environment, even if you will deploy your application mostly in a case-insensitive environment. The reason is simple: nearly all operations that work in a case-sensitive environment will also work in a case-insensitive environment, but the converse is not true. For example, if I write the statement *select * from authors* in the *pubs* database of a case-sensitive environment, it will work equally well in a case-insensitive environment. On the other hand, the statement *SELECT * FROM AUTHORS* will work fine in a case-insensitive environment but will fail in a case-sensitive environment. The table in *pubs* is actually named *authors,* which is lowercase. The only instance I can think of that would work in a case-sensitive environment but would fail in a case-insensitive environment is in the declaration of an object name, a column name, or a variable name. For example, with the statement *declare @myvar int*, using *@MYVAR int* would work fine in a case-sensitive environment because the two names are distinct, but it would fail in a case-insensitive environment because the names would be considered duplicates.

The easiest way to determine whether your environment is case-sensitive is to do a SELECT statement with a WHERE clause that compares a lowercase letter with its uppercase counterpart—you wouldn't need to access a table to do this. The following simple SELECT statement returns 1 if the server is case-sensitive and 0 if the server is case-insensitive:

```
SELECT CASE
    WHEN ('A'='a') THEN 0
    ELSE 1
END
```

Case sensitivity is just one of the issues surrounding the character set used by SQL Server. The character set choice will affect both the rows selected and their ordering in the result set for a query such as this:

```
SELECT au_lname, au_fname FROM authors
    WHERE au_lname='Josè'
ORDER BY au_lname, au_fname
```

If you never use characters that are not in the standard ASCII character set, case sensitivity is really your primary issue. But if your data has special characters like the *è* in this example, be sure that you understand character-set issues. (For more information, see Chapter 4, "Planning for and Installing SQL Server.")

Nullability and ANSI Compliance Settings

In order to pass the NIST test suite for ANSI SQL-92 compliance, various options had to be enabled in version 6.5 because of subtle differences in semantics between the traditional SQL Server behavior and what is mandated by ANSI. I have discussed the majority of these issues in earlier chapters. To preserve backward compatibility, the prior behavior couldn't simply be changed. So we added options (or in a few cases, previous options were toggled on) to change the semantics to comply with the ANSI SQL requirements. These options are summarized below. (I've also listed the statement used to change the behavior.)

- Disable SQL Server's = NULL extension (*SET ANSI_NULLS ON*).
- Automatically display a warning if a truncation would occur because the target column is too small to accept a value. By default, SQL Server truncates without any warning (*SET ANSI_WARNINGS ON*).
- Always right-pad *char* columns, and don't trim trailing blanks that were entered in *varchar* columns, as SQL Server would do by default (*SET ANSI_PADDING ON*).
- Make statements implicitly part of a transaction, requiring a subsequent COMMIT or ROLLBACK (*SET IMPLICIT TRANSACTIONS ON*).

- Terminate a query if an overflow or divide-by-zero error occurs (*SET ARITHABORT ON*). By default, SQL Server returns NULL for these operators, issues a warning message, and proceeds.
- Close any open cursors upon COMMIT of a transaction. By default, SQL Server keeps the cursor open so that it can be reused without incurring the overhead of reopening it (*SET CURSOR_CLOSE_ON_COMMIT ON*).
- Allow identifier names to include SQL Server keywords if the identifier is included in double quotation marks, which by default is not allowed. This causes single and double quotation marks to be treated differently (*SET QUOTED_IDENTIFIER ON*).
- By default, create as NOT NULL a column in a CREATE TABLE statement that is not specified as NULL or NOT NULL. *SET ANSI_NULL-_DFLT_ON ON* toggles this so that the column can be created with NULL. (I recommend that you always specify NULL or NOT NULL so that this setting option is irrelevant.) The nullability of a column not explicitly declared is determined by the setting at the time the table was created, which could be different from the current setting.

All of the above options can be set individually, but I'd avoid doing that because there are 256 (2^8) permutations to consider. You might want to set a few of the options individually, such as *SET ARITHABORT* or *SET ARITHIGNORE*. But by and large, I'd either leave them all at their default settings (my preference) or change them as a group to the ANSI SQL-92 behavior. These options can be enabled as a group by setting *SET ANSI_DEFAULTS ON*.

The ability to set these options on a per-connection basis makes life "interesting" for you as a SQL Server application programmer. Your challenge is to recognize and deal with the fact that these settings will change the behavior of your code. Basically, that means that you need to adopt some form of the following four strategies:

- The Optimistic Approach. Hope that none of your users or the person doing database administration will change such a setting. Augment your optimism by educating users not to change these settings.
- The Flexible Approach. Try to write all your procedures as to accommodate all permutations of the settings of all the various options (usually not practical).
- The Hard-Line Approach. Explicitly set your preferences at startup and periodically recheck them to determine that they have not been subsequently altered. Simply refuse to run if the settings are not exactly what you expect.

- The Clean Room Approach. Have a "sealed" system that prevents anyone from having direct access to change such a setting.

Whichever of these approaches you take is your choice, but recognize that if you don't think about the issue at all, you have basically settled for the Optimistic Approach. This approach is certainly adequate for many applications for which it's pretty clear that the user community would have neither the desire nor the ability to make environmental changes. But if you are deploying an application and the SQL Server will be accessed by applications that you do not control, it is probably an overly simplistic approach. Philosophically, the Flexible Approach is nice, but I don't think it's realistic unless the application is quite simple.

You can change the SQL Server default values for the server as a whole by using **sp_configure 'user options'**. A specific user connection can then further refine the environment by issuing one of the specific SET statements discussed above. The global variable @@OPTIONS can then be queried by any connection to see the current settings for that connection. The @@OPTIONS variable and the value to be set using **sp_configure 'user options'** are a bit mask with the following values:

Decimal Value	*Hex Value*	*Option and Description*
1	0x0001	DISABLE_DEF_CNST_CHK. Controls interim constraint checking.
2	0x0002	IMPLICIT_TRANSACTIONS. Controls whether a transaction is started implicitly when a statement is executed.
4	0x0004	CURSOR_CLOSE_ON_COMMIT. Controls behavior of cursors once a commit has been performed.
8	0x0008	ANSI_WARNINGS. Controls truncation and NULL in aggregate warnings.
16	0x0010	ANSI_PADDING. Controls padding of variables.
32	0x0020	ANSI_NULLS. Controls NULL handling by using equality operators.
64	0x0040	ARITHABORT. Terminates a query when an overflow or divide-by-zero error occurs during query execution.
128	0x0080	ARITHIGNORE. Returns NULL when an overflow or divide-by-zero error occurs during a query.
256	0x0100	QUOTED_IDENTIFIER. Differentiates between single and double quotation marks when evaluating an expression, allowing object names to include characters that would otherwise not conform to naming rules or would collide with a reserved word or a keyword.

Decimal Value	*Hex Value*	*Option and Description*
512	0x0200	NOCOUNT. Turns off the message returned at the end of each statement that states how many rows were affected by the statement.
1024	0x0400	ANSI_NULL_DFLT_ON. Alters the session's behavior to use ANSI compatibility for nullability. New columns defined without explicit nullability will be defined to allow NULLs.
2048	0x0800	ANSI_NULL_DFLT_OFF. Alters the session's behavior to not use ANSI compatibility for nullability. New columns defined without explicit nullability will be defined not to allow NULLS.

By default, none of these options is enabled. So in a brand-new SQL Server 6.5 installation, the run value for **sp_configure 'user options'** will be 0. The SA can set this so that all connections have the same initial default settings. If you query the value of @@OPTIONS from an application that has not modified the environment, the value will also be 0. However, be aware that many applications, or even the SQL Server ODBC driver that the application uses, might have changed the environment. For example, if you use ISQLW.EXE, you may well see a value of 512 for @@OPTIONS if you have the No Count Display option checked under Query Options. If you are using an ODBC-based application, you might have options 256 (QUOTED_IDENTIFIER) and 16 (ANSI_PADDING) set.

To change the default behavior, simply set the corresponding bit by doing a bitwise OR with the previous value. For example, suppose that your run value is 512, which indicates that NOCOUNT is the only option turned on. You want to leave NOCOUNT enabled, but you also want to enable option number 1, which turns off the ability to deal with interim constraint violations. You'd simply pass the decimal value 513 (or 0x201) to **sp_configure 'user options'**, which is the result of doing a bitwise OR between the two options (for example, SELECT 1 | 512).

You can examine current options that have been set using *DBCC USER OPTIONS*. The output is similar to this:

```
Set Option      Value
----------      ----------
textsize        64512
language        us_english
dateformat      mdy
datefirst       7
arithabort      SET
nocount         SET
```

This DBCC command shows only options that have been set—it doesn't show all the current settings for **sp_configure 'user options'**. But you can also decode your current connection settings pretty easily from @@OPTIONS using something like this:

```
SELECT "DISABLE_DEF_CNST_CHK" AS "OPTION",
    "SETTING"=CASE WHEN (@@OPTIONS & 0x0001 > 0) THEN 'ON' ELSE 'OFF'
    END
UNION
SELECT "IMPLICIT TRANSACTIONS", CASE WHEN
    (@@OPTIONS & 0x0002 > 0) THEN 'ON' ELSE 'OFF' END
UNION
SELECT "CURSOR_CLOSE_ON_COMMIT", CASE WHEN
    (@@OPTIONS & 0x0004 > 0) THEN 'ON' ELSE 'OFF' END
UNION
SELECT "ANSI_WARNINGS",CASE WHEN (@@OPTIONS & 0x0008 > 0) THEN
    'ON' ELSE 'OFF' END
UNION
SELECT "ANSI_PADDINGS", CASE WHEN (@@OPTIONS & 0x0010 > 0) THEN
    'ON' ELSE 'OFF' END
UNION
SELECT "ANSI_NULLS", CASE WHEN (@@OPTIONS & 0x0020 > 0) THEN 'ON'
    ELSE 'OFF' END
UNION
SELECT "ARITHABORT", CASE WHEN (@@OPTIONS & 0x0040 > 0) THEN 'ON'
    ELSE 'OFF' END
UNION
SELECT "ARITHIGNORE", CASE WHEN (@@OPTIONS & 0x0080 > 0)
    THEN 'ON' ELSE 'OFF' END
UNION
SELECT "QUOTED_IDENTIFIER", CASE WHEN (@@OPTIONS & 0x0100 > 0)
    THEN 'ON' ELSE 'OFF' END
UNION
SELECT "NOCOUNT", CASE WHEN (@@OPTIONS & 0x0200 > 0) THEN 'ON'
    ELSE 'OFF' END
UNION
SELECT "ANSI_NULL_DFLT_ON", CASE WHEN (@@OPTIONS & 0x0400 > 0)
    THEN 'ON' ELSE 'OFF' END
UNION
SELECT "ANSI_NULL_DFLT_OFF", CASE WHEN (@@OPTIONS & 0x0800 > 0)
    THEN 'ON' ELSE 'OFF' END
ORDER BY "OPTION"
```

Here's the result:

```
OPTION                  SETTING
----------------------  -------
ANSI_NULL_DFLT_OFF      OFF
ANSI_NULL_DFLT_ON       OFF
ANSI_NULLS              OFF
ANSI_PADDINGS           OFF
ANSI_WARNINGS           OFF
ARITHABORT              OFF
ARITHIGNORE             OFF
CURSOR_CLOSE_ON_COMMIT  OFF
DISABLE_DEF_CNST_CHK    ON
IMPLICIT TRANSACTIONS   OFF
NOCOUNT                 ON
QUOTED_IDENTIFIER       OFF
```

Locale-Specific SET Options

Beware of the locale-specific SET options. *SET DATEFORMAT* and *SET DATEFIRST* change the recognized default date format. If DATEFORMAT is changed to *dmy* instead of the (U.S.) default *mdy,* a date such as *'12/10/96'* will be interpreted as October 12, 1997. I think a good strategy for dates is to always use the ANSI format yyyy.mm.dd, which is recognized no matter what the setting is of DATEFORMAT.

DATEFIRST affects what is considered the first day of the week. By default (in the U.S.), it has the value 7 (Sunday). Date functions that work with the day-of-week as a value between 1 and 7 will be affected by this setting. These day-of-week values can be confusing, since their numbering depends on the DATEFIRST value; but the values for DATEFIRST don't change. For example, as far as DATEFIRST is considered, Sunday's value is always 7. But having then designated Sunday (7) as DATEFIRST, if you did a *SELECT DATEPART(dw,GETDATE())* and your date falls on a Sunday, the statement will return 1. You just defined Sunday to be the first day of the week, so 1 is correct.

SUMMARY

Transact-SQL statements can be grouped together in batches, they can persist in the database, they can repeatedly execute as stored procedures, and they can be made to automatically fire as triggers. It is essential that you understand the differences between these functions and that you understand that their actions are not mutually exclusive.

Transact-SQL stored procedures can be quite complex, and they can become a significant portion of your application's source code. Fortunately, Microsoft Developer Studio provides debugging support for SQL Server stored procedures.

Programming effectively with Transact-SQL also requires that you understand transactional topics, such as when transactions will be committed and when they can be rolled back. Since you'll likely be working in a multiuser environment, it is vital that you make the appropriate concurrency and consistency choices to suit the isolation level for your environment. Understanding isolation levels is also important for working with BLOBs in SQL Server using the special operators READTEXT, WRITETEXT, and UPDATETEXT. And no matter what task you are performing, it is important to realize that you must plan for various environmental options that will affect the behavior and semantics of your Transact-SQL code.

Naming Conventions

Many organizations and multiuser development projects adopt standard naming conventions, which are a good thing, in general. For example, assigning a standard moniker of *cust_id* to represent a customer number in every table makes it obvious that all the tables have data in common. If, instead, several monikers were used in the tables to represent a customer number, such as *cust_id, cust_num, customer_number,* and *customer_#,* it would not be so obvious that these monikers represented common data. One convention I see occasionally and recommend *against* using is Hungarian-style notation for column names. (Hungarian notation is a widely used practice in C programming, whereby variable names include information about their datatypes. Its name is attributed to its use by legendary Microsoft programmer Charles Simonyi, who is of Hungarian ancestry.) Hungarian-style notation uses names such as *sint_nn_custnum* to represent that the *custnum* column is a small integer (*smallint* of 2 bytes) and is NOT NULL (does not allow nulls). Although this practice makes good sense in C programming, it defeats the datatype independence that SQL Server provides.

Suppose it is discovered, for example, that the *custnum* column requires a 4-byte integer (*int*) instead of a 2-byte small integer. It is relatively simple to re-create the table with the column as an *int* instead of a *smallint.* In SQL Server, stored procedures will deal with the different datatype automatically. Applications using DB-Library or ODBC that bind the retrieved column to a character or integer datatype will be unaffected. The applications would need to change if they bound the column to a small integer variable, as the variable's type would need to be larger. For this reason, it is best to try not to be overly conservative with variable datatypes, especially in your client applications. You should be most concerned with the type on the server side; the type in the application can be larger and will automatically accommodate smaller values. By overloading the column name with datatype information, which is readily available from the system catalogs, the insulation from the underlying datatype is compromised. (You could, of course, change the datatype from a *smallint* to an *int,* but then the Hungarian-style name would no longer accurately reflect the column definition. Changing the column name would then result in the need to change application code or stored procedures or both.)

Datatypes

SQL Server provides a large number of datatypes, as shown in Table 6-1 on pages 202–03. Choosing the appropriate datatype is simply a matter of mapping the domain of values you need to store to the corresponding datatype. In choosing datatypes, you want to avoid wasting storage space while allowing enough space for a sufficient range of possible values over the life of your application.

Datatype synonyms

SQL Server syntactically accepts as datatypes both the words listed as synonyms and the base datatypes shown in Table 6-1, but it uses only the type listed as the datatype. For example, a column can be defined as *character(1), character,* or *char(1),* and SQL Server will accept all these as valid syntax. Internally, however, the expression is considered *char(1),* and subsequent querying of the SQL Server system catalogs for the datatype will show it as *char(1),* regardless of the syntax that was used when it was created.

Nullable columns are variable-length

Before deciding to use an ostensibly fixed-length datatype such as *char* instead of a variable-length one such as *varchar,* it is important that you understand *nullability*: all datatypes, with the exception of *bit,* can be declared either NULL or NOT NULL (that is, they can allow or disallow a null entry). Internally, declaring a column to allow a null entry makes that column a variable-length column. For example, a column declared as *char(5) NULL* is internally identical to one declared *varchar(5) NULL.* In both cases, if a null value is entered, no storage is consumed. If only 3 bytes are entered, then only 3 bytes of storage are used, even for the fixed-length type.

NOTE There is an exception to this. The command *SET ANSI_PADDING ON* instructs SQL Server to physically store spaces in the remaining 2 bytes of the *char(5)* type, in which case 5 bytes of storage would be used. This setting conforms to the ANSI SQL-92 standard.

Variable-length vs. fixed-length datatypes

Deciding to use a variable-length or a fixed-length datatype is not always straightforward or obvious. As a general rule, variable-length datatypes are most appropriate when you expect significant variance in the size of the data for a column and the data in the column will not be frequently changed.

Using variable-length datatypes can yield important storage savings. Choosing them can sometimes result in performance loss (as I will explain in a moment) and at other times can result in improved performance. A row with variable-length columns (including supposed fixed-length columns that allow NULLs) requires special offset and adjust entries to be internally maintained. These entries keep track of the actual length of the column. Calculating and maintaining the offsets requires slightly more overhead than a pure fixed-length row, which needs no such offsets at all. This is a CPU task of a few addition and subtraction operations to maintain the offset value. However, the extra overhead of maintaining these offsets is generally inconsequential, and I have not seen a system in which this alone made a significant difference. A more significant performance difference might arise from the method by which updates are processed.

Type of Data	Base Datatype	Synonyms	Range/Domain	Storage Size
Integer	*int*	*integer*	Whole numbers from –2,147,483,648 to 2,147,483,647	4 bytes
	smallint		Whole numbers from –32,768 to 32,767	2 bytes
	tinyint		Whole numbers from 0 to 255	1 byte
Packed decimal (exact numeric)	*numeric (p,s)* *decimal (p,s)*	*dec*	Whole or fractional numbers from -10^{38} to 10^{38}	2–17 bytes, depending on specified precision, p, which can range to 38 digits. On average, 1 byte of storage is required per every 2 digits of precision.
Floating point (approx numeric)	*float* (15-digit precision)	*float(n)*, where *n* is between 8 and 15 Double precision	Approximations of numbers from $-1.79E^{308}$ to $1.79E^{308}$ Positive range: $2.23E^{-308}$ to $1.79E^{308}$ Negative range: $-2.23E^{-308}$ to $-1.79E^{308}$	8 bytes
	real (7-digit precision)	*float(n)*, where *n* is between 1 and 7	Approximations of numbers from $-3.40E^{38}$ to $3.40E^{38}$ Positive range: $1.18E^{-38}$ to $3.40E^{38}$ Negative range: $-1.18E^{-38}$ to $-3.40E^{38}$	4 bytes
Character (fixed length)	*char(n)*	*character (n)* *character* (character without a specific size is synonymous to a 1-character field, char(1))	Up to 255 characters, as designated by *n*, of the installed character set	1 byte per character *n* declared, even if partially unused
Character (variable length)	*varchar(n)*	*character varying (n)*, *char varying (n)*	Up to 255 characters, as designated by *n*, of the installed character set	1 byte per character stored. Declared but unused characters do not consume storage.
Monetary	*money*		Numbers with accuracy to one ten-thousandth of a unit (four decimal places), typically used to store currency values. From –922,337,203,685,477.5808 to 922,337,203,685,477.5807	8 bytes
	smallmoney		Numbers with accuracy to one ten-thousandth of a unit (four decimal places), typically used to store currency values. From –214,748.3648 to 214,748.3647	4 bytes

Table 6-1. *SQL Server supplies many datatypes.*

Type of Data	Base Datatype	Synonyms	Range/Domain	Storage Size
Date and Time	*datetime*		Combined date and time representation. (SQL Server does not have separate DATE and TIME datatypes.) Date part: 01-JAN-1753 to 31-DEC-9999 Time part: Number of milliseconds since midnight of the given date	8 bytes
	smalldatetime		Combined date and time representation. Date part: 01-JAN-1900 to 06-JUN-2079 Time part: Number of minutes since midnight of the given date	4 bytes
Binary (fixed length)	*binary(n)*		Any binary representation (bit patterns) up to 255 bytes	*n* bytes, even if *n* is partially unused.
Binary (variable length)	*varbinary(n)*	*binary varying*	Any binary representation (bit patterns) up to 255 bytes	The number of bytes actually stored. No storage for space declared but not used.
Long text/BLOB	*text* and *image*		Text: Character data up to 2 GB. Image: Binary data up to 2 GB. The text and image datatypes are always variable length.	If not null, a 16-byte pointer is used on the data page, plus however many 2-KB pages are required to store the actual length. Text and image pages cannot be shared. A single byte entered in a text or image column requires its own 2-KB page (most being unused).
Boolean	*bit*		0 or 1	Bit datatypes share a byte with other bit columns of the same table. Hence, 8-bit columns of the same table use 1 byte of storage. If the table has only 1-bit columns, it still uses 1 byte, although 7 more such columns could be added "for free."

mspress.microsoft.com

Microsoft® Inside SQL Server™ 6.5

The authoritative inside story—with practical advice from the ultimate insider.

This comprehensive guide, written by Microsoft insider and SQL guru Ron Soukup, provides an authoritative conceptual and architectural overview and advice on installation, administration, and programming with SQL and Transact-SQL. Inside you'll find:

- A historical overview of the features, capabilities, and architecture of the SQL Server product
- Detailed SQL programming topics, including databases and devices; querying and modifying data; programming with Transact-SQL; using batches, transactions, stored procedures, and triggers; locking data; data replication; SQL Server database administration; and much more
- New SQL Server internals information for advanced performance optimizing and troubleshooting
- Examples and brainteasers for testing your knowledge of Transact-SQL
- System performance and tuning considerations, including system design and querying, plus how to configure and monitor your application's performance

Soukup also provides examples and candid answers to frequently asked questions gleaned from his years of service as general manager of the SQL Server development group at Microsoft. This book is for MIS professionals in large companies, vertical applications developers, custom solution providers, and anyone else working with mid-level to high-end relational databases. In fact, it's a must-read for all those who want to understand Microsoft SQL Server from the inside out.

The enclosed CD-ROM gives you great tools and files, including:

- Complete Microsoft SQL Server Books Online documentation, the ideal tool for handy reference or for product evaluation by technical management
- Sample code and scripts used in the book
- White papers on replication and front-end development tools and choices, plus a discussion of CUBE by database authority Jim Gray
- A benchmark kit for measuring system performance

Includes a foreword by Jim Gray, a leading database authority and now head of the Microsoft San Francisco Research Lab.

Microsoft Programming Series. The resources you need for making great software.

Inside titles: Inside information, from actual insiders, on the hows and whys of software design. ***Programming*** titles: Definitive shop manuals for professional programmers. ***Workshop*** titles: Working toolkits complete with ready-to-use code on CD-ROM. ***Active*** titles: The keys to using active Web technologies with Microsoft's Visual tools to create dazzling, active Web sites.

U.S.A.	**$49.99**	
U.K.	£46.99	[V.A.T. included]
Canada	$69.99	

[Recommended]

Programming/SQL Server

Microsoft Press