

RDBMS Concepts Using SQL Server 2000

Table of Content

RDBMS Concepts	3
Introduction	3
What is a Database?.....	3
Database tools	5
Introduction to Normalization.....	20
SQL Server Database Architecture	27
The SQL Server Engine	30
SQL Data Types	53
Key Data Type	54
User-Defined Datatypes :.....	55
Constraints	60
Primary key	61
Foreign key	66
Constraints	73
Joins	88
Subqueries	101
Correlated Subqueries	106
Functions	112
Introduction	112
Index	119
Index Space Requirements.....	128
Managing an Index	131
Using an Index	136
Views	139
Introduction	139
Programming with Transact-SQL	149
Variables	152
Control-of-Flow Tools.....	157
CASE	158
PRINT	161
RAISERROR.....	162
FORMATMESSAGE	164
Operators	164
Scalar Functions	170
Table-Valued Functions	197

RDBMS Concepts

Introduction

Microsoft SQL Server is a client/server database management system. A client/server database management system consists of two components: a *front-end* component (the client), which is used to present and manipulate data, and a *back-end* component (the database server), which is used to store, retrieve, and protect the databases.

You can use Microsoft Access on a client workstation to access databases on a Microsoft *SQL server.

When you query a database, the SQL server can process the query for you and send the results to your workstation. In contrast, if you query a database on a file server, the file server must send the entire database to your workstation so that your workstation can process the query. Thus, using SQL Server enables you to reduce the traffic on your network.

Suppose any company has a customer information database that is 50 MB in size. If you query the database for a single customer's information and the database is stored on a file server, all 50 MB of data must be sent to your workstation so that your computer can search for the customer. In contrast, if you query the same database stored on a SQL server, the SQL server processes your query and sends only the one customer's information to your workstation.

Structured Query Language (SQL) is a standardized set of commands used to work with databases.

Relational Database Management System (RDBMS) uses established relationships between the data in a database to ensure the integrity of the data. These relationships enable you to prevent users from entering incorrect data.

Transact-SQL is an enhanced version of SQL. Transact-SQL commands is used to create, maintain, and query databases. You can use these commands either directly, by manually entering commands, or indirectly, by using a client application such as Microsoft Access that is written to issue the necessary SQL commands.

What is a Database?

*That which contains the physical implementation of the schema and the data is called as database. The *schema conceptually describes the *problem space to the database.*

When you install SQL Server, the Setup utility automatically creates several system and sample user databases. *System* databases contain information used by SQL Server to operate. You create *user* databases-and they can contain any information you need to collect. You can use *SQL Server Query Analyzer to *query any of your SQL databases, including the system and sample databases.

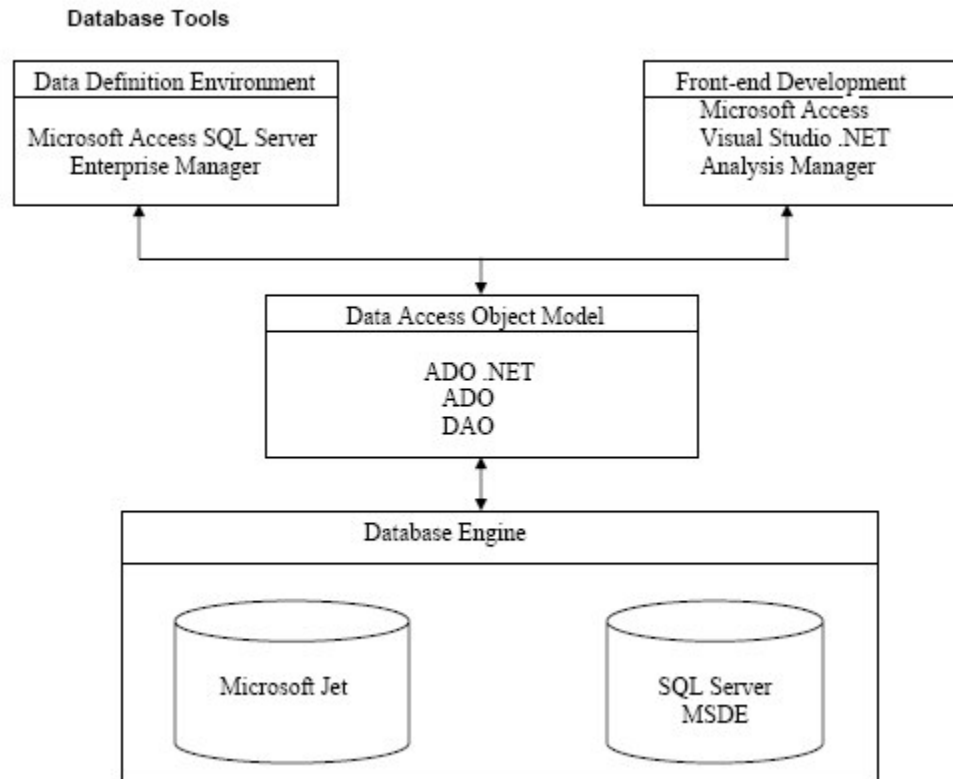
- Distribution
 - This database is not created until you configure replication because it contains history information about replication.
- Master

- Information about the operation of SQL Server, including user accounts, other SQL Servers, environment variables, error messages, databases, storage space allocated to databases, and the tapes and disk drives on the SQL Server.
- Model
 - A template for creating new database
- Msdb
 - Information about all scheduled job on your server. This information is used by the SQL Server Agent service.
- Northwind
 - A sample user database for learning SQL Server. It contains information about a fictitious gourmet food company's customer, sales and employees.
- Pubs
 - A sample user database for learning SQL Server. It contains information about a fictitious publishing company's authors, publishers, royalties and titles.
- Tempdb
 - It contains temporary information and is used as a scratchpad by SQL Server.

- *Database schema, which is conceptual, not physical is the translation of the conceptual model into a physical representation that can be implemented using a DBMS.*
- *Problem space is the well defined part of the real world which is relational databases analogies, intended to model some aspect of the real world.*
- *SQL Query Analyzer is used to run SQL queries as well as to optimize the performance of the queries.*
- *Query is simply a command consisting of one or more SQL statements, which you send to SQL server to request data from the server, change data, or delete data.*

The schema is nothing more than the data model expressed in the terms that you will use to describe it to the database engine tables and triggers and such creatures.

Database tools



Database Engines

At the lowest level are the database engines. These are sometimes called "back ends," but that's a bit sloppy since the term "back end" really refers to a specific physical architecture. These tools will handle the physical manipulation of storing data onto the disk and feeding it back on demand. We'll be looking at two: the Jet database engine and SQL Server. You may be surprised not to see Microsoft Access here. Access is technically a front-end development environment that uses either Jet or SQL Server natively and can, in fact, use any ODBC-compliant database engine as its data store. It uses the Jet database engine to manipulate data stored in .mdb files and SQL Server (or another ODBC data store) for data stored in .adp files. Access has always used the Jet database engine, although Microsoft didn't expose it as a separate entity until the release of Microsoft Visual Basic 3.

The Jet database engine and SQL Server, although very different, are both wonderful tools for storing and manipulating data. The difference between them lies in their architectures and the problems they are intended to address. Microsoft Jet is a "desktop" database engine, intended for small- to medium-sized systems. (Please note that this does not imply that the Jet database engine is appropriate only for trivial systems.) SQL Server, on the other hand, uses a client/server architecture and is intended for medium-sized to huge systems, scalable to potentially thousands of users running mission-critical applications. MSDE (an acronym for Microsoft Desktop Engine) is a scaled-down version of SQL Server intended for desktop use. From a designer's viewpoint, there is little difference between MSDE and the full version of SQL Server, and we won't be considering it further here. We'll be looking at the differences between the two database engines throughout and discussing the trade-offs between the two architectures.

Data Access Object Models

Microsoft Access, and to a lesser extent Visual Studio .NET, provides simple mechanisms for binding form controls directly to a data source, avoiding the necessity for dealing directly with the database engine. For various reasons that we'll be looking at, however, this is not always either possible or appropriate. In these instances, you'll need to use a data access object model to manipulate the data in code.

A data access object model is a kind of glue between the programming environment and the database engine; it provides a set of objects with properties and methods that can be manipulated in code. Since this book deals primarily with design rather than implementation, we won't be discussing the trade-offs between these models in any great depth, but I believe it useful to review them here.

Microsoft (currently) provides three data access object models: Data Access Objects (DAO), which comes in two flavors (DAO/Jet and DAO/ODBCDirect), Microsoft ActiveX Data Objects (ADO) and ADO.NET.

DAO, the oldest of the three, is the native interface to the Jet database engine. The statements of Microsoft's marketing department notwithstanding, it is the most efficient object model to use for manipulating Jet databases within Microsoft Access. ADO uses a smaller object hierarchy than DAO, consisting of only four primary objects, and provides some significant extensions to the model, for example, its support for disconnected recordsets and data shaping. It is used within Microsoft Access and other products that support VBA for manipulating any ODBC-compliant database, including both Jet and SQL Server. ADO.NET is, of course, the version of ADO that is used when working within the .NET Framework.

Data Definition Environments

Microsoft Jet and SQL Server handle the physical aspects of manipulating data for us, but we need some way to tell them how to structure the data. Microsoft provides a plethora of methods for doing this, but we'll be looking at only three in detail: Access and the SQL Server Enterprise Manager for relational models, and the Analysis Manager for dimensional models. There are other tools that provide roughly the same capabilities, but these are the ones I prefer. Of course, once you understand the principles, you can use whichever tools best get the job done for you. It's also possible to define the structure of your database using code, and we'll look at how you go about doing this, although under normal circumstances I don't recommend it. Unless for some reason you need to alter the structure of the data during the run-time use of your application (and with the possible exception of temporary tables, I'm highly suspicious of this practice if the database schema isn't stable, you probably haven't understood the problem domain), the interactive tools are quicker, easier, and a lot more fun to use.

Front-End Development

Once the physical definition of your database is in place, you'll need tools to create the forms and reports your users will interact with. We'll draw our example from two of these: Access and Visual Studio .NET (specifically, Visual Basic .NET). Again, there are hundreds of front-end tools around, but the design principles remain the same, so you should be able to apply what you learn here to your front-end tool of choice.

The Relational Model

The relational model is based on a collection of mathematical principles drawn primarily from set theory and predicate logic. These principles were first applied to the field of data modeling in the

late 1960s by Dr. E. F. Codd, then a researcher at IBM, and first published in 1970.¹The rules of the relational model define the way data can be represented (data structure), the way data can be protected (data integrity), and the operations that can be performed on data (data manipulation).

The relational model is not the only method available for storing and manipulating data. Alternatives include the hierarchical, network, and Object/Data models. Each of these models has its advocates, and each has its advantages for certain tasks.

Because of its efficiency and flexibility, the relational model is by far the most popular database technique. Both the Microsoft Jet database engine and Microsoft SQL Server implement the relational model. In general terms, relational database systems have the following characteristics:

- All data is conceptually represented as an orderly arrangement of data into rows and columns, called a relation.
- All values are scalar. That is, at any given row/column position in the relation there is one and only one value.
- All operations are performed on an entire relation and result in an entire relation, a concept known as closure

If you've worked with Microsoft Access databases at all, you'll recognize a "relation" as a 'recordset' or, in SQL Server terms, as a "result set." Dr. Codd, when formulating the relational model, chose the term "relation" because it was comparatively free of connotations, unlike, for example, the word "table." It's a common misconception that the relational model is so called because relationships are established between tables. In fact, the name is derived from the relations on, which it's based.

In fact, relations need not have a physical representation at all. A given relation might map to an actual physical table someplace on a disk, but it can just as well be based on columns drawn from half a dozen different tables, with a few calculated columns which aren't physically stored anywhere. A relation is a relation provided that it's arranged in row and column format and its values are scalar. Its existence is completely independent of any physical representation.

The requirement that all values in a relation be scalar can be somewhat treacherous. The concept of "one value" is necessarily subjective, based as it is on the semantics of the data model. To give a common example, a "Name" might be a single value in one model, but another environment might require that the value be split into "Title", "Given Name", and "Surname", and another might require the addition of "Middle Name" or "Title of Courtesy". None of these is more or less correct in absolute terms; it depends on the use to which the data will be put.

The principle of closure that both base tables and the results of operations are represented conceptually as relation enables the results of one operation to be used as the input to another operation. Thus, with both the Jet database engine and SQL Server we can use the results of one query as the basis for another. This provides database designers with functionality similar to a subroutine in procedural development: the ability to encapsulate complex or commonly performed operations and reuse them whenever and wherever necessary.

You might have created a query called FullNameQuery that concatenates the various attributes representing an individual's name into a single calculated field called FullName. You can create a second query using FullNameQuery as a source that uses the calculated FullName field just like any field that's actually present in the base table. There is no need to recalculate the name.

Relational Terminology

Conceptual	Physical	SQL Server
Relation	Table	Table or result set
Attribute	Field	Column
Tuple	Record	Row

The entire structure is said to be, a relation. Each row of data is a tuple (rhymes with "couple"). Technically, each row is an n-tuple, but the "n-" is usually dropped. The number of tuples in a relation determines its cardinality. In this case, the relation has a cardinality of 18. Each column in the tuple is called an attribute. The number of attributes in a relation determines its degree.

The relation is divided into two sections, the heading and the body. The tuples make up the body, while the heading is composed of, well, the heading. Note that in its relational representation the label for each attribute is composed of two terms separated by a colon.

UnitPrice:Currency. The first part of the label is the name of the attribute, while the second part is its domain. The domain of an attribute is the "kind" of data it represents in this case, currency. A domain is not the same as a data type.

The Data Model

The most abstract level of a database design is the data model, the conceptual description of a problem space. Data models are expressed in terms of entities, attributes, domains, and relationships. The remainder of this chapter discusses each of these in turn.

Entities

It's difficult to provide a precise formal definition of the term entity, but the concept is intuitively quite straightforward: An entity is anything about which the system needs to store information. When you begin to design your data model, compiling an initial list of entities isn't difficult. When you (or your clients) talk about the problem space, most of the nouns and verbs used will be candidate entities. "Customers buy products. Employees sell products. Suppliers sell us products." The nouns "Customers," "Products," "Employees," and "Suppliers" are all clearly entities.

The events represented by the verbs "buy" and "sell" are also entities, but a couple of traps exist here. First, the verb "sell" is used to represent two distinct events: the sale of a product to a customer (Salesman Customer) and the purchase of a product by the organization (Supplier Company). That's fairly obvious in this example, but it's an easy trap to fall into, particularly if you're not familiar with the problem space.

The second trap is the inverse of the first: two different verbs ("buy" in the first sentence and "sell" in the second) are used to describe the same event, the purchase of a product by a customer. Again, this isn't necessarily obvious unless you're familiar with the problem space. This problem is often trickier to track down than the first. If a client is using different verbs to describe what appears to be the same event, they might in fact be describing different kinds of events. If the client is a tailor, for example, "customer buys suit" and "customer orders suit" might both result in the sale of a suit, but in the first case it's a prêt-à-porter sale and in the second it's bespoke. These are very different processes that will probably need to be modeled differently. In addition to interviewing clients to establish a list of entities, it's also useful to review any documents that exist

in the problem space. Input forms, reports, and procedures manuals are all good sources of candidate entities. You must be careful with documents, however. Printed documents have a great deal of inertia input forms particularly are expensive to print and frequently don't keep up with changes to policies and procedures. If you stumble across an entity that's never come up in an interview, don't assume the client just forgot to mention it. Chances are that it's a legacy item that's no longer pertinent to the organization. You'll need to check.

Most entities model objects or events in the physical world: customers, products, or sales calls. These are concrete entities. Entities can also model abstract concepts. The most common example of an abstract entity is one that models the relationship between other entities for example, the fact that a certain sales representative is responsible for a certain client or that a certain student is enrolled in a certain class.

Sometimes all you need to model is the fact that a relationship exists. Other times you'll want to store additional information about the relationships, such as the date on which it was established or some characteristic of the relationship. The relationship between cougars and coyotes is competitive, that between cougars and rabbits is predatory, and it's useful to know this if you're planning an open-range zoo.

Whether relationships that do not have attributes ought to be modeled as separate entities is a matter of some discussion. I personally don't think anything is gained by doing so, and it complicates the process of deriving a database schema from the data model. However, understanding that relationships are as important as entities is crucial for designing an effective data model.

Attributes

Your system will need to keep track of certain facts about each entity. As we've seen, these facts are the entity's attributes. If your system includes a Customer entity, for example, you'll probably want to know the names and addresses of the customers and perhaps the businesses they're in. All of these are attributes.

Determining the attributes to be included in your model is a semantic process. That is, you must make your decisions based on what the data means and how it will be used. Let's look at one common example: an address. Do you model the address as a single entity (the Address) or as a set of entities (HouseNumber, Street, City, State, ZipCode)? Most designers (myself included) would tend to automatically break the address up into a set of attributes on the general principle that structured data is easier to manipulate, but this is not necessarily correct and certainly not straightforward.

Let's take, for instance, a local amateur musical society. It will want to store the addresses of its members in order to print mailing labels. That's the only purpose to which the address will be put, so there is no reason to ever look at an address as anything other than a single, multi-line chunk of text that gets spat out on demand.

But what about a mail-order company that does all its business on the Internet? For sales tax purposes, the company needs to know the states in which its customers reside. While it's possible to extract the state from the single text field used by the musical society, it isn't easy; so it makes sense in this case to at least model the state as a separate attribute. What about the rest of the address? Should it be composed of multiple attributes, and if so, what are they? You might think that a set of attributes {HouseNumber, Street, City, State, ZipCode} might be adequate. But then you need to deal with apartment numbers and post office boxes and APO addresses. What do you do with an address to be sent in care of someone else? And of course the world is getting smaller but not less complex, so what happens when you get your first customer outside the United States? While addresses in the United States conform to a fairly standard pattern, that isn't true when you start dealing with international orders.

Not only do you need to know the country and adjust the format of the zip code (and possibly start calling it a "post code"), but the arrangement of the attributes might need to change. In most of Europe, for example, the house number follows the street name. That's not too bad, but how many of data entry operators are going to know that in the address 4/32 Griffen Avenue, Bondi Beach, Australia, 4/32 means Apartment 4, Number 32?

The point here is not so much that addresses are hard to model, although they are, but rather that you can't make any assumptions about how you should model any specific kind of data. The complex schema that you develop for handling international mail order is completely inappropriate for the local musical society.

Matisse is reputed to have said that a painting was finished when nothing could be either added or subtracted. Entity design is a bit like that. How do you know when you've reached that point?

The unfortunate answer is that you can never know for certain (and even if you think you do, it'll probably change over time). At the current state of technology, there isn't any way to develop a provably correct database design. There are no rules, but there are some strategies.

The first strategy: Start with the result and don't make the design any more complex than it needs to be. What questions does your database have to answer? In our first example, the musical society, the only question was "Where do I mail a letter to this person?" so a single-attribute model was sufficient. The second example, the mail order company, also had to answer "In what state does this person live?" so we needed a different structure to provide the results.

You need to be careful, of course, that you try to provide the flexibility to handle not just the questions your users are asking now but also the ones you can foresee them asking in the future. I'd be willing to bet, for instance, that within a year of implementing the musical society system the society will come back asking you to sort the addresses by zip code so that they can qualify for bulk mail discounts.

You should also be on the lookout for questions the users would ask if they only knew they could, particularly if you're automating a manual system. Imagine asking a head librarian how many of the four million books in the collection were published in Chicago before 1900. He or she would point you to the card file and tell you to have fun. Yet this is trivial information to request from a well-designed database system.

As we saw with the address examples, the more ways you want to slice and dice the data, the more exceptions you have to handle, and you will reach a point of diminishing returns.

This leads me to strategy two: Find the exceptions. There are two sides to this strategy. First, that you must identify all the exceptions, and second, that you must design the system to handle as many exceptions as you can without confusing users. To illustrate what this means, let's walk through another example: personal names.

If your system will be used to produce correspondence, it's crucial that you get the name right. (Case in point: Unsolicited mail arriving at my house addressed to Mr. R. M. Riordan doesn't even get opened.) Most names are pretty straightforward. Ms. Jane Q. Public consists of the Title, FirstName, MiddleInitial, and LastName, right? Wrong. (You saw that coming, didn't you?) To start with, FirstName and LastName are culturally specific. It's safer (and more politically correct) to use GivenName and Surname. Next, what happens to Sir James Peddington Smythe, Lord Dunstable? Is Peddington Smythe his Surname or is Peddington his MiddleName, and what do you do about the "Lord Dunstable" part? And the singer Sting? Is that a GivenName or a Surname? And what will happen to The Artist Formerly Known as Prince? Do you really care? That last question isn't as flippant as it sounds. A letter addressed to Sir James Peddington Smythe probably won't offend anyone. But the gentleman in question is not Sir Smythe; he's Sir James, or maybe Lord Dunstable. Realistically, though, how many of your clients are lords of the realm? The local musical society is not going to thank you for giving them a membership system with a screen.

Remember that there's a trade-off between flexibility and complexity. While it's important to catch as many exceptions as possible, it's perfectly reasonable to eliminate some of them as too unlikely to be worth the cost of dealing with them.

Distinguishing between entities and attributes is sometimes difficult. Again, addresses are a good example, and again, your decision must be based on the problem space. Some designers advocate the creation of a single address entity used to store all the addresses modeled by the system. From an implementation viewpoint, this approach has certain advantages in terms of encapsulation and code reuse. From a design viewpoint, I have some reservations.

It's unlikely, for example, that addresses for employees and customers will be used in the same way. Mass mailings to employees, for example, are more likely to be done via internal mail than the postal service. This being the case, the rules and requirements are different. That awful data entry screen or something very like it might very well be justified for customer addresses, but by using a single address entity you're forced to use it for employees as well, where it's unlikely to be either necessary or appreciated.

Domains

You might recall from the beginning of this chapter that a relation heading contains an AttributeName:DomainName pair for each attribute. More particularly, a domain is the set of all possible values that an attribute may validly contain.

Domains are often confused with data types; they are not the same. Data type is a physical concept while domain is a logical one. "Number" is a data type; "Age" is a domain. To give another example, "StreetName" and "Surname" might both be represented as text fields, but they are obviously different kinds of text fields; they belong to different domains. Take, for example, the domain DegreeAwarded, which represents the degrees awarded by a university. In the database schema, this attribute might be defined as Text[3], but it's not just any three-character string, it's a member of the set {BA, BS, MA, MS, PhD, LLD, MD}.

Of course, not all domains can be defined by simply listing their values. Age, for example, contains a hundred or so values if we're talking about people, but tens of thousands if we're talking about museum exhibits. In such instances it's easier to define the domain in terms of the rules that can be used to determine the membership of any specific value in the set of all valid values rather than listing the values. For example, PersonAge could be defined as "an integer in the range 0 to 120," whereas ExhibitAge might simply be "an integer equal to or greater than 0."

At this point you might be thinking that a domain is the combination of the data type and the validation rule. But validation rules are strictly part of the data integrity, not part of the data description. For example, the validation rule for a zip code might refer to the State attribute, whereas the domain of Zip-Code is "a five-digit string" (or perhaps an eleven-character string, if you're using the Zip + 4 format.)

Note that each of these definitions makes some reference to the kind of data stored (number or string). Data types are physical; they're defined and implemented in terms of the database engine. It would be a mistake to define a domain as varchar(30) or Long Integer, which are engine-specific descriptions.

For any two domains, if it makes sense to compare attributes defined on them (and, by extension, to perform relational operations such as joins), then the two domains are said to be type-compatible. For example, given the two relations, it would be perfectly sensible to link them on EmployeeID = SalespersonID. The domains EmployeeID and SalespersonID are type-compatible. Trying to combine the relations on EmployeeID = OrderDate, however, would probably not result in a meaningful answer, even if the two domains were defined on the same data type.

Unfortunately, neither the Jet database engine nor SQL Server provides strong intrinsic support for domains beyond data types. And even within data types, neither engine performs strong checking; both will quietly convert data behind the scenes. For example, if you're using Microsoft Access and have defined EmployeeID as a long integer in the Employees table and InvoiceTotal as a currency value in the Invoices, you can create a query linking the two tables on EmployeeID = InvoiceTotal, and Microsoft Jet will quite happily give you a list of all employees who have an EmployeeID that matches the total value of an invoice. The two attributes are not type-compatible, but the Jet database engine doesn't know or care.

So why bother with domains at all? Because, they're extremely useful design tools. "Are these two attributes interchangeable?" "Are there any rules that apply to one but don't apply to the other?" These are important questions when you're designing a data model, and domain analysis helps you think about them.

Relationships

In addition to the attributes of each entity, a data model must specify the relationships between entities. At the conceptual level, relationships are simply associations between entities. The statement "Customers buy products" indicates that a relationship exists between the entities Customers and Products. The entities involved in a relationship are called its participants. The number of participants is the degree of the relationship. (The degree of a relationship is similar to, but not the same as, the degree of a relation, which is the number of attributes.)

The vast majority of relationships are binary, like the "Customers buy products" example, but this is not a requirement. Ternary relationships, those with three participants, are also common. Given the binary relationships "Employees sell products" and "Customers buy products," there is an implicit ternary relationship "Employees sell products to customers." However, specifying the two binary relationships does not allow us to identify which employees sold which products to which customers; only a ternary relationship can do that.

A special case of a binary relationship is an entity that participates in a relationship with itself. This is often called the bill of materials relationship and is most often used to represent hierarchical structures. A common example is the relationship between employees and managers: Any given employee might both be a manager and have a manager.

The relationship between any two entities can be one-to-one, one-to-many, or many-to-many. One-to-one relationships are rare, but can be useful in certain circumstances. One-to-many relationships are probably the most common type. An invoice includes many products. A salesperson creates many invoices. These are both examples of one-to-many relationships.

Although not as common as one-to-many relationships, many-to-many relationships are also not unusual and examples abound. Customers buy many products, and products are bought by many customers. Teachers teach many students, and students are taught by many teachers. Many-to-many relationships can't be directly implemented in the relational model, but their indirect implementation is quite straightforward.

The participation of any given entity in a relationship can be partial or total. If it is not possible for an entity to exist unless it participates in the relationship, the participation is total; otherwise, it is partial. For example, Salesperson details can't logically exist unless there is a corresponding Employee. The reverse is not true. An employee might be something other than a salesperson, so an Employee record can exist without a corresponding Salesperson record. Thus, the participation of Employee in the relationship is partial, while the participation of Salesperson is total.

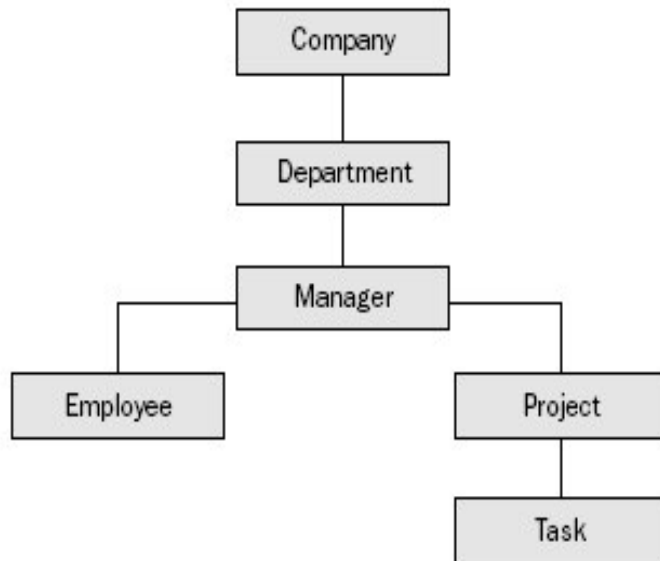
The trick here is to ensure that the specification of partial or total participation holds true for all instances of the entity for all time. It's not unknown for companies to change suppliers for a product, for example. If the participation of Products in the "Suppliers provide products" relation has been defined as total, it won't be possible to delete the current supplier without deleting the product details.

Hierarchical Model

The hierarchical database model is an inverted tree-like structure. The tables of this model take on a child-parent relationship. Each *child table* has a single *parent table*, and each parent table can have multiple child tables. Child tables are completely dependent on parent tables; therefore, a child table can exist only if its parent table does. It follows that any entries in child tables can only exist where corresponding parent entries exist in parent tables. The result of this structure is that the hierarchical database model supports *one-to-many* relationships. Figure shows an

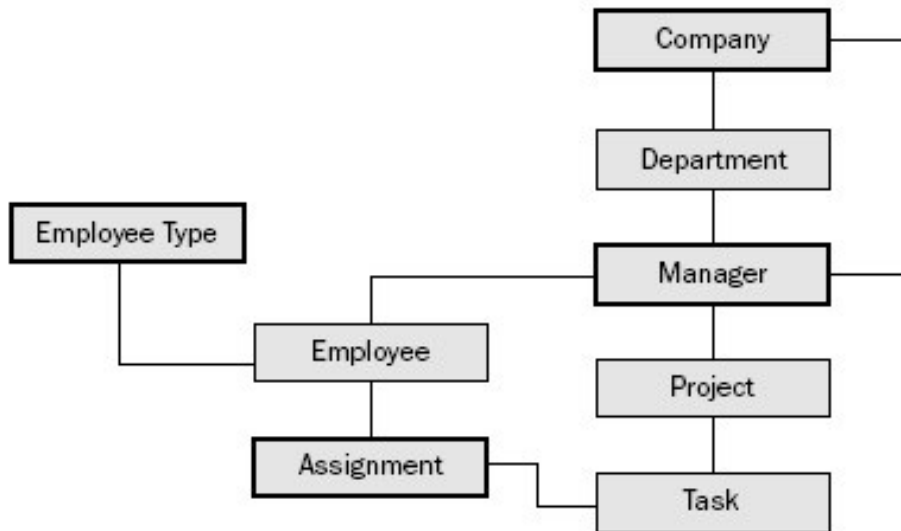
example hierarchical database model. Every task is part of a project, which is part of a manager, which is part of a division, which is part of a company. So, for example, there is a one-to-many relationship between companies and departments because there are many departments in every company.

The disadvantages of the hierarchical database model are that any access must originate at the root node, in the case of Figure, the Company. You cannot search for an employee without first finding the company, the department, the employee's manager, and finally the employee.



Network Model

The network database model is essentially a refinement of the hierarchical database model. The network model allows child tables to have more than one parent, thus creating a networked-like table structure. Multiple parent tables for each child allows for *many-to-many* relationships, in addition to one-to-many relationships. In an example network database model shown in Figure 1-5, there is a many-to-many relationship between employees and tasks. In other words, an employee can be assigned many tasks, and a task can be assigned to many different employees. Thus, many employees have many tasks, and visa versa. Figure shows how the managers can be part of both departments and companies. In other words, the network model in Figure is taking into account that not only does each department within a company have a manager, but also that each company has an overall manager (in real life, a Chief Executive Officer or CEO). Figure also shows the addition of table types where employees can be defined as being of different types (such as full-time, part-time, or contract employees). Most importantly to note from Figure is the new Assignment table allowing for the assignment of tasks to employees. The creation of the Assignment table is a direct result of the addition of the multiple-parent capability between the hierarchical and network models. As already stated, the relationship between the employee and task tables is a many to-many relationship, where each employee can be assigned multiple tasks and each task can be assigned to multiple employees. The Assignment table resolves the dilemma of the many-to-many relationship by allowing a unique definition for the combination of employee and task. Without that unique definition, finding a single assignment would be impossible.



The Relational Model

The relational model is based on a collection of mathematical principles drawn primarily from set theory and predicate logic. These principles were first applied to the field of data modeling in the late 1960s by Dr. E. F. Codd, then a researcher at IBM, and first published in 1970.¹ The rules of the relational model define the way data can be represented (data structure), the way data can be protected (data integrity), and the operations that can be performed on data (data manipulation).

The relational model is not the only method available for storing and manipulating data. Alternatives include the hierarchical, network, and Object/Data models. Each of these models has its advocates, and each has its advantages for certain tasks.

Because of its efficiency and flexibility, the relational model is by far the most popular database technique. Both the Microsoft Jet database engine and Microsoft SQL Server implement the relational model. In general terms, relational database systems have the following characteristics:

- All data is conceptually represented as an orderly arrangement of data into rows and columns, called a relation.
- All values are scalar. That is, at any given row/column position in the relation there is one and only one value.
- All operations are performed on an entire relation and result in an entire relation, a concept known as closure

If you've worked with Microsoft Access databases at all, you'll recognize a "relation" as a 'recordset' or, in SQL Server terms, as a "result set." Dr. Codd, when formulating the relational model, chose the term "relation" because it was comparatively free of connotations, unlike, for example, the word "table." It's a common misconception that the relational model is so called because relationships are established between tables. In fact, the name is derived from the relations on, which it's based.

In fact, relations need not have a physical representation at all. A given relation might map to an actual physical table someplace on a disk, but it can just as well be based on columns drawn from half a dozen different tables, with a few calculated columns which aren't physically stored anywhere. A relation is a relation provided that it's arranged in row and column format and its values are scalar. Its existence is completely independent of any physical representation.

The requirement that all values in a relation be scalar can be somewhat treacherous. The concept of "one value" is necessarily subjective, based as it is on the semantics of the data model. To give a common example, a "Name" might be a single value in one model, but another environment might require that the value be split into "Title", "Given Name", and "Surname", and another might require the addition of "Middle Name" or "Title of Courtesy". None of these is more or less correct in absolute terms; it depends on the use to which the data will be put.

The principle of closure that both base tables and the results of operations are represented conceptually as relation enables the results of one operation to be used as the input to another operation. Thus, with both the Jet database engine and SQL Server we can use the results of one query as the basis for another. This provides database designers with functionality similar to a subroutine in procedural development: the ability to encapsulate complex or commonly performed operations and reuse them whenever and wherever necessary.

You might have created a query called FullNameQuery that concatenates the various attributes representing an individual's name into a single calculated field called FullName. You can create a second query using FullNameQuery as a source that uses the calculated FullName field just like any field that's actually present in the base table. There is no need to recalculate the name.

Entity Relationship Diagrams

The Entity Relationship model, which describes data in terms of entities, attributes, and relations, was introduced by Peter Pin Shan Chen in 1976.^[2] At the same time, he proposed a method of diagramming called Entity Relationship (E/R) diagrams, which has become widely accepted. E/R diagrams use rectangles to describe entities, ellipses for attributes, and diamonds to represent relationships.

[2] Peter Pin Shan Chen, "The Entity Relationship Model Toward a Unified View of Data?" ACM TODS 1, No. 1 (March 1976).

The nature of the relationship between entities (one-to-one, one-to-many, or many-to-many) is represented in various ways. A number of people use 1 and M or 1 and ∞ (representing infinity) to represent one and many.

The great advantage of E/R diagrams is that they're easy to draw and understand. In practice, though, I usually diagram the attributes separately, since they exist at a different level of detail. Generally, one is either thinking about the entities in the model and the relationships between them or thinking about the attributes of a given entity, but rarely thinking about both at the same time.

Entity

It's difficult to provide a precise formal definition of the term entity, but the concept is intuitively quite straightforward: An entity is anything about which the system needs to store information. When you begin to design your data model, compiling an initial list of entities isn't difficult. When you (or your clients) talk about the problem space, most of the nouns and verbs used will be candidate entities. "Customers buy products. Employees sell products. Suppliers sell us products." The nouns "Customers," "Products," "Employees," and "Suppliers" are all clearly entities.

The events represented by the verbs "buy" and "sell" are also entities, but a couple of traps exist here. First, the verb "sell" is used to represent two distinct events: the sale of a product to a customer (Salesman Customer) and the purchase of a product by the organization (Supplier Company). That's fairly obvious in this example, but it's an easy trap to fall into, particularly if you're not familiar with the problem space.

The second trap is the inverse of the first: two different verbs ("buy" in the first sentence and "sell" in the second) are used to describe the same event, the purchase of a product by a customer. Again, this isn't necessarily obvious unless you're familiar with the problem space. This problem is often trickier to track down than the first. If a client is using different verbs to describe what appears to be the same event, they might in fact be describing different kinds of events. If the client is a tailor, for example, "customer buys suit" and "customer orders suit" might both result in the sale of a suit, but in the first case it's a prêt-à-porter sale and in the second it's bespoke. These are very different processes that will probably need to be modeled differently. In addition to interviewing clients to establish a list of entities, it's also useful to review any documents that exist in the problem space. Input forms, reports, and procedures manuals are all good sources of candidate entities. You must be careful with documents, however. Printed documents have a great deal of inertia input forms particularly are expensive to print and frequently don't keep up with changes to policies and procedures. If you stumble across an entity that's never come up in an interview, don't assume the client just forgot to mention it. Chances are that it's a legacy item that's no longer pertinent to the organization. You'll need to check. Most entities model objects or events in the physical world: customers, products, or sales calls.

These are concrete entities. Entities can also model abstract concepts. The most common example of an abstract entity is one that models the relationship between other entities for example, the fact that a certain sales representative is responsible for a certain client or that a certain student is enrolled in a certain class.

Sometimes all you need to model is the fact that a relationship exists. Other times you'll want to store additional information about the relationships, such as the date on which it was established or some characteristic of the relationship. The relationship between cougars and coyotes is competitive, that between cougars and rabbits is predatory, and it's useful to know this if you're planning an open-range zoo.

Whether relationships that do not have attributes ought to be modeled as separate entities is a matter of some discussion. I personally don't think anything is gained by doing so, and it complicates the process of deriving a database schema from the data model. However, understanding that relationships are as important as entities is crucial for designing an effective data model.

Attributes

Your system will need to keep track of certain facts about each entity. As we've seen, these facts are the entity's attributes. If your system includes a Customer entity, for example, you'll probably want to know the names and addresses of the customers and perhaps the businesses they're in. All of these are attributes.

Determining the attributes to be included in your model is a semantic process. That is, you must make your decisions based on what the data means and how it will be used. Let's look at one common example: an address. Do you model the address as a single entity (the Address) or as a set of entities (HouseNumber, Street, City, State, ZipCode)? Most designers (myself included).

Relationship

In addition to the attributes of each entity, a data model must specify the relationships between entities. At the conceptual level, relationships are simply associations between entities. The statement "Customers buy products" indicates that a relationship exists between the entities Customers and Products. The entities involved in a relationship are called its participants. The number of participants is the degree of the relationship. (The degree of a relationship is similar to, but not the same as, the degree of a relation, which is the number of attributes.)

The vast majority of relationships are binary, like the "Customers buy products" example, but this is not a requirement. Ternary relationships, those with three participants, are also common. Given the binary relationships "Employees sell products" and "Customers buy products," there is an implicit ternary relationship "Employees sell products to customers." However, specifying the two

binary relationships does not allow us to identify which employees sold which products to which customers; only a ternary relationship can do that.

A special case of a binary relationship is an entity that participates in a relationship with itself. This is often called the bill of materials relationship and is most often used to represent hierarchical structures. A common example is the relationship between employees and managers: Any given employee might both be a manager and have a manager.

The relationship between any two entities can be one-to-one, one-to-many, or many-to-many. One-to-one relationships are rare, but can be useful in certain circumstances.

One-to-many relationships are probably the most common type. An invoice includes many products. A salesperson creates many invoices. These are both examples of one-to-many relationships.

Although not as common as one-to-many relationships, many-to-many relationships are also not unusual and examples abound. Customers buy many products, and products are bought by many customers. Teachers teach many students, and students are taught by many teachers. Many-to-many relationships can't be directly implemented in the relational model, but their indirect implementation is quite straightforward.

The participation of any given entity in a relationship can be partial or total. If it is not possible for an entity to exist unless it participates in the relationship, the participation is total; otherwise, it is partial. For example, Salesperson details can't logically exist unless there is a corresponding Employee. The reverse is not true. An employee might be something other than a salesperson, so an Employee record can exist without a corresponding Salesperson record. Thus, the participation of Employee in the relationship is partial, while the participation of Salesperson is total.

The trick here is to ensure that the specification of partial or total participation holds true for all instances of the entity for all time. It's not unknown for companies to change suppliers for a product, for example. If the participation of Products in the "Suppliers provide products" relation has been defined as total, it won't be possible to delete the current supplier without deleting the product details.

Candidate Keys and Primary Keys

We have already defined a relation body as an unordered set of zero or more tuples and pointed out that, by definition, each member of a set is unique. This being the case, for any relation there must be some combination of attributes that uniquely identifies each tuple. If the rows cannot be uniquely identified, they do not constitute tuples in relational theory. *This set of one or more attributes is called a candidate key.*

There might be more than one candidate key for any given relation, but it must always be the case that each candidate key uniquely identifies each tuple, not just for any specific set of tuples but for all possible tuples for all time. The inverse of this principle must also be true, by the way. Given any two tuples with the same candidate key, both tuples must represent the same entity. Like so many things in relational design, the identification of a candidate key is semantic. You cannot do it by inspection. Just because some field or combination of attributes is unique for a given set of tuples, you cannot guarantee that it will be unique for all tuples, which it must be to qualify as a candidate key. Once again, you must understand the semantics what the data means of the data model, not what it appears to mean.

Consider the Orders relation shown at the bottom of figure 2 .The CustomerID is unique in the example, but it's extremely unlikely that it will remain that way. After all, most companies rely on repeat business. Despite appearances, the semantics of the model tell us that this field is not a candidate key.

By definition, all relations must have at least one candidate key: the set of all attributes comprising the tuple. Candidate keys can be composed of a single attribute (a simple key) or of multiple attributes (a composite key).

Some people are under the impression that composite keys are somehow incorrect, and that they must add an artificial identifier either an identity or autonumber field to their tables to avoid them.

Nothing could be further from the truth. Artificial keys are often more convenient, as we'll see, but composite keys are perfectly acceptable.

However, an additional requirement of a candidate key is that it must be irreducible, so the set of all attributes is not necessarily a candidate key. In the relation shown in figure 3, the attribute {CategoryName} is a candidate key, as is {Description}, but the set {CategoryName, Description}, although it is unique, is not a candidate key since the Description attribute is unnecessary.

Figure 3 Candidate key must be Irreducible, so {category Name} Qualifies, but {category Name, Description} Does not

Category Name	Description
Beverages	Soft drinks, coffees, teas, beers, and ales
Condiments	Sweet and savory sauces, relishes, spreads, and seasonings
Confections	Desserts, candies, and sweet breads
Dairy Products	Cheeses
Grains/Cereals	Breads, crackers, pasta, and cereal
Meat/Poultry	Prepared meats
Produce	Dried fruit and bean curd
Seafood	Seaweed and fish

It is sometimes the case although it doesn't happen often that there are multiple possible candidate keys for a relation. In this case, it is customary to designate one candidate key as a primary key and consider other candidate keys alternate keys. This is an arbitrary decision and isn't very useful at the logical level. (Remember that the data model is purely abstract.)

When the only possible candidate key is unwieldy it requires too many attributes or is too large, for example you can use an artificial unique field data type for creating artificial keys with values that will be generated by the system.

Called AutoNumber fields in Microsoft Jet and Identity fields in SQL Server, fields based on this data type are useful tools, provided you don't try to make them mean anything. They're just tags. They aren't guaranteed to be sequential, you have very little control over how they're generated, and if you try to use them to mean anything you'll cause more problems than you solve.

Although choosing candidate keys is, as we've seen, a semantic process, don't assume that the attributes you use to identify an entity in the real world will make an appropriate candidate key. Individuals, for example, are usually referred to by their names, but a quick look at any phone book will establish that names are hardly unique.

Of course, the name must provide a candidate key when combined with some other set of attributes, but this can be awkward to determine. I once worked in an office with about 20 people, of whom two were named Larry Simon and one was named Lary Simon. All three were Vice Presidents. Amongst ourselves, they were "Short Lary," "German Lary," and "Blond Lary"; that's height, nationality, and hair color combined with name, hardly a viable candidate key.

In situations like this, it's probably best to use a system-generated ID number, such as an Autonumber or Identity field, but remember, don't try to make it mean anything. You need to be careful, of course, that your users are adding apparent duplicates intentionally, but it's best to do this as part of the user interface rather than imposing artificial (and ultimately unnecessary) constraints on the data the system can maintain.

Primary key

PRIMARY KEY and UNIQUE Constraints

A central tenet of the relational model is that every row in a table is in some way unique and can be distinguished in some way from every other row in the table. You could use the combination of all columns in a table as this unique identifier, but the identifier is usually at most the combination of a handful of columns, and often it's just one column: the primary key. Although some tables might have multiple unique identifiers, each table can have only one primary key. For example, perhaps the *employee* table maintains both an *Emp_ID* column and an *SSN* (social security number) column, both of which can be considered unique. Such column pairs are often referred to as *alternate keys* or *candidate keys*, although both terms are design terms and aren't used by the ANSI SQL standard or by SQL Server. In practice, one of the two columns is logically promoted to primary key using the PRIMARY KEY constraint, and the other is usually declared by a UNIQUE constraint. Although neither the ANSI SQL standard nor SQL Server require it, it's good practice to declare a PRIMARY KEY constraint on every table. Furthermore, you must designate a primary key for a table that will be published for transaction-based replication. Internally, PRIMARY KEY and UNIQUE constraints are handled almost identically, so it will be discussed together here. Declaring a PRIMARY KEY or UNIQUE constraint simply results in a unique index being created on the specified column or columns, and this index enforces the column's uniqueness in the same way that a unique index created manually on a column would. The query optimizer makes decisions based on the presence of the unique index rather than on the fact that a column was declared as a primary key. How the index got there in the first place is irrelevant to the optimizer.

Foreign key

FOREIGN KEY Constraints

One of the fundamental concepts of the relational model is the logical relationships between tables. In most databases, certain relationships must exist (that is, the data must have referential integrity) or else the data will be logically corrupt.

SQL Server automatically enforces referential integrity through the use of FOREIGN KEY constraints. This feature is sometimes referred to as declarative referential integrity, or DRI, to distinguish it from other features, such as triggers, that you can also use to enforce the existence of the relationships.

Referential Actions

The full ANSI SQL-92 standard contains the notion of the *referential action*, sometimes (incompletely) referred to as a *cascading delete*. SQL Server 7 doesn't provide this feature as part of FOREIGN KEY constraints, but this notion warrants some discussion here because the capability exists via triggers.

The idea behind referential actions is this: sometimes, instead of just preventing an update of data that would violate a foreign key reference, you might be able to perform an additional, compensating action that will still enable the constraint to be honored. For example, if you were to delete a *customer* table that has references to *orders*, you can have SQL Server automatically delete all those related *order* records (that is, cascade the delete to *orders*), in which case the constraint won't be violated and the *customer* table can be deleted. This feature is intended for both UPDATE and DELETE statements, and four possible actions are defined: NO ACTION, CASCADE, SET DEFAULT, and SET NULL.

- **NO ACTION** The delete is prevented. This default mode, per the ANSI standard, occurs if no other action is specified. NO ACTION is often referred to as RESTRICT, but this usage is slightly incorrect in terms of how ANSI defines RESTRICT and NO ACTION. ANSI uses RESTRICT in DDL statements such as DROP TABLE, and it uses NO

- ACTION for FOREIGN KEY constraints. (The difference is subtle and unimportant. It's common to refer to the FOREIGN KEY constraint as having an action of RESTRICT.)
- **CASCADE** A delete of all matching rows in the referencing table occurs.
 - **SET DEFAULT** The delete is performed, and all foreign key values in the referencing table are set to a default value.
 - **SET NULL** The delete is performed, and all foreign key values in the referencing table are set to NULL.

Introduction to Normalization

Basic Principles

The process of structuring the data in the problem space to achieve the two goals i.e. eliminating redundancy and ensuring flexibility is called *normalization*. The principles of normalization re tools for controlling the structure of data in the same way that a paper clip controls sheets of paper. The normal forms (we'll discuss six) specify increasingly stringent rules for the structure of relations. Each form extends the previous one in such a way as to prevent certain kinds of update anomalies.

Bear in mind that the normal forms are not a prescription for creating a "correct" data model. A data model could be perfectly normalized and still fail to answer the questions asked of it; or, it might provide the answers, but so slowly and awkwardly that the database system built around it is unusable. But if your data model is normalized that is, if it conforms to the rules of relational structure the chances are high that the result will be an efficient, effective data model.

Before we turn to normalization, however, you should be familiar with a couple of underlying principles.

Lossless Decomposition

The relational model allows relations to be joined in various ways by linking attributes. The process of obtaining a fully normalized data model involves removing redundancy by dividing relations in such a way that the resultant relations can be recombined without losing any of the information. This is the principle of lossless decomposition.

From the given relation in figure1, you can derive two relations as shown in figure 2

Figure 1 An Unnormalized Relation

Order ID	Order Date	Required Date	Customer ID	Company Name	Address	City
10653	27-Jan-1998	24-Feb-1998	BLAUS	Blauer See Delikatessen	Forsterstr. 57	Mannheim
10905	24-Feb-1998	24-Mar-1998	WELLI	Wellington Importadora	Rua do Mercado, 12	Resende
10953	16-Mar-1998	30-Mar-1998	AROUT	Around the Horn	120 Hanover Sq.	London
11016	10-Apr-1998	08-May-1998	AROUT	Around the Horn	120 Hanover Sq.	London
10736	11-Nov-1997	09-Dec-1997	HUNGO	Hungry Owl All-Night Grocers	8 Johnstown Road	Cork
11022	14-Apr-1998	12-May-1998	HANAR	Hanari Carnes	Rua do Paço, 67	Rio de Janeiro
10577	23-Jun-1997	04-Aug-1997	TRAIH	Trail's Head Gourmet Provisioners	722 DaVinci Blvd.	Kirkland
10750	21-Nov-1997	19-Dec-1997	WARTH	Wartian Herkku	Tonikatu 38	Oulu
10324	08-Oct-1996	05-Nov-1996	SAVEA	Save-a-lot Markets	187 Suffolk Ln.	Boise

Figure 2 Can be divided into These 2 relations without losing any information

Customer ID	Company Name	Address	City
ALFKI	Alfreds Futterkiste	Obere Str. 57	Berlin
ANATR	Ana Trujillo Emparedados y helados	Avda. de la Constitución 2222	México D.F.
ANTON	Antonio Moreno Taquería	Mataderos 2312	México D.F.
AROUT	Around the Horn	120 Hanover Sq.	London
BERGS	Berglunds snabbköp	Berguvsvägen 8	Luleå
BLAUS	Blauer See Delikatessen	Forsterstr. 57	Mannheim
BLONP	Blondel père et fils	24, place Kléber	Strasbourg
BOLID	Bólido Comidas preparadas	C/ Araquil, 67	Madrid

Order ID	Order Date	Required Date	Customer
11077	06-May-1998	03-Jun-1998	RATTC
11076	06-May-1998	03-Jun-1998	BONAP
11075	06-May-1998	03-Jun-1998	RICSU
11074	06-May-1998	03-Jun-1998	SIMOB
11073	05-May-1998	02-Jun-1998	PERIC
11072	05-May-1998	02-Jun-1998	ERNSH
11071	05-May-1998	02-Jun-1998	LILAS
11070	05-May-1998	02-Jun-1998	LEHMS
11069	04-May-1998	01-Jun-1998	TORTU

The concept of functional dependency is an extremely useful tool for thinking about data structures. Given any tuple T , with two sets of attributes $\{X_1...X_n\}$ and $\{Y_1...Y_n\}$ (the sets need not be mutually exclusive), then set Y is functionally dependent on set X if, for any legal value of X , there is only one legal value for Y .

For example, in the relation shown in Figure 3, every tuple that has the same values for $\{CategoryName\}$ will have the same value for $\{Description\}$. We can therefore say that the attribute $CategoryName$ functionally determines the attribute $Description$. Note that functional dependency doesn't necessarily work the other way: knowing a value for $Description$ won't allow us to determine the corresponding value for $CategoryID$. You can indicate the functional dependency between sets of attributes as shown in figure 4

In text, you can express functional dependencies as $X \twoheadrightarrow Y$, which reads "X functionally determines Y."

figure 4 Functional Dependency Diagram Are Largely self - Explanatory

Functional dependency is interesting to academics because it provides a mechanism for developing something that starts to resemble a mathematics of data modeling. You can, for example, discuss the reflexivity and transitivity of a functional dependency if you are so inclined. In practical application, functional dependency is a convenient way of expressing what is a fairly self-evident concept: Given any relation, there will be some set of attributes that is unique to each tuple, and knowing those, it is possible to determine those attributes that are not unique.

Thus, given the tuple {X, Y}, if {X} is a candidate key, then all attributes {Y} must necessarily be functionally dependent on {X}; this follows from the definition of candidate key. If {X} is not a candidate key and the functional dependency is not trivial (that is, {Y} is not a subset of {X}), then the relation will necessarily involve some redundancy, and further normalization will be required. To return to the example shown in figure 4, by knowing the Category Name, we can easily determine the Category Description.

First Normal Form

A relation is in first normal form if the domains on which its attributes are defined are scalar. This is at once both the simplest and most difficult concept in data modeling. The principle is straightforward: Each attribute of a tuple must contain a single value. But what constitutes a single value? In the relation shown in figure 5, the Items attribute obviously contains multiple values and is therefore not in first normal form. But the issue is not always so clear cut.

Figure 5 First Normal Form

OrderID	CustomerID	OrderDate	Items	OrderTotal
1	CACTU	1/1/1999	3 Zaanse koeken, 1 Tarte au sucre	\$89.70
2	BSBEV	1/5/1999	4 Mozzarella di Giovanni	\$139.20
3	SUPRD	5/2/1999	3 Ravioli Angelo, 6 Tofu	\$198.06

We saw some of the problems involved in determining whether an attribute is scalar when we looked at the modeling of names and addresses earlier. Dates are another tricky domain. They consist of three distinct components: the day, the month, and the year. Ought they be stored as three attributes or as a composite? As always, the answer can be determined only by looking to the semantics of the problem space you're modeling.

If your system most often uses all three components of a date together, it is scalar. But if your system must frequently manipulate the individual components of the date, you might be better off storing them as separate attributes. You might not care about the day, for example, but only the month and year.

In the specific instance of dates, because date arithmetic is tedious to perform, it will often make your life easier if you use an attribute defined on the DateTime data type, which combines all three components of the date and the time. The DateTime data types allow you to offload the majority of the work involved in, for example, determining the date 37 days from today, to the development environment.

Another place people frequently have problems with non-scalar values is with codes and flags. Many companies assign case numbers or reference numbers that are calculated values, usually something along the lines of REF0010398, which might indicate that this is the first case opened in March 1998. While it's unlikely that you'll be able to alter company policy, it's not a good idea to attempt to manipulate the individual components of the reference number in your data model. It's far easier in the long run to store the values separately: {Reference#, Case#, Month, Year}. This way, determining the next case number or the number of cases opened in a given year becomes a simple query against an attribute and doesn't require additional manipulation. This has important performance implications, particularly in client/server environments, where extracting a value from the middle of an attribute might require that each individual record be examined locally (and transferred across the network) rather than by the database server. Another type of non-scalar attribute that causes problems for people is the bit flag. In conventional programming environments, it's common practice to store sets of Boolean values as individual bits in a word, and then to use bitwise operations to check and test them. Windows API programming relies heavily on this technique, for example. In conventional programming environments, this is a

perfectly sensible thing to do. In relational data models, it is not. Not only does the practice violate first normal form, but it's extraordinarily tedious and, as a general rule, inefficient.

There's another kind of non-scalar value to be wary of when checking a relation for first normal form: the repeating group. Figure 6 shows an Invoice relation. Someone, at some point, decided that customers are not allowed to buy more than three items. This is almost certainly an artificial constraint imposed by the system, not the business. Artificial system constraints are evil, and in this case, just plain wrong as well.

OrderID	CustomerID	Item1	Qty1	Item2	Qty2	Item3	Qty3
1	ANTON	Queso Cabrales	4	Tofu	3	Ravioli Angelo	1
2	BLAUS	Chai	2		0		

Figure 6 This Data Model Restricts the number of Items a Customer Can Purchase

Figure 7 This is repeating Group

Another example of a repeating group is shown in figure 7. This isn't as obvious an error, and many successful systems have been implemented using a model similar to this. But this is really just a variation of the structure shown in figure 6 and has the same problems. Imagine the query to determine which products exceeded target by more than 10 percent any time in the first quarter.

Product	Year	TargetJan	ActualJan	TargetFeb	ActualFeb
Aniseed Syrup	2004	\$1,000.00	\$1,300.00	\$0.00	\$0.00
Chai	2004	\$4,000.00	\$2,000.00	\$0.00	\$0.00
Chang	2004	\$3,000.00	\$8,022.00	\$0.00	\$0.00

Second Normal Form

A relation is in second normal form if it is in first normal form and, in addition, all its attributes are dependent on the entire candidate key. The key in figure 8, for example, is {ProductName, SupplierName}, but the SupplierPhoneNumber field is dependent only on the Supplier-Name, not on the full composite key.

Figure 8 All Attributes in Relation should Depend on the Whole Key

Product Name	SupplierName	Category Name	SupplierPhoneNumber
Chai	Exotic Liquids	Beverages	(171) 555-2222
Chang	Exotic Liquids	Beverages	(171) 555-2222
Guaraná Fantástica	Refrescos Americanas LTDA	Beverages	(11) 555 4640
Sasquatch Ale	Bigfoot Breweries	Beverages	(503) 555-9931
Steeleye Stout	Bigfoot Breweries	Beverages	(503) 555-9931
Côte de Blaye	Aux joyeux ecclésiastiques	Beverages	(1) 03.83.00.68
Chartreuse verte	Aux joyeux ecclésiastiques	Beverages	(1) 03.83.00.68
Ipoh Coffee	Leka Trading	Beverages	555-8787
Laughing Lumberjack Lager	Bigfoot Breweries	Beverages	(503) 555-9931
Outback Lager	Pavlova, Ltd.	Beverages	(03) 444-2343

in unpleasant maintenance problems.

Figure 9 These two Relation are in second normal Form

Product ID	Product Name	Category
1	Chai	Beverages
2	Chang	Beverages
3	Aniseed Syrup	Condiments
4	Chef Anton's Cajun Seasoning	Condiments
5	Chef Anton's Gumbo Mix	Condiments
6	Grandma's Boysenberry Spread	Condiments
7	Uncle Bob's Organic Dried Pears	Produce

Supplier ID	SupplierName	SupplierPhoneNumber
1	Exotic Liquids	(171) 555-2222
2	New Orleans Cajun Delights	(100) 555-4822
3	Grandma Kelly's Homestead	(313) 555-5735
4	Tokyo Traders	(03) 3555-5011
5	Cooperativa de Quesos 'Las Cabras'	(98) 598 76 54
6	Mayumi's	(06) 431-7877
7	Pavlova, Ltd.	(03) 444-2343
8	Specialty Biscuits, Ltd.	(161) 555-4448
9	PB Knäckebröd AB	031-987 65 43

A better model would be that shown in figure 9. Logically, this is an issue of not trying to represent two distinct entities, Products and Suppliers, in a single relation. By separating the representation, you're not only eliminating the redundancy, you're also providing a mechanism for storing information that you couldn't otherwise capture. In the example in figure 9, it becomes possible to capture information about Suppliers before obtaining any information regarding their products. That could not be done in the first relation, since neither component of a primary key can be empty. The other way that people get into trouble with second normal form is in confusing constraints that happen to be true at any given moment with those that are true for all time.

The relation shown in figure 10, for example, assumes that a supplier has only one address, which might be true at the moment but will not necessarily remain true in the future.

Figure 10 Suppliers might have more than one Address

Supplier ID	Address	City	Region	Postal Code
1	49 Gilbert St.	London		EC1 4SD
2	P.O. Box 78934	New Orleans	LA	70117
3	707 Oxford Rd.	Ann Arbor	MI	48104
4	9-8 Sekimai	Tokyo		100
5	Calle del Rosal 4	Oviedo	Asturias	33007
6	92 Setsuko	Osaka		545
7	74 Rose St.	Melbourne	Victoria	3058
8	29 King's Way	Manchester		M14 GSD
9	Kalodagatan 13	Göteborg		S-345 67
10	Av. das Americanas 12.890	São Paulo		5442

Third Normal Form

A relation is in third normal form if it is in second normal form and in addition all non-key attributes are mutually independent. Let's take the example of a company that has a single salesperson in each state. Given the relation shown in figure 11, there is a dependency between Region and Salesperson, but neither of these attributes is reasonably a candidate key for the relation.

Figure 11 Although Mutually Dependent, neither Region nor Salesperson should be a Candidate key

Order ID	Company Name	Region	Salesperson
11076	Bon app'	WA	Margaret Peacock
11077	Rattlesnake Canyon Grocery	WA	Nancy Davolio
11075	Richter Supermarkt	WA	Laura Callahan
11074	Simons bistro		Robert King
11071	LILA-Supermercado	WA	Nancy Davolio
11072	Ernst Handel	WA	Margaret Peacock

It's possible to get really pedantic about third normal form. In most places, for example, you can determine a PostalCode value based on the City and Region values, so the relation shown in figure 12 is not strictly in third normal form.

Figure 12 This Relationship is not in Strict Third Normal Form

Company Name	Address	City	Region	Postal Code
Exotic Liquids	49 Gilbert St.	London		EC1 4SD
New Orleans Cajun Delights	P.O. Box 78934	New Orleans	LA	70117
Grandma Kelly's Homestead	707 Oxford Rd.	Ann Arbor	MI	48104
Tokyo Traders	9-8 Sekimai	Tokyo		100
Cooperativa de Quesos 'Las Cabras'	Calle del Rosal 4	Oviedo	Asturias	33007
Mayumi's	92 Setsuko	Osaka		545
Pavlova, Ltd.	74 Rose St.	Melbourne	Victoria	3058

The two relations shown in figure 13 are technically more correct, but in reality the only benefit you're gaining is the ability to automatically look up the Postal Code when you're entering new records, saving users a few keystrokes. This isn't a trivial benefit, but there are probably better

ways to implement this functionality, ones that don't incur the overhead of a relation join every time the address is referenced.

Figure 13 These 2 relations are in Third Normal Form

Company Name	Address	City	Region
Exotic Liquids	49 Gilbert St.	London	
New Orleans Cajun Delights	P.O. Box 78934	New Orleans	LA
Grandma Kelly's Homestead	707 Oxford Rd.	Ann Arbor	MI
Tokyo Traders	9-8 Sekimai	Tokyo	
Cooperativa de Quesos 'Las Cabras'	Calle del Rosal 4	Oviedo	Asturias
Mayumi's	92 Setsuko	Osaka	
Pavlova, Ltd.	74 Rose St.	Melbourne	Victoria

City	Region	Postal Code
Melbourne	Victoria	3058
Ste-Hyacinthe	Québec	J2S 7S8
Montréal	Québec	H1J 1C3
Bend	OR	97101
Sydney	NSW	2042
Ann Arbor	MI	48104
Boston	MA	02134
New Orleans	LA	70117
Oviedo	Asturias	33007

As with every other decision in the data modeling process, when and how to implement third normal form can only be determined by considering the semantics of the model. It's impossible to give fixed rules, but there are some guidelines:

You should create a separate relation only when

- The entity is important to the model, or
- The data changes frequently, or
- You're certain there are technical implementation advantages

Postal codes do change, but not often; and they aren't intrinsically important in most systems. In addition, a separate postal code table is impractical in most real-world applications because of the varying rules for how postal codes are defined.

Further Normalisation

The first three normal forms were included in Codd's original formulation of relational theory, and in the vast majority of cases they're all you'll need to worry about. The further normal forms Boyce/Codd, fourth, and fifth have been developed to handle special cases, most of which are rare.

Boyce/Codd Normal Form

Boyce/Codd normal form, which is considered a variation of third normal form, handles the special case of relations with multiple candidate keys. In fact, for Boyce/Codd normal form to apply, the following conditions must hold true:

- The relation must have two or more candidate keys.
- At least two of the candidate keys must be composite.

- The candidate keys must have overlapping attributes.

The easiest way to understand Boyce/Codd normal form is to use functional dependencies. Boyce/Codd normal form states, essentially, that there must be no functional dependencies between candidate keys. Take, for example, the relation shown in figure 14. The relation is in third normal form (assuming supplier names are unique), but it still contains significant redundancy.

Figure 14 This Relation is in Third Normal form but not in Boyce/Codd normal Form

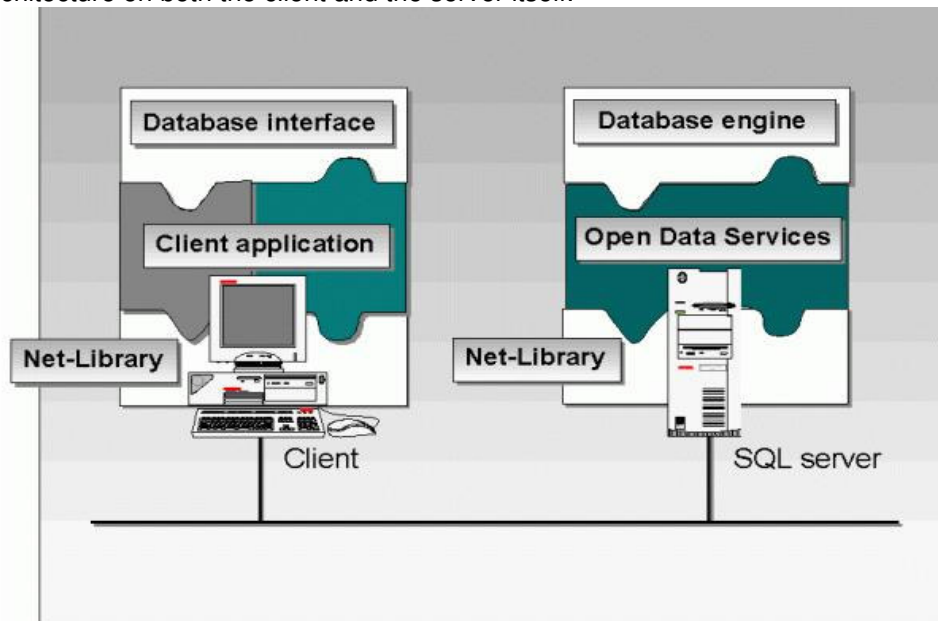
The two candidate keys in this case are {SupplierID, ProductID} and {SupplierName, ProductID}, and the functional dependency diagram is shown in figure 15

Supplier ID	SupplierName	Product	Quantity	Unit Price
5	Cooperativa de Queses 'Las Cabras'	Queso Cabrales	12	\$14.00
20	Leka Trading	Singaporean Hokkien Fried Mee	10	\$9.80
14	Fornaggi Fortini s.r.l.	Mozzarella di Giovanni	5	\$34.80
24	G'day, Mate	Manjimup Dried Apples	40	\$42.40
6	Mayumi's	Tofu	9	\$18.60
24	G'day, Mate	Manjimup Dried Apples	35	\$42.40
19	New England Seafood Cannery	Jack's New England Clam Chowder	10	\$7.70
2	New Orleans Cajun Delights	Louisiana Fiery Hot Pepper Sauce	15	\$16.80

SQL Server Database Architecture

Introduction

Because SQL Server is a client/server database management system, you will find components of its architecture on both the client and the server itself.



Client-Server Architecture

Client Architecture

On the client, the SQL Server architecture consists of the client application, a database interface, and a Net-Library. Clients use applications such as the SQL Server utilities, Microsoft Access, or a custom application to access the SQL Server. The client application uses a *database interface* to connect to and access resources on the SQL server. The database interface consists of an application programming interface (API) and a data object interface. SQL Server supports two classes of APIs: Object Linking And Embedding Database (OLE DB) and Open Database Connectivity (ODBC). The *OLE DB API* uses the Component Object Model (COM)-based interface and is modular. In contrast, the *ODBC API* uses calls to access the SQL Server directly via the Tabular Data Stream (TDS) protocol. While OLE DB can be used to access many different types of data sources, ODBC can be used to access data only in relational databases.

SQL Server supports two data object interfaces: ActiveX Data Objects and Remote Data Objects. *ActiveX Data Objects (ADO)* enable you to encapsulate the OLE DB API commands in order to reduce application development time. You can use ADO from Visual Basic, Visual Basic for Applications, Active Server Pages, and the Microsoft Internet Explorer scripting object model. *Remote Data Objects (RDO)* enable you to encapsulate the ODBC API. You can use RDO from Visual Basic and Visual Basic for Applications. As a programmer, the API you choose dictates which data object interface you can use (and vice versa). For example, if you use the OLE DB API, you use ActiveX Data Objects. In contrast, if you use the ODBC API, you must use Remote Data Objects.

At the lowest layer of the client architecture, you connect to the SQL server by using a Net-Library. You use Net-Libraries to prepare read and write requests for sending by the appropriate network protocol. You can configure both the client and the SQL server to use more than one Net-Library. Note: Both the client and the server must have a Net-Library in common. SQL Server supports TCP/IP, Named Pipes, IPX/SPX (NWLink), Banyan Vines, and AppleTalk ADSP.

Server Architecture

On the server, the SQL Server architecture consists of the SQL Server database engine, Open Data Services, and the server's Net-Library. The SQL Server database engine actually processes client requests. The database engine consists of two components: the relational engine and the storage engine. The relational engine is responsible for parsing and optimizing queries; the relational engine retrieves data from the storage engine. The storage engine is responsible for retrieving and modifying data on your server's hard drives. Open Data Services manages clients' connections to the server. Specifically, the server receives client requests and responds through Open Data Services. The server's Net-Library accepts connection requests from a client's Net-Library.

Tabular Data Stream

SQL Server uses the Tabular Data Stream (TDS) protocol to send data between the client and the server. TDS packets, in turn, are then encapsulated in the protocol stack used by the Net-Library. For example, if you are using SQL Server on a TCP/IP-based network, TDS packets are encapsulated in TCP/IP packets.

Administration Architecture



SQL Server uses the SQL Distributed Management Objects (SQL-DMO) to write all of its administrative utilities. SQL-DMO is a collection of Component Object Model (COM)-based objects that are used by the administrative tools. SQL-DMO essentially enables you to create administrative tools that hide the Transact SQL statements from the user. All of the SQL Server graphical administrative tools use SQL-DMO. For example, you can create a database within SQL Server Enterprise Manager without ever having to know or understand the CREATE DATABASE Transact SQL-statement.

Application Architecture

When you design an application, there are several different client/server architectures you can choose from. These architectures vary as to how much of the data processing is done by the SQL server as compared to the client. Before you look at the architectures, it is important that you understand that all client/server applications consist of three layers:

- *Presentation*- The user interface (this layer usually resides on the client).
- *Business*- The application's logic and rules for working with the data (this layer can be on the server, client, or both).
- *Data*- The actual database itself, its rules for database integrity, and stored procedures(this layer is typically only on the server).

Application architectures are usually categorized based on the number of computers that are involved in the application. For example, an application that consists of a portion running on the server and on the client is usually referred to as 2-tier. The following table describes the different application architectures.

Application Architecture	Description
Intelligent Server (2-tier)	An application that resides on both the server and the client, but the majority of the processing is performed by the server. Only the presentation layer resides on the client.
Intelligent Client (2-tier)	An application that resides on both the server and the client, but the majority of the processing is performed by the client. Only the data layer resides on the server.  Microsoft Access.
N-tier	An application that resides on the database server (the data layer), an application server (the Business layer) and the client (the presentation layer)
Internet	An application where the Business and Presentation layers reside on a Web server, the Data layer on a Database server and the client uses a Web browser to access the information .  Web sites that uses SQL Databases

1. What components make up the database interface in the SQL Server architecture?

The database interface consists of an API and a data object interface. SQL Server supports the OLE DB and ODBC APIs, and the ActiveX Data Objects and Remote Data Objects data object interfaces.

2. What are the three layers in all client/server applications?

All client/server applications consist of the Presentation layer, the Business layer and the Data layer.

Logical Architecture

The SQL Server Engine

Figure 3-1 shows the general architecture of SQL Server. For simplicity, I've made some minor omissions and simplifications and ignored certain "helper" modules. Now let's look in detail at the major modules.

The Net-Library

The Net-Library (often called Net-Lib, but in this book I'll use Net-Library) abstraction layer enables SQL Server to read from and write to many different network protocols, and each such protocol (such as TCP/IP sockets) can have a specific driver. The Net-Library layer makes it relatively easy to support many different network protocols without having to change the core server code.

A Net-Library is basically a driver that's specific to a particular network interprocess communication (IPC) mechanism. (Be careful not to confuse *driver* with *device driver*.) All code in SQL Server, including Net-Library code, makes calls only to the Microsoft Win32 subsystem. SQL Server uses a common internal interface between Microsoft Open Data Services (ODS)—which manages its use of the network—and each Net-Library. If your development project needs to support a new and different network protocol, you can handle all network-specific issues by simply writing a new Net-Library. In addition, you can load multiple Net-Libraries simultaneously, one for each network IPC mechanism in use.

SQL Server uses the Net-Library abstraction layer on both the server and client machines, making it possible to support several clients simultaneously on different networks. Microsoft Windows NT/2000 and Windows 98 support the simultaneous use of multiple protocol stacks. Net-Libraries are paired. For example, if a client application is using a Named Pipes Net-Library, SQL Server must also be listening on a Named Pipes Net-Library. The client application determines which Net-Library is actually used for the communication, and you can control the client application's choice by using a tool called the Client Network Utility. You can easily configure SQL Server to listen on multiple Net-Libraries by using the Server Network Utility, which is available under Programs\Microsoft SQL Server on the Start menu.

SQL Server 2000 has two primary Net-Libraries: Super Socket and Shared Memory. TCP/IP, Named Pipes, IPX/SPX, and so on are referred to as secondary Net-Libraries. The OLE DB Provider for SQL Server, SQL Server ODBC driver, DB-Library, and the database engine communicate directly with these two primary network libraries. Intercomputer connections communicate through the Super Socket Net-Library. Local connections between an application and a SQL Server instance on the same computer use the Shared Memory Net-Library if Shared Memory support has been enabled (which it is, by default). SQL Server 2000 supports the Shared Memory Net-Library on all Windows platforms.

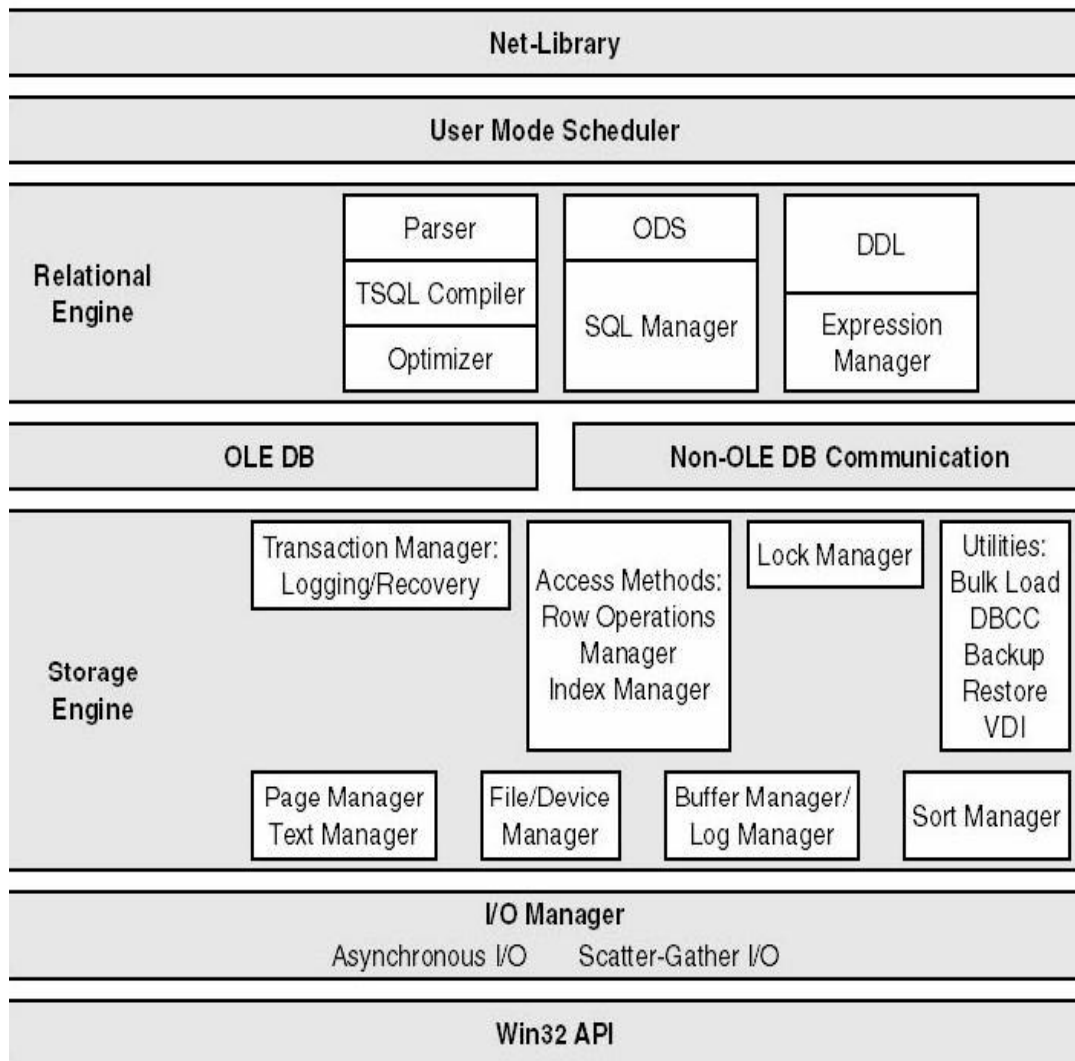


Figure 3-1. The major components of the SQL Server architecture.

The Super Socket Net-Library has two components:

Communication path

If the client is configured to communicate over TCP/IP Sockets or NWLink IPX/SPX connection, the Super Socket Net-Library directly calls the Windows Socket 2 API for the communication between the application and the SQL Server instance.

If the client is configured to communicate over a Named Pipes, Multiprotocol, AppleTalk, or Banyan VINES connection, a subcomponent of the Super Socket Net-Library called the Net-Library router loads the secondary Net-Library for the chosen protocol and routes all Net-Library calls to it.

Encryption layer

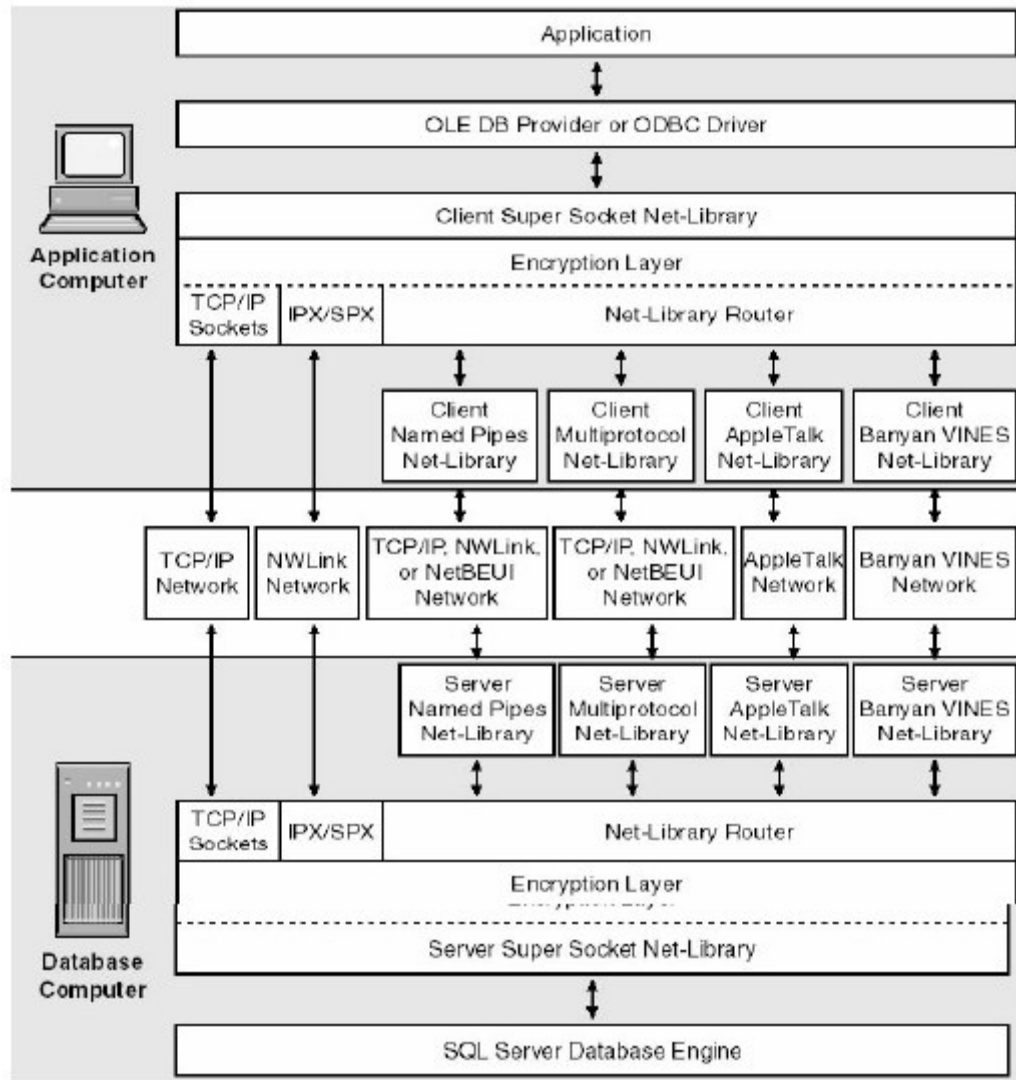
The encryption is implemented using the Secure Sockets Layer (SSL) API. The level of encryption, 40-bit or 128-bit, depends on the Windows version on the application and SQL Server-based computers. Enabling encryption can slow network performance not only because of the extra work of encrypting and decrypting communication between the client and the server but also because an extra roundtrip is required between the client and the server every time a connection is made.

Shared Memory Net-Library communication is inherently secure without the need for encryption. The Shared Memory Net-Library never participates in intercomputer communication. The area of memory shared between the application process and the database engine process cannot be accessed from any other Windows process.

For compatibility with earlier versions of SQL Server, the Multiprotocol Net-Library continues to support its own encryption. This encryption is specified independently of the SSL encryption and is implemented by calling the Windows RPC encryption API.

Figure 3-2 (taken from *SQL Server Books Online*) shows SQL Server 2000's Net-Library architecture.

The distinction between the IPC mechanisms and the underlying network protocols is important. IPC mechanisms used by SQL Server include Named Pipes, RPC, SPX, and Windows Sockets. Network protocols used include TCP/IP, NetBEUI, Shared Memory, NWLink IPX/SPX, Banyan VINES SPP, and AppleTalk ADSP. Two Net-Libraries, Multiprotocol and Named Pipes, can be used simultaneously over multiple network protocols (NetBEUI, NWLink IPX/SPX, and TCP/IP). You can have multiple network protocols in your environment and still use only one Net-Library. SQL Server 2000 running on Windows NT/2000 supports impersonation of security contexts to provide an integrated logon authentication capability called Windows Authentication. Windows Authentication operates over network protocols that support authenticated connections between clients and servers. Such connections are referred to as *trusted connections* and are supported by SQL Server 2000 using any available Net-Library. Instead of requiring a separate user ID/password logon each time a connection to SQL Server is requested, SQL Server can impersonate the security context of the user running the application that requests the connection. If that user has sufficient privileges (or is part of a Windows NT/2000 domain group that does), the connection is established. Note that Windows Authentication is not available when SQL Server is running on Windows 98. When you connect to SQL Server running on Windows 98, you must specify a SQL Server logon ID and password.



Which Net-Library Is Fastest?

Strictly speaking, the TCP/IP Sockets Net-Library is the fastest Net-Library. In a pure network test that does nothing except throw packets back and forth between Net-Library pairs, the TCP/IP Sockets Net-Library is perhaps 30 percent faster than the slowest Net-Library. But in LAN environments and applications, the speed of the Net-Library probably makes little difference because the network interface is generally not a limiting factor in a well-designed application.

On a LAN, however, turning on encryption does cause a performance hit. But again, most applications probably wouldn't notice the difference. Your best bet is to choose the Net-Library that matches your network protocols and provides the services you need in terms of unified logon, encryption, and dynamic name resolution.

Open Data Services

Open Data Services (ODS) functions as the client manager for SQL Server; it's basically an interface between server Net-Libraries and server-based applications, including SQL Server.

ODS manages the network: it listens for new connections, cleans up failed connections, acknowledges "attentions" (cancellations of commands), coordinates threading services to SQL Server, and returns result sets, messages, and status values back to the client.

SQL Server clients and the server speak a private protocol known as *Tabular Data Stream (TDS)*. TDS is a self-describing data stream. In other words, it contains tokens that describe column names, datatypes, events (such as cancellations), and status values in the "conversation" between client and server. The server notifies the client that it is sending a result set, indicates the number of columns and datatypes of the result set, and so on—all encoded in TDS. Neither clients nor servers write directly to TDS. Instead, the open interfaces of OLE-DB, ODBC, and DB-Library at the client emit TDS using a client implementation of the Net-Library.

ODS accepts new connections, and if a client unexpectedly disconnects (for example, if a user reboots the client computer instead of cleanly terminating the application), ODS automatically frees resources such as locks held by that client.

You can use the ODS open interface to extend the functionality of SQL Server by writing extended stored procedures. The code for the ODS API itself was formerly stored outside of SQL Server as part of the DLL `opens60.dll`. In SQL Server 2000, to enhance the performance of SQL Server's own internal mechanisms that need to use presupplied extended procedures, the code that makes up ODS is internal to the SQL Server engine. Any extended stored procedure DLLs that you create must link against `opens60.dll`, which contains stubs to access the real ODS routines internal to SQL Server.

ODS input and output buffers

After SQL Server puts result sets into a network output buffer (write buffer) that's equal in size to the configured packet size, the Net-Library dispatches the buffer to the client. The first packet is sent as soon as the network output buffer is full or, if an entire result set fits in one packet, when the batch is completed. (A *batch* is one or more commands sent to SQL Server to be parsed and executed together. For example, if you're using `OSQL.EXE`, a batch is the collection of all the commands that appear before a specific `GO` command.) In some exceptional operations (such as one that provides progress information for database dumping or provides `DBCC` messages), the output buffer is flushed and sent even before it is full or before the batch completes.

SQL Server has two input buffers (read buffers) and one output buffer per client. Double-buffering is needed for the reads because while SQL Server reads a stream of data from the client connection, it must also look for a possible attention. (This allows that "Query That Ate Cleveland" to be canceled directly from the issuer. Although the ability to cancel a request is extremely important, it's relatively unusual among client/server products.) Attentions can be thought of as "out-of-band" data, although they can be sent with network protocols that do not explicitly have an out-of-band channel. The SQL Server development team experimented with double-buffering and asynchronous techniques for the output buffers, but these didn't improve performance substantially. The single network output buffer works nicely. Even though the writes are not posted asynchronously, SQL Server doesn't need to bypass the operating system caching for these as it does for writes to disk.

Because the operating system provides caching of network writes, write operations appear to complete immediately with no significant latency. If, however, several writes are issued to the same client and the client is not currently reading data from the network, the network cache eventually becomes full and the write blocks. This is essentially a throttle. As long as the client application is processing results, SQL Server has a few buffers queued up and ready for the client connection to process. But if the client's queue is already stacked up with results and is not processing them, SQL Server stalls sending them and the network write operation to that connection has to wait. Since the server has only one output buffer per client, data cannot be sent to that client connection until it reads information off the network to free up room for the write to complete. (Writes to other client connections are not held up, however; only those for the laggard client are affected.) Stalled network writes can also affect locks. For example, if `READ COMMITTED` isolation is in effect (the default), a share lock can normally be released after SQL

Server has completed its scan of that page of data. (Exclusive locks used for changing data must always be held until the end of the transaction to ensure that the changes can be rolled back.) However, if the scan finds more qualifying data and the output buffer is not free, the scan stalls. When the previous network write completes, the output buffer becomes available and the scan resumes. But as I stated earlier, that write won't complete until the client connection "drains" (reads) some data to free up some room in the pipe (the virtual circuit between SQL Server and client connection).

If a client connection delays processing results that are sent to it, concurrency issues can result because locks are held longer than they otherwise would be. A kind of chain reaction occurs: if the client connection has not read several outstanding network packets, further writing of the output buffer at the SQL Server side must wait because the pipe is full. Since the output buffer is not available, the scan for data might also be suspended because no space is available to add qualifying rows. Since the scan is held up, any lock on the data cannot be released. In short, if a client application does not process results in a timely manner, database concurrency can suffer. The size of the network output buffer can also affect the speed at which the client receives the first result set. As mentioned earlier, the output buffer is sent when the batch, not simply the command, is done, even if the buffer is not full. If two queries exist in the same batch and the first query has only a small amount of data, its results are not sent back to the client until the second query is done or has supplied enough data to fill the output buffer. If both queries are fast, waiting for both queries to finish is not a problem. But suppose the first query is fast and the second is slow. And suppose the first query returns 1000 bytes of data. If the network packet size is 4096 bytes, the first result set must wait in the output buffer for the second query to fill it. The obvious solution here is to either make the first command its own batch or make the network packet size smaller. The first solution is probably best in this case, since it is typically difficult to fine-tune your application to determine the best buffer size for each command. But this doesn't mean that each command should be its own batch. Quite the contrary. In fact, under normal circumstances, grouping multiple commands into a single batch is most efficient and is recommended because it reduces the amount of handshaking that must occur between client and server.

ODS default Net-Libraries

On the server side, ODS provides functionality that mirrors that of ODBC, OLE DB, or DB-Library at the client. Calls exist for an ODS server application to describe and send result sets, to convert values between datatypes, to assume the security context associated with the specific connection being managed, and to raise errors and messages to the client application.

ODS uses an event-driven programming model. Requests from servers and clients trigger events that your server application must respond to. Using the ODS API, you create a custom routine, called an *event handler*, for each possible type of event. Essentially, the ODS library drives server application by calling its custom event handlers in response to incoming requests. ODS server applications respond to the following events:

Connect events

When a connect event occurs, SQL Server initiates a security check to determine whether a connection is allowed. Other ODS applications, such as a gateway to DB/2, have their own logon handlers that determine whether connections are allowed. Events also exist that close a connection, allowing the proper connection cleanup to occur.

Language events

When a client sends a command string, such as an SQL statement, SQL Server passes this command along to the command parser. A different ODS application, such as a gateway, would install its own handler that accepts and is responsible for execution of the command.

Remote stored procedure events

These events occur each time a client or SQL Server directs a remote stored procedure request to ODS for processing.

ODS also generates events based on certain client activities and application activities. These events allow an ODS server application to respond to changes to the status of the client connection or of the ODS server application.

In addition to handling connections, ODS manages threads (and fibers) for SQL Server. It takes care of thread creation and termination and makes the threads available to the User Mode Scheduler (UMS). Since ODS is an open interface with a full programming API and toolkit, independent software vendors (ISVs) writing server applications with ODS get the same benefits that SQL Server derives from this component, including SMP-capable thread management and pooling, as well as network handling for multiple simultaneous networks. This multithreaded operation enables ODS server applications to maintain a high level of performance and availability and to transparently use multiple processors under Windows NT/2000 because the operating system can schedule any thread on any available processor.

The Relational Engine and the Storage Engine

The SQL Server database engine is made up of two main components, the relational engine and the storage engine. These two pieces are clearly separated, and their primary method of communication with each other is through OLE DB. The relational engine comprises all the components necessary to parse and optimize any query. It also manages the execution of queries as it requests data from the storage engine in terms of OLE DB row sets and then processes the row sets returned. (*Row set* is the OLE DB term for a result set.) The storage engine comprises the components needed to actually access and modify data on disk.

The command parser

The command parser handles language events raised by ODS. It checks for proper syntax and translates Transact-SQL commands into an internal format that can be operated on. This internal format is known as a *query tree*. If the parser doesn't recognize the syntax, a syntax error is immediately raised and identifies where the error occurred. However, non-syntax error messages cannot be explicit about the exact source line that caused the error. Because only the command parser can access the source of the statement, the statement is no longer available in source format when the command is actually executed.

The optimizer

The optimizer takes the query tree from the command parser and prepares it for execution. This module compiles an entire command batch, optimizes queries, and checks security. The query optimization and compilation result in an execution plan.

The first step in producing such a plan is to *normalize* each query, which potentially breaks down a single query into multiple, fine-grained queries. After the optimizer normalizes a query, it *optimizes* it, which means that the optimizer determines a plan for executing that query. Query optimization is cost-based; the optimizer chooses the plan that it determines would cost the least based on internal metrics that include estimated memory requirements, estimated CPU utilization, and the estimated number of required I/Os. The optimizer considers the type of statement requested, checks the amount of data in the various tables affected, looks at the indexes available for each table, and then looks at a sampling of the data values kept for each index or column referenced in the query. The sampling of the data values is called *distribution statistics*. Based on the available information, the optimizer considers the various access methods and join strategies it could use to resolve a query and chooses the most cost-effective plan. The optimizer also decides which indexes, if any, should be used for each table in the query, and, in the case of a multitable query, the order in which the tables should be accessed and the join strategy to use. The optimizer also uses pruning heuristics to ensure that more time isn't spent optimizing a query than it would take to simply choose a plan and execute it. The optimizer doesn't necessarily do exhaustive optimization. Some products consider every possible plan and then choose the most cost-effective one. The advantage of this exhaustive optimization is that the syntax chosen for a query would theoretically never cause a performance difference, no matter what syntax the user employed. But if you deal with an involved query, it could take much longer to estimate the cost of every conceivable plan than it would to accept a good plan, even if not the best one, and execute it.

After normalization and optimization are completed, the normalized tree produced by those processes is compiled into the execution plan, which is actually a data structure. Each command included in it specifies exactly which table will be affected, which indexes will be used (if any), which security checks must be made, and which criteria (such as equality to a specified value) must evaluate to TRUE for selection. This execution plan might be considerably more complex than is immediately apparent. In addition to the actual commands, the execution plan includes all the steps necessary to ensure that constraints are checked. Steps for calling a trigger are a bit different from those for verifying constraints. If a trigger is included for the action being taken, a call to the procedure that comprises the trigger is appended. If the trigger is an *instead-of* trigger, the call to the trigger's plan replaces the actual data modification command. For *after* triggers, the trigger's plan is branched to right after the plan for the modification statement that fired the trigger, before that modification is committed. The specific steps for the trigger are not compiled into the execution plan, unlike those for constraint verification.

A simple request to insert one row into a table with multiple constraints can result in an execution plan that requires many other tables to also be accessed or expressions to be evaluated. The existence of a trigger can also cause many additional steps to be executed. The step that carries out the actual INSERT statement might be just a small part of the total execution plan necessary to ensure that all actions and constraints associated with adding a row are carried out.

The SQL manager

The SQL manager is responsible for everything having to do with managing stored procedures and their plans. It determines when a stored procedure needs recompilation based on changes in the underlying objects' schemas, and it manages the caching of procedure plans so that other processes can reuse them.

The SQL manager also handles autoperparameterization of queries. In SQL Server 2000, certain kinds of ad hoc queries are treated as if they were parameterized stored procedures, and query plans are generated and saved for them. This can happen if a query uses a simple equality comparison against a constant, as in the following statement:

```
SELECT * FROM pubs.dbo.titles
WHERE type = 'business'
```

This query can be parameterized as if it were a stored procedure with a parameter for the value of *type*:

```
SELECT * FROM pubs.dbo.titles
WHERE type = @param
```

A subsequent query, differing only in the actual value used for the value of *type*, can use the same query plan that was generated for the original query.

The expression manager

The expression manager handles computation, comparison, and data movement. Suppose your query contains an expression like this one:

```
SELECT @myqty = qty * 10 FROM mytable
```

The expression manager copies the value of *qty* from the row set returned by the storage engine, multiplies it by 10, and stores the result in *@myqty*.

The query executor

The query executor runs the execution plan that the optimizer produced, acting as a dispatcher for all the commands in the execution plan. This module loops through each command step of the execution plan until the batch is complete. Most of the commands require interaction with the storage engine to modify or retrieve data and to manage transactions and locking.

Communication between the relational engine and the storage engine The relational engine uses OLE DB for most of its communication with the storage engine. The following description of that communication is adapted from the section titled "Database Engine Components" in *SQL Server Books Online*. It describes how a SELECT statement that processes data from local tables only is processed:

The relational engine compiles the SELECT statement into an optimized execution plan. The execution plan defines a series of operations against simple OLE DB row sets from the individual tables or indexes referenced in the SELECT statement. The row sets requested by the relational engine return the amount of data needed from a table or index to perform one of the operations used to build the SELECT result set. For example, this SELECT statement requires a table scan if it references a table with no indexes:

```
SELECT * FROM Northwind.dbo.ScanTable
```

The relational engine implements the table scan by requesting one row set containing all the rows from *ScanTable*. This next SELECT statement needs only information available in an index:

```
SELECT DISTINCT LastName
FROM Northwind.dbo.Employees
```

The relational engine implements the index scan by requesting one row set containing the leaf rows from the index that was built on the *LastName* column. The following SELECT statement needs information from two indexes:

```
SELECT CompanyName, OrderID, ShippedDate
FROM Northwind.dbo.Customers AS Cst
JOIN Northwind.dbo.Orders AS Ord
ON (Cst.CustomerID = Ord.CustomerID)
```

The relational engine requests two row sets: one for the nonclustered index on *Customers* and the other for one of the clustered indexes on *Orders*.

The relational engine uses the OLE DB API to request that the storage engine open the row sets. As the relational engine works through the steps of the execution plan and needs data, it uses OLE DB to fetch the individual rows from the row sets it asked the storage engine to open. The storage engine transfers the data from the data buffers to the relational engine.

The relational engine combines the data from the storage engine row sets into the final result set transmitted back to the user.

Not all communication between the relational engine and the storage engine uses OLE DB. Some commands cannot be expressed in terms of OLE DB row sets. The most obvious and common example is when the relational engine processes data definition language (DDL) requests to create a table or other SQL Server object.

The Access Methods Manager

When SQL Server needs to locate data, it calls the access methods manager. The access methods manager sets up and requests scans of data pages and index pages and prepares the OLE DB row sets to return to the relational engine. It contains services to open a table, retrieve qualified data, and update data. The access methods manager doesn't actually retrieve the pages; it makes the request to the buffer manager, which ultimately serves up the page already in its cache or reads it to cache from disk. When the scan is started, a look-ahead mechanism qualifies the rows or index entries on a page. The retrieving of rows that meet specified criteria is known as a *qualified retrieval*. The access methods manager is employed not only for queries (selects) but also for qualified updates and deletes

for example, UPDATE with a WHERE clause.

A session opens a table, requests and evaluates a range of rows against the conditions in the WHERE clause, and then closes the table. A session descriptor data structure (SDES) keeps track of the current row and the search conditions for the object being operated on (which is identified by the object descriptor data structure, or DES).

The Row Operations Manager and the Index Manager

You can consider the row operations manager and the index manager as components of the access methods manager because they carry out the actual method of access. Each is responsible for manipulating and maintaining its respective on-disk data structures, namely rows of data or B-tree indexes. They understand and manipulate information on data and index pages.

The row operations manager

The row operations manager retrieves, modifies, and performs operations on individual rows. It performs an operation within a row, such as "retrieve column 2" or "write this value to column 3." As a result of the work performed by the access methods manager, as well as by the lock manager and transaction manager, which will be discussed shortly, the row will have been found and will be appropriately locked and part of a transaction. After formatting or modifying a row in memory, the row operations manager inserts or deletes a row.

The row operations manager also handles updates. SQL Server 2000 offers three methods for handling updates. All three are direct, which means that there's no need for two passes through

the transaction log, as was the case with deferred updates in versions of SQL Server prior to SQL Server 7. SQL Server 2000 has no concept of a deferred data modification operation.

SQL Server 2000 has three update modes:

In-place mode

This mode is used to update a heap or clustered index when none of the clustering keys change. The update can be done in place, and the new data is written to the same slot on the data page.

Split mode

This mode is used to update nonunique indexes when the index keys change. The update is split into two operations—a delete followed by an insert—and these operations are performed independently of each other.

Split with collapse mode

This mode is used to update a unique index when the index keys change. After the update is rewritten as delete and insert operations, if the same index key is both deleted and then reinserted with a new value, the delete and insert are "collapsed" into a single update operation.

The index manager

The index manager maintains and supports searches on B-trees, which are used for SQL Server indexes. An index is structured as a tree, with a root page and intermediate-level and lower-level pages (or branches). A B-tree groups records that have similar index keys, thereby allowing fast access to data by searching on a key value. The B-tree's core feature is its ability to balance the index tree. (*B* stands for *balanced*.) Branches of the index tree are spliced together or split apart as necessary so that the search for any given record always traverses the same number of levels and thus requires the same number of page accesses.

The traversal begins at the root page, progresses to intermediate index levels, and finally moves to bottom-level pages called *leaf pages*. The index is used to find the correct leaf page. On a qualified retrieval or delete, the correct leaf page is the lowest page of the tree at which one or more rows with the specified key or keys reside. SQL Server supports both clustered and nonclustered indexes. In a nonclustered index, shown in Figure 3-3, the leaf level of the tree (the leaf pages of the index) contains every key value in the index along with a bookmark for each key value. The bookmark indicates where to find the referenced data and can have one of two forms, depending on whether the base table has a clustered index. If the base table has no clustered index, the table is referred to as a *heap*. The bookmarks in nonclustered index leaf pages for a heap are pointers to the actual records in which the data can be found, and these pointers consist of a row ID (RID), which is a file number, a page number, and a row number on the page. If the base table has a clustered index, the bookmark in any nonclustered index leaf page contains the clustered index key value for the row.

After reaching the leaf level in a nonclustered index, you can find the exact location of the data, although you still must separately retrieve the page on which that data resides. Because you can access the data directly, you don't need to scan all the data pages to find a qualifying row. Better yet, in a clustered index, shown in Figure 3-4, the leaf level actually contains the full data rows, not simply the index keys. A clustered index keeps the data in a table logically ordered around the key of the clustered index, and the leaf pages of a clustered index are in fact the data pages of the table. All the data pages of a table with a clustered index are linked together in a doubly linked list. Following the pages, and the rows on those pages, from the first page to the last page provides the logical order to the data.

Because data can be ordered in only one way, only one clustered index can exist per table. This makes the selection of the appropriate key value on which to cluster data an important performance consideration.

You can also use indexes to ensure the uniqueness of a particular key value. In fact, the PRIMARY KEY and UNIQUE constraints on a column work by creating a unique index on the column's values. The optimizer can use the knowledge that an index is unique in formulating an effective query plan. Internally, SQL Server always ensures that clustered indexes are unique by adding a 4-byte *uniqueifier* to clustered index key values that occur more than once. This uniqueifier becomes part of the key and is used in all levels of the clustered index and in references to the clustered index key through all nonclustered indexes.

Since SQL Server maintains ordering in index leaf levels, you do not need to unload and reload data to maintain clustering properties as data is added and moved. SQL Server always inserts rows into the correct page in clustered sequence. For a clustered index, the correct leaf page is the data page in which a row is inserted. For a nonclustered index, the correct leaf page is the one into which SQL Server inserts a row containing the key value (and bookmark) for the newly inserted row. If data is updated and the key values of an index change, or if the row is moved to a different page, SQL Server's transaction control ensures that all affected indexes are modified to reflect these changes. With transaction control, index operations are performed as atomic operations. The operations are logged and fully recovered in the event of a system failure.

Locking and Index Pages

As you'll see later, in the section on the lock manager, pages of an index use a slightly different locking mechanism than regular data pages. A lightweight lock called a *latch* is used to lock upper levels of indexes. Latches are not involved in deadlock detection because SQL Server 2000 uses "deadlock-proof" algorithms for index maintenance.

You can customize the locking strategy for indexes on a table basis or index basis. You can use the system stored procedure *sp_indexoption* to enable or disable page or row locks with any particular index or, by specifying a table name, for every index on that table. The settable options are *DisAllowPageLocks* and *DisAllowRowLocks*. If both of these options are set to TRUE for a particular index, only table level locks are applied.

The Page Manager and the Text Manager

The page manager and the text manager cooperate to manage a collection of pages as named databases. Each database is a collection of 8-KB disk pages, which are spread across one or more physical files.

SQL Server uses eight types of disk pages: data pages, text/image pages, index pages, Page Free Space (PFS) pages, Global Allocation Map (GAM and SGAM) pages, Index Allocation Map (IAM) pages, Bulk Changed Map pages, and Differential Changed Map pages. All user data, except for the *text*, *ntext*, and *image* datatypes, are stored on data pages. These three datatypes, which are used for storing large objects (up to 2 GB each of text or binary data), can use a separate collection of pages, so the data is not typically stored on regular data pages with the rest of the rows. Instead, a pointer on the regular data page identifies the starting page and offset of the text/image data. However, in SQL Server 2000, large object data that contains only a few bytes can optionally be stored in the data row itself. Index pages store the Btrees that allow fast access to data. PFS pages keep track of which pages in a database are available to hold new data. Allocation pages (GAMs, SGAMs, and IAMs) keep track of the other pages. They contain no database rows and are used only internally. Bulk Changed Map pages and Differential Changed Map pages are used to make backup and recovery more efficient.

The page manager allocates and deallocates all types of disk pages, organizing extents of eight pages each. An extent can be either a uniform extent, for which all eight pages are allocated to the same object (table or index), or a mixed extent, which can contain pages from multiple objects. If an object uses fewer than eight pages, the page manager allocates new pages for that object from mixed extents. When the size of the object exceeds eight pages, the page manager allocates new space for that object in units of entire uniform extents. This optimization prevents

the overhead of allocation from being incurred every time a new page is required for a large table; this overhead is incurred only every eighth time. Perhaps most important, this optimization forces data of the same table to be contiguous, for the most part. At the same time, the ability to use mixed extents keeps SQL Server from wasting too much space if a database contains many small tables.

To determine how contiguous a table's data is, you use the DBCC SHOWCONTIG command. A table with a lot of allocation and deallocation can get fairly fragmented, and rebuilding the clustered index (which also rebuilds the table) or running DBCC INDEXDEFRAG can improve performance, especially when a table is accessed frequently using ordered index scans.

The Transaction Manager

A core feature of SQL Server is its ability to ensure that transactions follow the ACID properties. Transactions must be *atomic*—that is, all or nothing. If a transaction has been committed, it must be recoverable by SQL Server no matter what—even if a total system failure occurs one millisecond after the commit was acknowledged. In SQL Server, if work was in progress and a system failure occurred before the transaction was committed, all the work is rolled back to the state that existed before the transaction began. Write-ahead logging makes it possible to always roll back work in progress or roll forward committed work that has not yet been applied to the data pages. Write-ahead logging ensures that a transaction's changes—the "before and after" images of data—are captured on disk in the transaction log before a transaction is acknowledged as committed. Writes to the transaction log are always synchronous—that is, SQL Server must wait for them to complete. Writes to the data pages can be asynchronous because all the effects can be reconstructed from the log if necessary. The transaction manager coordinates logging, recovery, and buffer management. These topics are discussed later in this chapter; at this point, we'll just look at transactions themselves.

The transaction manager delineates the boundaries of statements that must be grouped together to form an operation. It handles transactions that cross databases within the same SQL Server, and it allows nested transaction sequences. (However, nested transactions simply execute in the context of the first-level transaction; no special action occurs when they are committed. And a rollback specified in a lower level of a nested transaction undoes the entire transaction.) For a distributed transaction to another SQL Server (or to any other resource manager), the transaction manager coordinates with the Microsoft Distributed Transaction Coordinator (MS DTC) service using operating system remote procedure calls. The transaction manager marks *savepoints*, which let you designate points within a transaction at which work can be partially rolled back or undone.

The transaction manager also coordinates with the lock manager regarding when locks can be released, based on the isolation level in effect. The isolation level in which your transaction runs determines how sensitive your application is to changes made by others and consequently how long your transaction must hold locks to protect against those changes. Four isolation-level semantics are available in SQL Server 2000: Uncommitted Read (also called "dirty read"), Committed Read, Repeatable Read, and Serializable.

The behavior of your transactions depends on the isolation level. We'll look at these levels now, but a complete understanding of isolation levels also requires an understanding of locking because the topics are so closely related.

Uncommitted Read

Uncommitted Read, or dirty read (not to be confused with "dirty page," which I'll discuss later) lets your transaction read any data that is currently on a data page, whether or not that data has been committed. For example, another user might have a transaction in progress that has updated data, and even though it's holding exclusive locks on the data, your transaction can read it anyway. The other user might then decide to roll back his or her transaction, so logically those

changes were never made. If the system is a single-user system and everyone is queued up to access it, the changes will not have been visible to other users. In a multiuser system, however, you read the changes and take action based on them. Although this scenario isn't desirable, with Uncommitted Read you can't get stuck waiting for a lock, nor do your reads issue share locks (described in the next section) that might affect others.

When using Uncommitted Read, you give up the assurance of strongly consistent data in favor of high concurrency in the system without users locking each other out. So when should you choose Uncommitted Read? Clearly, you don't want to use it for financial transactions in which every number must balance. But it might be fine for certain decision-support analyses—for example, when you look at sales trends—for which complete precision isn't necessary and the tradeoff in higher concurrency makes it worthwhile.

Committed Read

Committed Read is SQL Server's default isolation level. It ensures that an operation never reads data that another application has changed but not yet committed. (That is, it never reads data that logically never existed.) With Committed Read, if a transaction is updating data and consequently has exclusive locks on data rows, your transaction must wait for those locks to be released before you can use that data (whether you're reading or modifying). Also, your transaction must put share locks (at a minimum) on the data that will be visited, which means that data might be unavailable to others to use. A share lock doesn't prevent others from reading the data, but it makes them wait to update the data. Share locks can be released after the data has been sent to the calling client—they don't have to be held for the duration of the transaction.

□ Although a transaction can never read uncommitted data when running with Committed Read isolation, if the transaction subsequently revisits the same data, that data might have changed or new rows might suddenly appear that meet the criteria of the original query. If data values have changed, we call that a *non-repeatable read*. New rows that appear are called *phantoms*.

Repeatable Read

The Repeatable Read isolation level adds to the properties of Committed Read by ensuring that if a transaction revisits data or if a query is reissued, the data will not have changed. In other words, issuing the same query twice within a transaction will not pick up any changes to data values made by another user's transaction. However, Repeatable Read isolation level does allow phantom rows to appear.

Preventing nonrepeatable reads from appearing is a desirable safeguard. But there's no free lunch. The cost of this extra safeguard is that all the shared locks in a transaction must be held until the completion (COMMIT or ROLLBACK) of the transaction. (Exclusive locks must always be held until the end of a transaction, no matter what the isolation level, so that a transaction can be rolled back if necessary. If the locks were released sooner, it might be impossible to undo the work.) No other user can modify the data visited by your transaction as long as your transaction is outstanding. Obviously, this can seriously reduce concurrency and degrade performance. If transactions are not kept short or if applications are not written to be aware of such potential lock contention issues, SQL Server can appear to "hang" when it's simply waiting for locks to be released.

You can control how long SQL Server waits for a lock to be released by using the session option LOCK_TIMEOUT.

Serializable

The Serializable isolation level adds to the properties of Repeatable Read by ensuring that if a query is reissued, rows will not have been added in the interim. In other words, phantoms will not appear if the same query is issued twice within a transaction. More precisely, Repeatable Read and Serializable affect sensitivity to another connection's changes, whether or not the user ID of the other connection is the same. Every connection within SQL Server has its own transaction and lock space. I use the term "user" loosely so as not to obscure the central concept.

Preventing phantoms from appearing is another desirable safeguard. But once again, there's no free lunch. The cost of this extra safeguard is similar to that of Repeatable Read—all the shared locks in a transaction must be held until completion of the transaction. In addition, enforcing the Serializable isolation level requires that you not only lock data that has been read, but also lock data *that does not exist!* For example, suppose that within a transaction we issue a SELECT statement to read all the customers whose zip code is between 98000 and 98100, and on first execution no rows satisfy that condition. To enforce the Serializable isolation level, we must lock that "range" of *potential* rows with zip codes between 98000 and 98100 so that if the same query is reissued, there will still be no rows that satisfy the condition. SQL Server handles this by using a special kind of lock called a *key-range lock*. The Serializable level gets its name from the fact that running multiple serializable transactions at the same time is the equivalent of running them one at a time—that is, serially. For example, suppose transactions A, B, and C run simultaneously at the Serializable level and each tries to update the same range of data. If the order in which the transactions acquire locks on the range of data is B, C, and A, the result obtained by running all three simultaneously is the same as if they were run one at a time in the order B, C, and A. Serializable does not imply that the order is known in advance. The order is considered a chance event. Even on a single-user system, the order of transactions hitting the queue would be essentially random. If the batch order is important to your application, you should implement it as a pure batch system.

The Lock Manager

Locking is a crucial function of a multiuser database system such as SQL Server. SQL Server lets you manage multiple users simultaneously and ensures that the transactions observe the properties of the chosen isolation level. At the highest level, Serializable, SQL Server must make the multiuser system perform like a single-user system—as though every user is queued up to use the system alone with no other user activity. Locking guards data and the internal resources that make it possible for many users to simultaneously access the database and not be severely affected by others' use.

The lock manager acquires and releases various types of locks, such as share locks for reading, exclusive locks for writing, intent locks to signal a potential "plan" to perform some operation, extent locks for space allocation, and so on. It manages compatibility between the lock types, resolves deadlocks, and escalates locks if needed. The lock manager controls table, page, and row locks as well as system data locks. (System data, such as page headers and indexes, are private to the database system.)

The lock manager provides two separate locking systems. The first enables row locks, page locks, and table locks for all fully shared data tables, data pages and rows, text pages, and leaf-level index pages and index rows. The second locking system is used internally only for restricted system data; it protects root and intermediate index pages while indexes are being traversed.

This internal mechanism uses *latches*, a lightweight, short-term variation of a lock for protecting data that does not need to be locked for the duration of a transaction. Full-blown locks would slow the system down. In addition to protecting upper levels of indexes, latches protect rows while they are being transferred from the storage engine to the relational engine. If you examine locks by using the *sp_lock* stored procedure or a similar mechanism that gets its information from the *syslockinfo* system table, you won't see or be aware of latches; you'll see only the locks for fully

shared data. However, counters are available in the System Monitor to monitor latch requests, acquisitions, and releases.

Other Managers

Also included in the storage engine are managers for controlling utilities such as bulk load, DBCC commands, backup and restore operations, and the Virtual Device Interface (VDI). VDI allows ISVs to write their own backup and restore utilities and to access the SQL Server data structures directly, without going through the relational engine. There is a manager to control sorting operations and one to physically manage the files and backup devices on disk.

Data Integrity

Creating a model of the entities in the problem space and the relationships between them is only part of the data modeling process. You must also capture the rules that the database system will use to ensure that the actual physical data stored in it is, if not correct, at least plausible. In other words, you must model the data integrity.

It's important to understand that the chances of being able to guarantee the literal correctness of the data are diminishingly small. Take, for example, an order record showing that Mary Smith purchased 17 hacksaws on July 15, 1999. The database system can ensure that Mary Smith is a customer known to the system, that the company does indeed sell hacksaws, and that it was taking orders on July 15, 1999. It can even check that Mary Smith has sufficient credit to pay for the 17 hacksaws. What it can't do is verify that Ms. Smith actually ordered 17 hacksaws and not 7 or 1, or 17 screwdrivers instead. The best the system might do is notice that 17 is rather a lot of hacksaws for an individual to purchase and notify the person entering the order to that effect. Even having the system do this much is likely to be expensive to implement, probably more expensive than its value warrants.

My point is that the system can never verify that Mary Smith did place the order as it's recorded; it can verify only that she could have done so. Of course, that's all any record-keeping system can do, and a well-designed database system can certainly do a better job than the average manual system, if for no other reason than its consistency in applying the rules. But no database system, and no database system designer, can guarantee that the data in the database is true, only that it could be true. It does this by ensuring that the data complies with the integrity constraints that have been defined for it.

Integrity Constraints

Some people refer to integrity constraints as business rules. However, the concept of business rules is much broader; it includes all of the constraints on the system rather than just the constraints concerning the integrity of the data. In particular, system security—the definition of which users can do what and under what circumstances they can do it—is part of system administration, not data integrity. But certainly security is a business requirement and will constitute one or more business rules.

Data integrity is implemented at several levels of granularity. Domain, transition, and entity constraints define the rules for maintaining the integrity of the individual relations. Referential integrity constraints ensure that necessary relationships between relations are maintained. Database integrity constraints govern the database as a whole, and transaction integrity

constraints control the way data is manipulated either within a single database or between multiple databases.

Domain Integrity

A domain is the set of all possible values for a given attribute. A domain integrity constraint usually just called a domain constraint is a rule that defines these legal values. It might be necessary to define more than one domain constraint to describe a domain completely.

A domain isn't the same thing as a data type, and defining domains in terms of physical data types is a tempting (and common) practice, but it can backfire. The danger is that you will unnecessarily constrain the values for example, by choosing an Integer data type because you think it will be big enough in practice, rather than because 255 is the largest permitted value for the domain.

That being said, however, data type can be a convenient shorthand in the data model, and for this reason choosing a logical data type is often the first step in determining the domain constraints in a system. Dates are probably the best example of the benefits of this approach. I'd recommend against defining the domain TransactionDate as "DateTime", which is a physical representation. But defining it as "a date" allows you to concentrate on it being "between the commencement of business and the present date, inclusive" and ignore all those rather tedious rules about leap years.

Having chosen a logical data type, it might be appropriate to narrow the definition by, for example, indicating the scale and precision of a numeric type, or the maximum length of string values. This is very close to specifying a physical data type, but you should still be working at the logical level.

The next aspect of domain integrity to consider is whether a domain is permitted to contain unknown or nonexistent values. The handling of these values is contentious, and we'll be discussing them repeatedly as we examine various aspects of database system design. For now, it's necessary to understand only that there is a difference between an unknown value and a nonexistent value, and that it is often (although not always) possible to specify whether either or both of these is permitted for the domain.

The first point here, that "unknown" and "nonexistent" are different, doesn't present too many problems at the logical level. My father does not have a middle name; I do not know my neighbor's. These are quite different issues. There are implementation issues that need not yet concern us, but the logical distinction is quite straightforward.

The second point is that, having determined whether a domain is allowed to include unknown or nonexistent values, you'll need to decide whether either of these can be accepted by the system. To return to our TransactionDate example, it's certainly possible for the date of a transaction to be unknown, but if it occurred at all it occurred at some fixed point in time and therefore cannot be nonexistent. In other words, there must be a transaction date; we just might not know it.

Now, obviously, we can be ignorant of anything, so any value can be unknown. That's not a useful distinction. What we're actually defining here is not so much whether a value can be unknown as whether an entity with an unknown value in this attribute should be stored. It might be that it's not worth storing data unless the value is known, or it might be that we can't identify an entity without knowing the value. In either case, you would prevent a record containing an unknown value in the specified field from being added to the database.

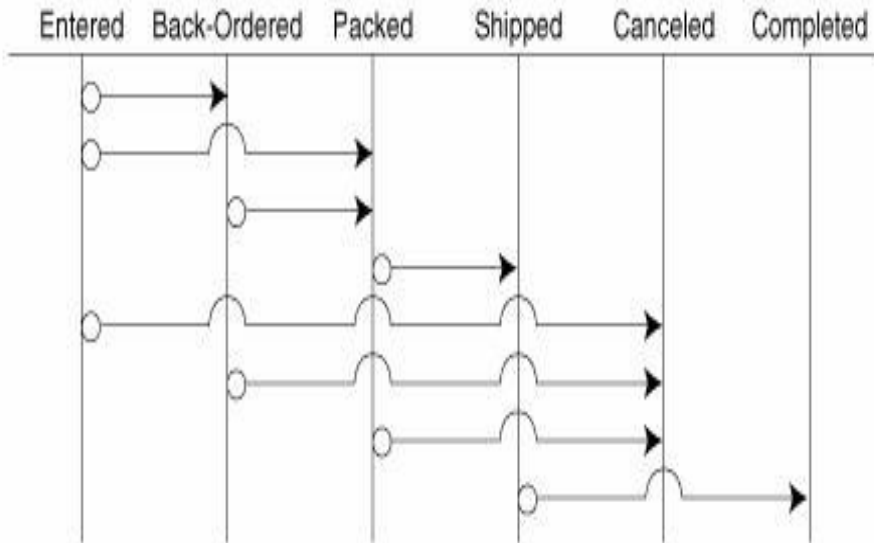
This decision can't always be made at the domain level, but it's always worth considering since doing so can make the job a little easier down the line. To some extent, your decision depends on how generic your domains are. As an example, say that you have defined a Name domain and declared the attributes GivenName, MiddleName, Surname, and CompanyName against it. You might just as well have defined these attributes as separate domains, but there are some advantages to using the more general domain definition because doing so allows you to capture the overlapping rules (and in this case, there are probably a lot of them) in a single place. However, in this case you won't be able to determine whether empty or unknown values are acceptable at the domain level; you will have to define these properties at the entity level. The final aspect of domain integrity is that you'll want to define the set of values represented by a domain as specifically as possible. Our TransactionDate domain, for example, isn't just the set of all dates; it's the set of dates from the day the company began trading until the current date. It might be further restricted to eliminate Sundays, public holidays, and any other days on which the company does not trade, although you need to be careful here. It's not unknown for employees to be in the office even though business is officially "closed," and absolutely no reason to restrict them from answering the phone to take an order when they are.

Sometimes the easiest way to describe a domain is to simply list the domain values. The domain of Weekends is completely described by the set {"Saturday", "Sunday"}. Sometimes it will be easier to list one or more rules for determining membership, as we did for TransactionDate. Both techniques are perfectly acceptable, although a specific design methodology might dictate a particular method of documenting constraints. The important thing is that the constraints be captured as carefully and completely as possible.

Transition Integrity

Transition integrity constraints define the states through which a tuple can validly pass. The State-Transition diagram, for example, shows the states through which an order can pass.

State Diagram for order processing



You would use transitional integrity constraints, for instance, to ensure that the status of a given order never changed from "Entered" to "Completed" without passing through the interim states, or to prevent a canceled order from changing status at all.

The status of an entity is usually controlled by a single attribute. In this case, transition integrity can be considered a special type of domain integrity. Sometimes, however, the valid transitions are controlled by multiple attributes or even multiple relations. Because transition constraints can exist at any level of granularity, it's useful to consider them a separate type of constraint during preparation of the data model.

The status of a customer might only be permitted to change from "Normal" to "Preferred" if the customer's credit limit is above a specified value and he or she has been doing business with the company for at least a year. The credit limit requirement would most likely be controlled by an attribute of the Customers relation, but the length of time the customer has been doing business with the company might not be explicitly stored anywhere. It might be necessary to calculate the value based on the oldest record for the customer in the Orders relation.

Entity Integrity

Entity constraints ensure the integrity of the entities being modeled by the system. At the simplest level, the existence of a primary key is an entity constraint that enforces the rule "every entity must be uniquely identifiable."

In a sense, this is the entity integrity constraint; all others are technically entity-level integrity constraints. The constraints defined at the entity level can govern a single attribute, multiple attributes, or the relation as a whole.

The integrity of an individual attribute is modeled first and foremost by defining the attribute against a specific domain. An attribute within a relation inherits the integrity constraints defined for its domain. At the entity level, these inherited constraints can properly be made more rigorous, but they cannot be relaxed. Another way of thinking about this is that the entity constraint can specify a subset of the domain constraints but not a superset. For example, an OrderDate attribute defined against the TransactionDate domain might specify that the date must be in the current year, whereas the TransactionDate domain allows any date between the date business commenced and the current date. An entity constraint cannot, however, allow OrderDate to contain dates in the future, since the attribute's domain prohibits these.

Similarly, a CompanyName attribute defined against the Name domain might prohibit empty values, even though the Name domain permits them. Again, this is a narrower, more rigorous definition of permissible values than that specified in the domain.

In addition to narrowing the range of values for a single attribute, entity constraints can also affect multiple attributes. A requirement that Shipping-Date be on or after OrderDate is an example of such a constraint. Entity constraints can't reference other relations, however. It wouldn't be appropriate, for example, to define an entity constraint that sets a customer DiscountRate (an attribute of the Customer relation) based on the customer's TotalSales (which is based on multiple records in the OrderItems relation). Constraints that depend on multiple relations are database-level constraints; we'll discuss them later in this chapter.

Be careful of multiple-attribute constraints; they often indicate that your data model isn't fully normalized. If you are restricting or calculating the value of one attribute based on another, you're probably OK. An entity constraint that says "Status is not allowed to be 'Preferred' unless the Customer record is at least one year old" would be fine. But if the value of one attribute determines the value of another for example, "If the customer record is older than one year, then Status = 'Preferred'" then you have a functional dependency and you're in violation of third normal form.

Referential Integrity

We already looked at the decomposition of relations to minimize redundancy and at foreign keys to implement the links between relations. If these links are ever broken, the system will be unreliable at best and unusable at worst. Referential integrity constraints maintain and protect these links.

There is really only one referential integrity constraint: Foreign keys cannot become orphans. In other words, no record in the foreign table can contain a foreign key that doesn't match a record in the primary table. Tuples that contain foreign keys that don't have a corresponding candidate key in the primary relation are called orphan entities.

There are four ways orphan entities can be created:

- A tuple is added to the foreign table with a key that does not match a candidate key in the primary table.
- The foreign key is changed to a value not in the primary table.
- The candidate key in the primary table is changed.
- The referenced record in the primary table is deleted.

All four of these cases must be handled if the integrity of a relationship is to be maintained. The first case, the addition of an unmatched foreign key, is usually simply prohibited. But note that unknown and nonexistent values don't count that's a separate rule. If the relationship is declared as optional, any number of unknown and nonexistent values can be entered without compromising referential integrity.

The second and third causes of orphaned entities changing the candidate key value in the referenced table shouldn't occur very often. (This would be an entity constraint, by the way: "Candidate keys are not allowed to change.")

If your model does allow candidate keys to be changed, you must ensure that these changes are made in the foreign keys as well. This is known as a cascading update. Both Microsoft Jet and Microsoft SQL Server provide mechanisms for easily implementing cascading updates.

If your model allows foreign keys to change, in effect allowing the entity to be re-assigned, you must ensure that the new value is valid. Again, Microsoft Jet and Microsoft SQL Server both allow this constraint to be easily implemented, even though it doesn't have a specific name.

The final cause of orphan foreign keys is the deletion of the record containing the primary entity. If one deletes a Customer record, for example, what becomes of that customer's orders? As with candidate key changes, you can simply prohibit the deletion of tuples in the primary relation if they are referenced in a foreign relation. This is certainly the cleanest solution if it is a reasonable restriction for your system. When it's not, both the Jet database engine and SQL Server provide a simple means of cascading the operation, known as a cascading delete.

But in this case, you also have a third option that's a little harder to implement because it can't be implemented automatically. You might want to re-assign the dependent records. This isn't often appropriate, but it is sometimes necessary. Say, for example, that CustomerA purchases CustomerB. It might make sense to delete CustomerB and reassign all of CustomerB's orders to CustomerA.

A special kind of referential integrity constraint is the maximum cardinality issue. In the data model, rules such as "Managers are allowed to have a maximum of five individuals reporting to them" are defined as referential constraints

Database Integrity

The most general form of integrity constraint is the database constraint. Database constraints reference more than one relation: "All Products sold to Customers with the status of 'Government' must have more than one Supplier." The majority of database constraints take this form, and, like this one, may require calculations against multiple relations.

You should be careful not to confuse a database constraint with the specification of a work process. A work process is something that is done with the database, such as adding an order, whereas an integrity constraint is a rule about the contents of the database. The rules that define the tasks that are performed using the database are part of the work process, not database constraints. Work processes, can have a major impact on the data model, but they shouldn't be made a part of it.

It isn't always clear whether a given business rule is an integrity constraint or a work process (or something else entirely). The difference might not be desperately important. All else being equal, implement the rule where it's most convenient to do so. If it's a straightforward process to express a rule as a database constraint, do so. If that gets tricky (as it often can, even when the rule is clearly an integrity constraint), move it to the middle tier or the front end, where it can be implemented procedurally.

Transaction Integrity

The final form of database integrity is transaction integrity. Transaction integrity constraints govern the ways in which the database can be manipulated. Unlike other constraints, transaction constraints are procedural and thus are not part of the data model per se.

Transactions are closely related to work processes. The concepts are, in fact, orthogonal, inasmuch as a given work process might consist of one or more transactions and vice versa. It isn't quite correct, but it's useful to think of a work process as an abstract construct ("add an order") and a transaction as a physical one ("update the OrderDetail table").

A transaction is usually defined as a "logical unit of work," which I've always found to be a particularly unhelpful bit of rhetoric. Essentially, a transaction is a group of actions, all of which (or none of which) must be completed. The database must comply with all of the defined integrity constraints before the transaction commences and after it's completed, but might be temporarily in violation of one or more constraints during the transaction.

The classic example of a transaction is the transfer of money from one bank account to another. If funds are debited from Account A but the system fails to credit them to Account B, money has been lost. Clearly, if the second command fails, the first must be undone. In database parlance, it must be "rolled back." Transactions can involve multiple records, multiple relations, and even multiple databases.

To be precise, all operations against a database are transactions. Even updating a single existing record is a transaction. Fortunately, these low-level transactions are implemented transparently by the database engine, and you can generally ignore this level of detail.

Both the Jet database engine and SQL Server provide a means of maintaining transactional integrity by way of the BEGIN TRANSACTION, COMMIT TRANSACTION, and ROLLBACK TRANSACTION statements. As might be expected, SQL Server's implementation is more robust and better able to recover from hardware failure as well as certain kinds of software failure. However, these are implementation issues and are outside the scope of this book. What is

important from a design point of view is to capture and specify transaction dependencies, those infamous "logical units of work."

Declarative and Procedural Integrity

Relational database engines provide integrity support in two ways: declarative and procedural. Declarative integrity support is explicitly defined ("declared") as part of the database schema. Both the Jet database engine and SQL Server provide some declarative integrity support. Declarative integrity is the preferred method for implementing data integrity. You should use it wherever possible. SQL Server implements procedural integrity support by way of trigger procedures that are executed ("triggered") when a record is inserted, updated, or deleted. The Jet database engine does not support procedural integrity. When an integrity constraint cannot be implemented using declarative integrity, and the system is using the Jet engine, the constraint must be implemented in the front end.

Domain Integrity

SQL Server provides a limited kind of support for domains in the form of user-defined data types (UDDTs). Fields defined against a UDDT will inherit the data type declaration as well as domain constraints defined for the UDDT.

Equally importantly, SQL Server will prohibit comparison between fields declared against different UDDTs, even when the UDDTs in question are based on the same system data type. For example, even though the CityName domain and the CompanyName domain are both defined as being char(30), SQL Server would reject the expression `CityName = CompanyName`. This can be explicitly overridden by using the convert function `CityName = CONVERT(char(30), CompanyName)`, but the restriction forces you to think about it before comparing fields declared against different domains. This is a good thing, since these comparisons don't often make sense. UDDTs can be created either through the SQL Server Enterprise Manager or through the system stored procedure `sp_addtype`. Either way, UDDTs are initially declared with a name or a data type and by whether they are allowed to accept Nulls. Once a UDDT has been created, default values and validation rules can be defined for it. A SQL Server rule is a logical expression that defines the acceptable values for the UDDT (or for a field, if it is bound to a field rather than a UDDT). A default is simply that a default value to be inserted by the system into a field that would otherwise be Null because the user did not provide a value.

Binding a rule or default to a UDDT is a two-step procedure. First you must create the rule or default, and then bind it to the UDDT (or field). The "it's not a bug, it's a feature" justification for this two-step procedure is that, once defined, the rule or default can be reused elsewhere. I find this tedious since in my experience these objects are reused only rarely. When defining a table, SQL Server provides the ability to declare defaults and CHECK constraints directly, as part of the table definition. (CHECK constraints are similar to rules, but more powerful.) Unfortunately this one-step declaration is not available when declaring UDDTs, which must use the older "create-and-then-bind" methodology. It is heartily to be wished that Microsoft add support for default and CHECK constraint declarations to UDDTs in a future release of SQL Server.

A second way of implementing a kind of deferred domain integrity is to use lookup tables. This technique can be used in both Microsoft Jet and SQL Server. As an example, take the domain of USStates. Now, theoretically, you can create a rule listing all 50 states. In reality, this would be a painful process, particularly with the Jet database engine, where the rule would have to be retyped for every field declared against the domain.

Entity Integrity

In the database schema, entity constraints can govern individual fields, multiple fields, or the table as a whole. Both the Jet database engine and SQL Server provide mechanisms for ensuring integrity at the entity level. Not surprisingly, SQL Server provides a richer set of capabilities, but the gap is not as great as one might expect.

At the level of individual fields, the most fundamental integrity constraint is of course the data type. Both the Jet database engine and SQL Server provide a rich set of data types, as shown in

Table 1 The physical Data Types Suported by Microsoft Jet and SQL Server

Logical Data Type	SQL Server Data Type	Microsoft Jet Data Type	Value Range	Storage Size
Integer	Int	Long 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)	Decimal	Number (type varies)	Whole or fractional numbers from 1038^1 to 1038^1	217 bytes
Floating point (approx. numeric)	Float (15-digit precision)	Double	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	Single	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

SQL Server Data Type	SQL Server Data Type	SQL Server Data Type	SQL Server Data Type	SQL Server Data Type
Character (fixed length)	Char	N/A	Maximum of 255 characters in Jet; 8000 characters in SQL Server 7.0 (255 in previous versions)	1 byte per character declared
Character (variable length)	Varchar	Text	Maximum of 255 characters in Jet; 8000 characters in SQL Server 7.0 (255 in previous versions)	1 byte per character stored
Monetary	Money	Currency	Numbers accurate to four decimal places, from 922,337,208,685,477.5808 to 922,337,208,685,477.5807	8 bytes
	Smallmoney	N/A	Numbers accurate to four decimal places, from 214,748.3648 to 214,748.3647	4 bytes
Date and Time	Datetime	Date/Time	1 January 1753 to 31 December 9999 in SQL Server; 1 January 100 to 31 December 9999 in Jet	8 bytes
Binary (fixed length)	Smalldatetime	N/A	1 January 1900 to 6 June 2079	4 bytes
Binary (variable length)	Binary	N/A	Maximum of 8000 bytes	Number of bytes declared plus 4 bytes
Long Text/Binary	Varbinary	(Supported only for linked tables)	Maximum of 8000 bytes	Number of bytes actually stored plus 4 bytes
Long Object	Text	Memo	Character data up to 2 GB in SQL Server, or 1 GB in a Microsoft Jet database	Amount of data stored plus 16 bytes
(BLOB)	Image	OLE Object	Binary data up to 2 GB in SQL Server, or 1 GB in a Microsoft Jet database	Amount of data stored plus 16 bytes
Boolean	Bit	Yes/No	0 or 1	1 byte, but bit columns in a table are combined in SQL Server; thus 8 or fewer columns will take 1 byte in total

SQL Data Types

Data stored in a relational database can be stored using a variety of data types. The primary ORACLE data types are NUMBER, VARCHAR, and CHAR for storing numbers and a text data; however, there are additional data types that are supported to support backward compatibility with products given below:

Key Data Type

Key Data Types	
<ul style="list-style-type: none"> • CHAR(<i>size</i>) 	<ul style="list-style-type: none"> • Fixed-length character data, <i>size</i> characters long. Maximum size=255; default=1 byte. Padded on right with blanks to full length of <i>size</i>.
<ul style="list-style-type: none"> • DATE 	<ul style="list-style-type: none"> • Valid dates range from Jan 1, 4712 B.C. to Dec 31, 4712 A.D.
<ul style="list-style-type: none"> • NUMBER 	<ul style="list-style-type: none"> • For NUMBER column with space for 40 digits, plus space for a decimal point and sign. Numbers may be expressed in two ways: first, with numbers 0 to 9, the signs + and -, and a decimal point(.); second, in scientific notation, e.g. 1.85E3 for 1850. Valid values are 0 and positive and negative numbers from 1.0E-130 to 9.99...E125.
<ul style="list-style-type: none"> • VARCHAR2(<i>size</i>) 	<ul style="list-style-type: none"> • Variable length character string, maximum <i>size</i> up to 2000 bytes.
<p>MISCELLANEOUS DATA TYPES AND VARIATIONS</p>	
<ul style="list-style-type: none"> • DECIMAL 	<ul style="list-style-type: none"> • Same as NUMBER.
<ul style="list-style-type: none"> • FLOAT 	<ul style="list-style-type: none"> • Same as NUMBER.
<ul style="list-style-type: none"> • INTEGER 	<ul style="list-style-type: none"> • Same as NUMBER.
<ul style="list-style-type: none"> • INTEGER(<i>size</i>) 	<ul style="list-style-type: none"> • Integer of specified <i>size</i> digits wide; same as NUMBER(<i>size</i>) of specific <i>size</i> digits wide.
<ul style="list-style-type: none"> • LONG 	<ul style="list-style-type: none"> • Character data of variable <i>size</i> up to 2Gb in length. Only one LONG column may be defined per table. LONG columns may not be used in subqueries, functions, expressions, where clauses, or indexes. A table containing LONG data may not be clustered.
<ul style="list-style-type: none"> • LONG RAW 	<ul style="list-style-type: none"> • Raw binary data; otherwise the same as LONG (used for images).

• LONG VARCHAR	• Same as LONG
• NUMBER(size)	• For NUMBER column of specified size in digits.
• NUMBER(size,d)	• For NUMBER column of specified size with <i>d</i> digits after the decimal point, e.g. NUMBER(5,2) could contain nothing larger than 999.99 without an error being generated.
• NUMBER(*)	• Same as NUMBER.
• SMALLINT	• Same as NUMBER.
• RAW(size)	• Raw binary data, size bytes long, maximum size=255 bytes.
• ROWID	• A value that uniquely identifies a row in an Oracle database - it is returned by the pseudo-column ROWID. Table columns may not be assigned this type.
• VARCHAR(size)	• Same as VARCHAR2. Always use VARCHAR2.

User-Defined Datatypes :

- Raw binary data, size bytes long, maximum size=255 bytes.
- A value that uniquely identifies a row in an Oracle database -it is returned by the pseudo-column ROWID. Table columns may not be assigned this type.
- Same as VARCHAR2. Always use VARCHAR2.

A user-defined datatype (UDDT) provides a convenient way for you to guarantee consistent use of underlying native datatypes for columns known to have the same domain of possible values. For example, perhaps your database will store various phone numbers in many tables. Although no single, definitive way exists to store phone numbers, in this database consistency is important. You can create a phone_number UDDT and use it consistently for any column in any table that keeps track of phone numbers to ensure that they all use the same datatype.

Here's how to create this UDDT:

```
EXEC sp_addtype phone_number, 'varchar(20)', 'not null'
```

And here's how to use the new UDDT when you create a table:

```
CREATE TABLE customer
(
cust_id    smallint    NOT NULL,
cust_name  varchar(50) NOT NULL,
cust_addr1 varchar(50) NOT NULL,
cust_addr2 varchar(50) NOT NULL,
cust_city  varchar(50) NOT NULL,
cust_state char(2)     NOT NULL,
cust_zip   varchar(10) NOT NULL,
cust_phone phone_number,
cust_fax   varchar(20) NOT NULL,
cust_email varchar(30) NOT NULL,
cust_web_url varchar(20) NOT NULL
)
```

When the table is created, internally the *cust_phone* datatype is known to be *varchar(20)*. Notice that both *cust_phone* and *cust_fax* are *varchar(20)*, although *cust_phone* has that declaration through its definition as a UDDT.

Here's how the customer table appears in the entries in the *syscolumns* table for this table:

```
SELECT colid, name, xtype, length, xusertype, offset
FROM syscolumns WHERE id=object_id('customer')
```

colid	name	xtype	length	xusertype	offset
1	cust_id	52	2	52	2
2	cust_name	167	50	167	-1
3	cust_addr1	167	50	167	-2
4	cust_addr2	167	50	167	-3
5	cust_city	167	50	167	-4
6	cust_state	175	2	175	4
7	cust_zip	167	10	167	-5
8	cust_phone	167	20	261	-6
9	cust_fax	167	20	167	-7
10	cust_email	167	30	167	-8
11	cust_web_url	167	20	167	-9

You can see that both the *cust_phone* and *cust_fax* columns have the same *xtype* (datatype), although the *cust_phone* column shows that the datatype is a UDDT (*xusertype* = 261). The type is resolved when the table is created, and the UDDT can't be dropped or changed as long as one or more tables are currently using it. Once declared, a UDDT is static and immutable, so no inherent performance penalty occurs in using a UDDT instead of the native datatype.

The use of UDDTs can make your database more consistent and clear. SQL Server implicitly converts between compatible columns of different types (either native types or UDDTs of different types).

Currently, UDDTs don't support the notion of subtyping or inheritance, nor do they allow a *DEFAULT* value or *CHECK* constraint to be declared as part of the UDDT itself.

Changing a Datatype

By using the ALTER COLUMN clause of ALTER TABLE, you can modify the datatype or NULL property of an existing column. But be aware of the following restrictions:

- The modified column can't be a *text*, *image*, *ntext*, or *rowversion (timestamp)* column.
- If the modified column is the ROWGUIDCOL for the table, only DROP ROWGUIDCOL is allowed; no datatype changes are allowed.
- The modified column can't be a computed or replicated column.
- The modified column can't have a PRIMARY KEY or FOREIGN KEY constraint defined on it.
- The modified column can't be referenced in a computed column.
- The modified column can't have the type changed to *timestamp*.
- If the modified column participates in an index, the only type changes that are allowed are increasing the length of a variable-length type (for example, VARCHAR(10) to VARCHAR(20)), changing nullability of the column, or both. If the modified column has a UNIQUE OR CHECK constraint defined on it, the only
 - change allowed is altering the length of a variable-length column. For a UNIQUE
 - constraint, the new length must be greater than the old length.
- If the modified column has a default defined on it, the only changes that are allowed are increasing or decreasing the length of a variable-length type, changing nullability, or changing the precision or scale.
- The old type of the column should have an allowed implicit conversion to the new type.
- The new type always has ANSI_PADDING semantics if applicable, regardless of the current setting.
- If conversion of an old type to a new type causes an overflow (arithmetic or size), the ALTER TABLE statement is aborted.

using the ALTER COLUMN clause of the

ALTER TABLE statement

SYNTAX

```
ALTER TABLE table-name ALTER COLUMN column-name
    { type_name [ ( prec [ , scale ] ) ] [COLLATE <collation name> ]
    [ NULL | NOT NULL ]
    | {ADD | DROP} ROWGUIDCOL }
```

```
/* Change the length of the emp_name column in the employee
   table from varchar(30) to varchar(50) */
ALTER TABLE employee
ALTER COLUMN emp_name varchar(50)
```

Identity Property

It is common to provide simple counter-type values for tables that don't have a natural or efficient primary key. Columns such as *cust_id* are usually simple counter fields. The IDENTITY property makes generating unique numeric values easy. IDENTITY isn't a datatype; it's a *column property* that you can declare on a whole-number datatype such as *tinyint*, *smallint*, *int*, or *numeric/decimal* (having a scale of zero). Each table can have only one column with the IDENTITY property. The table's creator can specify the starting number (seed) and the amount that this value increments or decrements. If not otherwise specified, the seed value starts at 1 and increments by 1.

```
CREATE TABLE customer
(
cust_id  smallint  IDENTITY NOT NULL,
cust_name varchar(50)  NOT NULL
)
```

To find out which seed and increment values were defined for a table, you can use the `IDENT_SEED(tablename)` and `IDENT_INCR(tablename)` functions.

Statement:

```
SELECT IDENT_SEED('customer'), IDENT_INCR('customer')
```

Output:

```
1      1
```

for the *customer* table because values weren't explicitly declared and the default values were used.

explicitly starts the numbering at 100 (seed) and increments the value by 20:

```
CREATE TABLE customer
(
cust_id  smallint  IDENTITY(100, 20) NOT NULL,
cust_name varchar(50)  NOT NULL
)
```

The value automatically produced with the `IDENTITY` property is normally unique, but that isn't guaranteed by the `IDENTITY` property itself. Nor is it guaranteed to be consecutive. For efficiency, a value is considered used as soon as it is presented to a client doing an `INSERT` operation. If that client doesn't ultimately commit the `INSERT`, the value never appears, so a break occurs in the consecutive numbers. An unacceptable level of serialization would exist if the next number couldn't be parceled out until the previous one was actually committed or rolled back.

If you need exact sequential values without gaps, `IDENTITY` isn't the appropriate feature to use. Instead, you should implement a *next_number*-type table in which you can make the operation of bumping the number contained within it part of the larger transaction (and incur the serialization of queuing for this value).

To temporarily disable the automatic generation of values in an identity column, you use the `SET IDENTITY_INSERT tablename ON` option. In addition to filling in gaps in the identity sequence, this option is useful for tasks such as bulk-loading data in which the previous values already exist. For example, perhaps you're loading a new database with customer data from your previous system. You might want to preserve the previous customer numbers but have new ones automatically assigned using `IDENTITY`. The `SET` option was created exactly for cases like this. Because of the `SET` option's ability to allow you to determine your own values for an `IDENTITY` column, the `IDENTITY` property alone doesn't enforce uniqueness of a value within the table. Although `IDENTITY` will generate a unique number if `IDENTITY_INSERT` has never been enabled, the uniqueness is not guaranteed once you have used the `SET` option. To enforce uniqueness (which you'll almost always want to do when using `IDENTITY`), you should also declare a `UNIQUE` or `PRIMARY KEY` constraint on the column. If you insert your own values for an identity column (using `SET IDENTITY_INSERT`), when automatic generation resumes, the next value is the next incremented value (or decremented value) of the highest value that exists in the table, whether it was generated previously or explicitly inserted.

If you're using the *bcp* utility for bulk loading data, be aware of the -E (uppercase) parameter if your data already has assigned values that you want to keep for a column that has the IDENTITY property. You can also use the Transact-SQL BULK INSERT command with the KEEPIDENTITY option. For more information, see the SQL Server documentation for *bcp* and BULK INSERT.

The keyword IDENTITYCOL automatically refers to the specific column in a table that has the IDENTITY property, whatever its name. If that column is *cust_id*, you can refer to the column as IDENTITYCOL without knowing or using the column name or you can refer to it explicitly as

cust_id.

The following two statements work identically and return the same data:

```
SELECT IDENTITYCOL FROM customer
SELECT cust_id FROM customer
```

The column name returned to the caller is *cust_id*, not IDENTITYCOL, in both of these cases. When inserting rows, you must omit an identity column from the column list and VALUES section. (The only exception is when the IDENTITY_INSERT option is on.) If you do supply a column list, you must omit the column for which the value will be automatically supplied.

Here are two valid INSERT statements for the *customer* table shown earlier:

```
INSERT customer VALUES ('ACME Widgets')
INSERT customer (cust_name) VALUES ('AAA Gadgets')
```

Selecting these two rows produces this output:

```
cust_id  cust_name
-----  -
1        ACME Widgets
2        AAA Gadgets
(2 row(s) affected)
```

Sometimes in applications, it's desirable to immediately know the value produced by IDENTITY for subsequent use. For example, a transaction might first add a new customer and then add an order for that customer. To add the order, you probably need to use the *cust_id*. Rather than selecting the value from the *customer* table, you can simply select the special system function @@IDENTITY, which contains the last identity value used by that connection. It doesn't necessarily provide the last value inserted in the table, however, because another user might have subsequently inserted data. If multiple INSERT statements are carried out in a batch on the same or different tables, the variable has the value for the last statement only. In addition, if there is an INSERT trigger that fires after you insert the new row and if that trigger inserts rows into a table with an identity column, @@IDENTITY will not have the value inserted by the original INSERT statement.

It might look like you're inserting and then immediately checking the value:

```
INSERT customer (cust_name) VALUES ('AAA Gadgets')
SELECT @@IDENTITY
```

However, if a trigger was fired for the INSERT, the value of @@IDENTITY might have changed. There are two other functions that you might find useful when working with identity columns. SCOPE_IDENTITY returns the last identity value inserted into a table in the same scope, which

could be a stored procedure, trigger, or batch. So if we replace @@IDENTITY with the SCOPE_IDENTITY function in the code snippet above, we can see the identity value inserted into the customer table. If an INSERT trigger also inserted a row that contained an identity column, it would be in a different scope:

```
INSERT customer (cust_name) VALUES ('AAA Gadgets')
SELECT SCOPE_IDENTITY()
```

In other cases, you might want to know the last identity value inserted in a specific table, from any application or user. You can get this value using the IDENT_CURRENT function, which takes a table name as an argument:

```
SELECT IDENT_CURRENT('customer')
```

This doesn't always guarantee that you can predict the next identity value to be inserted, because another process could insert a row between the time you check the value of IDENT_CURRENT and the time you execute your INSERT statement.

You can't define the IDENTITY property as part of a UDDT, but you can declare the IDENTITY property on a column that uses a UDDT. A column that has the IDENTITY property must always be declared NOT NULL (either explicitly or implicitly); otherwise, error number 8147 will result from the CREATE TABLE statement and CREATE won't succeed. Likewise, you can't declare the IDENTITY property and a DEFAULT on the same column. To check that the current identity value is valid based on the current maximum values in the table, and to reset it if an invalid value is found (which should never be the case), use the DBCC CHECKIDENT(*tablename*) statement. Identity values are fully recoverable. If a system outage occurs while insert activity is taking place with tables that have identity columns, the correct value will be recovered when SQL Server is restarted. SQL Server accomplishes this during the checkpoint processing by flushing the current identity value for all tables. For activity beyond the last checkpoint, subsequent values are reconstructed from the transaction log during the standard database recovery process. Any inserts into a table that have the IDENTITY property are known to have changed the value, and the current value is retrieved from the last INSERT statement (post-checkpoint) for each table in the transaction log. The net result is that when the database is recovered, the correct current identity value is also recovered.

In rare cases, the identity value can get out of sync. If this happens, you can use the DBCC CHECKIDENT command to reset the identity value to the appropriate number. In addition, the RESEED option to this command allows you to set a new starting value for the identity sequence.

Constraints

Constraints provide a powerful yet easy way to enforce the data integrity in your database. Data integrity comes in three forms:

- Entity integrity ensures that a table has a primary key. In SQL Server 2000, you can guarantee entity integrity by defining PRIMARY KEY or UNIQUE constraints or by building unique indexes. Alternatively, you can write a trigger to enforce entity integrity, but this is usually far less efficient.
- Domain integrity ensures that data values meet certain criteria. In SQL Server 2000, domain integrity can be guaranteed in several ways. Choosing appropriate datatypes can ensure that a data value meets certain conditions—for example, that the data represents a valid date. Other approaches include defining CHECK constraints or FOREIGN KEY constraints or writing a trigger. You might also consider DEFAULT constraints as an aspect of enforcing domain integrity.
- Referential integrity enforces relationships between two tables, a referenced table, and a referencing table. SQL Server 2000 allows you to define FOREIGN KEY constraints to enforce referential integrity, and you can also write triggers for enforcement. It's crucial to

note that there are always two sides to referential integrity enforcement. If data is updated or deleted from the referenced table, referential integrity ensures that any data in the referencing table that refers to the changed or deleted data is handled in some way. On the other side, if data is updated or inserted into the referencing table, referential integrity ensures that the new data matches a value in the referenced table.

Constraints are also called *declarative data integrity* because they are part of the actual table definition. This is in contrast to *programmatic data integrity enforcement*, which uses stored procedures or triggers. Here are the five types of constraints:

- PRIMARY KEY
- UNIQUE
- FOREIGN KEY
- CHECK
- DEFAULT

You might also sometimes see the IDENTITY property and the nullability of a column described as constraints. I typically don't consider these attributes to be constraints; instead, I think of them as properties of a column, for two reasons. First, as we'll see, each constraint has its own row in the *sysobjects* system table, but IDENTITY and nullability information is not stored in *sysobjects*, only in *syscolumns*. This makes me think that these properties are more like datatypes, which are also stored in *syscolumns*. Second, when you use the special command SELECT INTO, a new table can be created that is a copy of an existing table. All column names and datatypes are copied, as well as IDENTITY information and column nullability. However, constraints are *not* copied to the new table. This makes me think that IDENTITY and nullability are more a part of the actual table structure than constraints are.

Primary key

PRIMARY KEY and UNIQUE Constraints

A central tenet of the relational model is that every row in a table is in some way unique and can be distinguished in some way from every other row in the table. You could use the combination of all columns in a table as this unique identifier, but the identifier is usually at most the combination of a handful of columns, and often it's just one column: the primary key. Although some tables might have multiple unique identifiers, each table can have only one primary key. For example, perhaps the *employee* table maintains both an *Emp_ID* column and an *SSN* (social security number) column, both of which can be considered unique. Such column pairs are often referred to as *alternate keys* or *candidate keys*, although both terms are design terms and aren't used by the ANSI SQL standard or by SQL Server. In practice, one of the two columns is logically promoted to primary key using the PRIMARY KEY constraint, and the other is usually declared by a UNIQUE constraint. Although neither the ANSI SQL standard nor SQL Server require it, it's good practice to declare a PRIMARY KEY constraint on every table. Furthermore, you must designate a primary key for a table that will be published for transaction-based replication. Internally, PRIMARY KEY and UNIQUE constraints are handled almost identically, so it will be discussed together here. Declaring a PRIMARY KEY or UNIQUE constraint simply results in a unique index being created on the specified column or columns, and this index enforces the column's uniqueness in the same way that a unique index created manually on a column would. The query optimizer makes decisions based on the presence of the unique index rather than on the fact that a column was declared as a primary key. How the index got there in the first place is irrelevant to the optimizer.

Nullability

All columns that are part of a primary key must be declared (either explicitly or implicitly) as NOT NULL. Columns that are part of a UNIQUE constraint can be declared to allow NULL. However, for the purposes of unique indexes, all NULLs are considered equal. So if the unique index is on a single column, only one NULL value can be stored (another good reason to try to avoid NULL whenever possible). If the unique index is on a composite key, one of the columns can have many NULLs as long as the value in the other column is unique.

If the constraint contains two int columns, exactly one row of each of these combinations will be allowed:

```
NULL  NULL
0     NULL
NULL  0
1     NULL
NULL  1
```

This behavior is questionable: NULL represents an unknown, but using it this way clearly implies that NULL is equal to NULL. As you'll recall, avoid using NULLs, especially in key columns.

Index Attributes

You can explicitly specify the index attributes CLUSTERED or NONCLUSTERED when you declare a constraint. If you don't, the index for a UNIQUE constraint will be nonclustered and the index for a PRIMARY KEY constraint will be clustered (unless CLUSTERED has already been explicitly stated for a unique index, because only one clustered index can exist per table). You can specify the index FILLFACTOR attribute if a PRIMARY KEY or UNIQUE constraint is added to an existing table using the ALTER TABLE command. FILLFACTOR doesn't make sense in a CREATE TABLE statement because the table has no existing data, and FILLFACTOR on an index affects how full pages are only when the index is initially created. FILLFACTOR isn't maintained when data is added.

Choosing Keys

Try to keep the key lengths as compact as possible. Columns that are the primary key or that are unique are most likely to be joined and frequently queried. Compact key lengths allow more index entries to fit on a given 8-KB page, thereby reducing I/O, increasing cache hits, and speeding up character matching. Clustered index keys are used as bookmarks in all your nonclustered indexes, so a long clustered key will increase the size and decrease the I/O efficiency of all your indexes. So if your primary key has a clustered index, you've got plenty of good reasons to keep it short. When no naturally efficient compact key exists, it's often useful to manufacture a surrogate key using the IDENTITY property on an int column. If int doesn't provide enough range, a good second choice is a numeric column with the required precision and with scale 0.

Alternatively, you can consider a bigint for an identity column. You might use this surrogate as the primary key, use it for most join and retrieval operations, and declare a UNIQUE constraint on the natural but inefficient columns that provide the logical unique identifier in your data. Or you might dispense with creating the UNIQUE constraint altogether if you don't need to have SQL Server enforce the uniqueness. Indexes slow performance of data modification statements because the index as well as the data must be maintained.

Although it's permissible to do so, don't create a PRIMARY KEY constraint on a column of type float or real. Because these are approximate datatypes, the uniqueness of such columns is also approximate, and the results can sometimes be unexpected.

Removing PRIMARY KEY or UNIQUE Constraints

You can't directly drop a unique index created as a result of a PRIMARY KEY or UNIQUE constraint by using the DROP INDEX statement. Instead, you must drop the constraint by using ALTER TABLE DROP CONSTRAINT (or you must drop the table itself). This feature was designed so that a constraint can't be compromised accidentally by someone who doesn't realize that the index is being used to enforce the constraint.

There is no way to temporarily disable a PRIMARY KEY or UNIQUE constraint. If disabling is required, use ALTER TABLE DROP CONSTRAINT and then later restore the constraint by using ALTER TABLE ADD CONSTRAINT. If the index you use to enforce uniqueness is clustered, when you add the constraint to an existing table, the entire table and all nonclustered indexes will be internally rebuilt to establish the cluster order. This can be a time-consuming task, and it requires about 1.2 times the existing table space as a temporary work area in the database would require (2.2 times total, counting the permanent space needed) so that the operation can be rolled back if necessary. However, you can rebuild an index that is enforcing a constraint by using the DROP_EXISTING option of the CREATE INDEX statement. This drops and re-creates the index behind the scenes as a single operation.

Creating PRIMARY KEY and UNIQUE Constraints

Typically, you declare PRIMARY KEY and UNIQUE constraints when you create the table (CREATE TABLE). However, you can add or drop both by subsequently using the ALTER TABLE command. To simplify matters, you can declare a PRIMARY KEY or UNIQUE constraint that includes only a single column on the same line that you define that column in the CREATE TABLE statement. Or you can declare the constraint as a separate line in your CREATE TABLE statement. The approach you use is largely a matter of personal preference unless your constraint involves multiple columns. Constraints defined along with a column definition can refer only to that column.

Four ways to declare a PRIMARY KEY constraint on a single column. They all cause a unique, clustered index to be created. Note the (abridged) output of the sp_helpconstraint procedure for each—especially the constraint name.

EXAMPLE 1

```
CREATE TABLE customer
(
  cust_id   int          IDENTITY NOT NULL PRIMARY KEY,
  cust_name varchar(30)  NOT NULL
)
GO
```

```
EXEC sp_helpconstraint customer
GO
```

```
>>>>
```

```
Object Name
-----
customer
```

```
constraint_type
```

constraint_name

PRIMARY KEY (clustered) PK__customer__68E79C55

EXAMPLE 2

```
CREATE TABLE customer
(
  cust_id   int      IDENTITY NOT NULL
           CONSTRAINT cust_pk PRIMARY KEY,
  cust_name varchar(30) NOT NULL
)
GO
```

```
EXEC sp_helpconstraint customer
GO
```

>>>>

Object Name

customer

constraint_type constraint_name

PRIMARY KEY (clustered) cust_pk

No foreign keys reference this table.

EXAMPLE 3

```
CREATE TABLE customer
(
  cust_id   int      IDENTITY NOT NULL,
  cust_name varchar(30) NOT NULL,
           PRIMARY KEY (cust_id)
)
GO
```

```
EXEC sp_helpconstraint customer
GO
```

>>>>

Object Name

customer


```
constraint_type      constraint_name
-----
PRIMARY KEY (clustered)  PK__customer__59063A47
```

No foreign keys reference this table.

EXAMPLE 4

```
CREATE TABLE customer
(
  cust_id  int      IDENTITY NOT NULL,
  cust_name varchar(30) NOT NULL,
  CONSTRAINT customer_PK PRIMARY KEY (cust_id)
)
GO
```

```
EXEC sp_helpconstraint customer
GO
```

```
>>>>
```

```
Object Name
-----
customer
```

```
constraint_type
-----
```

```
constraint_name
-----
```

```
PRIMARY KEY (clustered)  customer_PK
```

No foreign keys reference this table.

In Examples 1 and 3, an explicit name for the constraint is not provided, so SQL Server comes up with the names. The names, PK__customer__68E79C55 and PK__customer__59063A47, seem cryptic, but there is some method to this apparent madness. All types of single-column constraints use this same naming scheme, which will be discussed later in the chapter, and multi-column constraints use a similar scheme. Whether you choose a more intuitive name, such as cust_pk in Example 2 or customer_PK in Example 4, or the less intuitive (but information-packed), system-generated name is up to you. Here's an example of creating a multi-column, UNIQUE constraint on the combination of multiple columns. (The primary key case is essentially identical.)

```
CREATE TABLE customer_location
(
  cust_id          int NOT NULL,
  cust_location_number int NOT NULL,
  CONSTRAINT customer_location_unique UNIQUE
    (cust_id, cust_location_number)
)
GO
```

```
EXEC sp_helpconstraint customer_location
GO
```

```
>>>
```

```
Object Name
```

```
-----  
customer_location
```

```
constraint_type constraint_name constraint_keys
```

```
-----  
UNIQUE customer_location_unique cust_id,  
(non-clustered) cust_location_number
```

No foreign keys reference this table.

As noted earlier, a unique index is created to enforce either a PRIMARY KEY or a UNIQUE constraint. The name of the index is the same as the constraint name, whether the name was explicitly defined or system-generated. The index used to enforce the CUSTOMER_LOCATION_UNIQUE constraint in the above example is also named customer_location_unique. The index used to enforce the column-level, PRIMARY KEY constraint of the customer table in Example 1 is named PK__customer__68E79C55, which is the system-generated name of the constraint. You can use the sp_helpindex stored procedure to see information for all indexes of a given table.

```
EXEC sp_helpindex customer
```

```
>>>
```

```
index_name index_description index_keys
```

```
-----
```

```
-----  
PK__customer__68E79C55 clustered, unique, cust_id  
primary key  
located on default
```

Foreign key

FOREIGN KEY Constraints

One of the fundamental concepts of the relational model is the logical relationships between tables. In most databases, certain relationships must exist (that is, the data must have referential integrity) or else the data will be logically corrupt.

SQL Server automatically enforces referential integrity through the use of FOREIGN KEY constraints. This feature is sometimes referred to as declarative referential integrity, or DRI, to distinguish it from other features, such as triggers, that you can also use to enforce the existence of the relationships.

Referential Actions

The full ANSI SQL-92 standard contains the notion of the *referential action*, sometimes (incompletely) referred to as a *cascading delete*. SQL Server 7 doesn't provide this feature as

part of FOREIGN KEY constraints, but this notion warrants some discussion here because the capability exists via triggers.

The idea behind referential actions is this: sometimes, instead of just preventing an update of data that would violate a foreign key reference, you might be able to perform an additional, compensating action that will still enable the constraint to be honored. For example, if you were to delete a *customer* table that has references to *orders*, you can have SQL Server automatically delete all those related *order* records (that is, cascade the delete to *orders*), in which case the constraint won't be violated and the *customer* table can be deleted. This feature is intended for both UPDATE and DELETE statements, and four possible actions are defined: NO ACTION, CASCADE, SET DEFAULT, and SET NULL.

- **NO ACTION** The delete is prevented. This default mode, per the ANSI standard, occurs if no other action is specified. NO ACTION is often referred to as RESTRICT, but this usage is slightly incorrect in terms of how ANSI defines RESTRICT and NO ACTION. ANSI uses RESTRICT in DDL statements such as DROP TABLE, and it uses NO ACTION for FOREIGN KEY constraints. (The difference is subtle and unimportant. It's common to refer to the FOREIGN KEY constraint as having an action of RESTRICT.)
- **CASCADE** A delete of all matching rows in the referencing table occurs.
- **SET DEFAULT** The delete is performed, and all foreign key values in the referencing table are set to a default value.
- **SET NULL** The delete is performed, and all foreign key values in the referencing table are set to NULL.

SQL Server 2000 allows you to specify either NO ACTION or CASCADE when you define your foreign key constraints. If you want to implement either SET DEFAULT or SET NULL, you can use a trigger. Implementing the SET NULL action is very straightforward, which discusses triggers in detail. One reason it's so straightforward is that while a FOREIGN KEY requires any value in the referencing table to match a value in the referenced table, NULLs are not considered to be a value, and having a NULL in the referencing table doesn't break any relationships.

Implementing the SET DEFAULT action is just a little more involved. Suppose you want to remove a row from the referenced table and set the foreign key value in all referencing rows to -9999. If you want to maintain referential integrity, you have to have a "dummy" row in the referenced table with the primary key value of -9999.

Keep in mind that enforcing a foreign key implies that SQL Server will check data in both the referenced and the referencing tables. The referential actions just discussed apply only to actions that are taken on the referenced table. On the referencing table, the only action allowed is to not allow an update or an insert if the relationship will be broken as a result.

Because a constraint is checked before a trigger fires, you can't have both a FOREIGN KEY constraint to enforce the relationship when a new key is inserted into the referencing table and a trigger that performs an operation such as SET NULL on the referenced table. If you do have both, the constraint will fail and the statement will be aborted before the trigger to cascade the delete fires. If you want to allow the SET NULL action, you have to write two triggers. You have to have a trigger for delete and update on the referenced table to perform the SET NULL action and a trigger for insert and update on the referencing table that disallows new data that would violate the referential integrity.

If you do decide to enforce your referential integrity with triggers, you might still want to declare the foreign key relationship largely for readability so that the relationship between the tables is clear. You can then use the NOCHECK option of ALTER TABLE to disable the constraint, and then the trigger will fire.

This note is just referring to AFTER triggers. SQL Server 2000 provides an additional type of trigger called an INSTEAD OF trigger.

Here's a simple way to declare a primary key/foreign key relationship:

```
CREATE TABLE customer
(
cust_id int NOT NULL IDENTITY PRIMARY KEY,
cust_name varchar(50) NOT NULL
)
```

```
CREATE TABLE orders
```

```
(
order_id int NOT NULL IDENTITY PRIMARY KEY,
cust_id int NOT NULL REFERENCES customer(cust_id)
)
```

Since the default referential action is NO ACTION, the orders table could also have been written in this way:

```
CREATE TABLE orders
```

```
(
order_id int NOT NULL IDENTITY PRIMARY KEY,
cust_id int NOT NULL REFERENCES customer(cust_id)
ON UPDATE NO ACTION ON DELETE NO ACTION
)
```

The *orders* table contains the column *cust_id*, which references the primary key of the *customer* table. An order (*order_id*) must not exist unless it relates to an existing customer (*cust_id*). With this definition, you can't delete a row from the *customer* table if a row that references it currently exists in the *orders* table, and you can't modify the *cust_id* column in a way that would destroy the relationship.

A referential action can be specified for both DELETE and UPDATE operations on the referenced table, and the two operations can have different actions.

You can choose to CASCADE any updates to the *cust_id* in *customer* but not allow (NO ACTION) any deletes of referenced rows in *customer*.

Adding a FOREIGN KEY constraint to the *orders* table that does just that: it cascades any updates to the *cust_id* in *customer* but does not allow any deletes of referenced rows in *customer*.

```
CREATE TABLE orders
```

```
(
order_id int NOT NULL IDENTITY PRIMARY KEY,
cust_id int NOT NULL REFERENCES customer(cust_id)
ON UPDATE CASCADE ON DELETE NO ACTION
)
```

The previous examples show the syntax for a single-column constraint—both the primary key and foreign key are single columns. This syntax uses the keyword REFERENCES, and the term "foreign key" is implied but not explicitly stated. The name of the FOREIGN KEY constraint is generated internally, following the same general form described earlier for PRIMARY KEY and UNIQUE constraints. Here's a portion of the output of *sp_helpconstraint* for both the *customer* and *orders* tables. (The tables were created in the *pubs* sample database.)

```
EXEC sp_helpconstraint customer
>>>
```

Object Name

```

-----
customer

constraint_type  constraint_name      constraint_keys
-----
PRIMARY KEY      PK__customer__07F6335A  cust_id
(clustered)

```

```

Table is referenced by
-----
pubs.dbo.orders: FK__orders__cust_id__0AD2A005

```

```

EXEC sp_helpconstraint orders
>>>

```

```

Object Name
-----
orders

constraint_type  constraint_name      delete_  update_
                  action  action
-----
FOREIGN KEY      FK__orders__cust_id__0AD2A005  no action no action
PRIMARY KEY      PK__orders__09DE7BCC
(clustered)

```

```

constraint_keys
-----
cust_id REFERENCES pubs.dbo.customer(cust_id)
order_id

```

No foreign keys reference this table. Constraints defined inline with the column declaration can reference only a single column. This is also true for PRIMARY KEY and UNIQUE constraints. You must declare a multi-column constraint as a separate table element. The following example shows a multi-column FOREIGN KEY constraint:

```

CREATE TABLE customer
(
  cust_id      int          NOT NULL,
  location_num smallint     NULL,
  cust_name    varchar(50)  NOT NULL,
  CONSTRAINT CUSTOMER_UNQ UNIQUE CLUSTERED (location_num, cust_id)
)

CREATE TABLE orders
(
  order_id int          NOT NULL IDENTITY CONSTRAINT ORDER_PK
              PRIMARY KEY NONCLUSTERED,
  cust_num  int          NOT NULL,
  cust_loc  smallint     NULL,

```

```
CONSTRAINT FK_ORDER_CUSTOMER FOREIGN KEY (cust_loc, cust_num)
REFERENCES customer (location_num, cust_id)
)
GO
```

```
EXEC sp_helpconstraint customer
EXEC sp_helpconstraint orders
GO
```

```
>>>
```

```
Object Name
```

```
-----
customer
```

```
constraint_type  constraint_name  constraint_keys
-----
UNIQUE (clustered)  CUSTOMER_UNQ    location_num, cust_id
```

```
Table is referenced by
```

```
-----
pubs.dbo.orders: FK_ORDER_CUSTOMER
```

```
Object Name
```

```
-----
orders
```

```
constraint_type  constraint_name  constraint_keys
-----
FOREIGN KEY      FK_ORDER_CUSTOMER  cust_loc, cust_num
                  REFERENCES pubs.dbo.customer
                  (location_num, cust_id)

PRIMARY KEY      ORDER_PK          order_id
(non-clustered)
```

No foreign keys reference this table.

The following example shows the following variations on how you can create constraints:

- You can use a FOREIGN KEY constraint to reference a UNIQUE constraint (an alternate key) instead of a PRIMARY KEY constraint. (Note, however, that referencing a PRIMARY KEY is much more typical and is generally better practice.)
- You don't have to use identical column names in tables involved in a foreign key reference, but doing so is often good practice. The *cust_id* and *location_num* column names are defined in the *customer* table. The *orders* table, which references the *customer* table, uses the names *cust_num* and *cust_loc*.
- The datatypes of the related columns must be identical, except for nullability and variable-length attributes. (For example, a column of *char(10) NOT NULL* can reference one of *varchar(10) NULL*, but it can't reference a column of *char(12) NOT NULL*. A column of type *smallint* can't reference a column of type *int*.) Notice in the preceding example that *cust_id* and *cust_num* are both *int NOT NULL* and that *location_num* and *cust_loc* are both *smallint NULL*.

Unlike a PRIMARY KEY or UNIQUE constraint, an index isn't automatically built for the column or columns declared as FOREIGN KEY. However, in many cases, you'll want to build indexes on these columns because they're often used for joining to other tables. To enforce foreign key relationships, SQL Server must add additional steps to the execution plan of every insert, delete, and update (if the update affects columns that are part of the relationship) that affects either the table referencing another table or the table being referenced. The execution plan, determined by the SQL Server optimizer, is simply the collection of steps that carries out the operation. If no FOREIGN KEY constraints exist, a statement specifying the update of a single row of the *orders* table might have an execution plan like the following:

1. Find a qualifying *order* record using a clustered index.
2. Update the *order* record.

When a FOREIGN KEY constraint exists on the *orders* table, the same operation has more steps in the execution plan:

1. Check for the existence of a related record in the *customer* table (based on the updated *order* record) using a clustered index.
2. If no related record is found, raise an exception and terminate the operation.
3. Find a qualifying *order* record using a clustered index.
4. Update the *order* record.

The execution plan is more complex if the *orders* table has many FOREIGN KEY constraints declared. Internally, a simple update or insert operation might no longer be possible. Any such operation requires checking many other tables for matching entries. Because a seemingly simple operation might require checking as many as 253 other tables (see the next paragraph) and possibly creating multiple worktables, the operation might be much more complicated than it looks and much slower than expected.

A table can have a maximum of 253 FOREIGN KEY references. This limit is derived from the internal limit of 256 tables in a single query. In practice, an operation on a table with 253 or fewer FOREIGN KEY constraints might still fail with an error because of the 256-table query limit if worktables are required for the operation.

A database designed for excellent performance doesn't reach anything close to this limit of 253 FOREIGN KEY references. For best performance results, use FOREIGN KEY constraints judiciously. Some sites unnecessarily use too many FOREIGN KEY constraints because the constraints they declare are logically redundant. Take the following example. The *orders* table declares a FOREIGN KEY constraint to both the *master_customer* and *customer_location* tables:

```
CREATE TABLE master_customer
```

```
(  
  cust_id  int          NOT NULL IDENTITY PRIMARY KEY,  
  cust_name varchar(50) NOT NULL  
)
```

```
CREATE TABLE customer_location
```

```
(  
  cust_id  int          NOT NULL,  
  cust_loc smallint    NOT NULL,  
  CONSTRAINT PK_CUSTOMER_LOCATION PRIMARY KEY (cust_id,cust_loc),  
  CONSTRAINT FK_CUSTOMER_LOCATION FOREIGN KEY (cust_id)  
    REFERENCES master_customer (cust_id)  
)
```

```
CREATE TABLE orders
```

```
(  
  
  order_id int          NOT NULL IDENTITY PRIMARY KEY,
```

```
cust_id int NOT NULL,  
cust_loc smallint NOT NULL,  
CONSTRAINT FK_ORDER_MASTER_CUST FOREIGN KEY (cust_id)  
REFERENCES master_customer (cust_id),  
CONSTRAINT FK_ORDER_CUST_LOC FOREIGN KEY (cust_id, cust_loc)  
REFERENCES customer_location (cust_id, cust_loc)  
)
```

Although logically the relationship between the *orders* and *master_customer* tables exists, the relationship is redundant to and subsumed by the fact that *orders* is related to *customer_location*, which has its own FOREIGN KEY constraint to *master_customer*. Declaring a foreign key for *master_customer* adds unnecessary overhead without adding any further integrity protection.

In the case just described, declaring a foreign key improves the readability of the table definition, but you can achieve the same result by simply adding comments to the CREATE TABLE command. It's perfectly legal to add a comment practically anywhere—even in the middle of a CREATE TABLE statement. A more subtle way to achieve this result is to declare the constraint so that it appears in *sp_helpconstraint* and in the system catalogs but then disable the constraint by using the ALTER TABLE NOCHECK option. Because the constraint will then be unenforced, an additional table isn't added to the execution plan.

The CREATE TABLE statement shown in the following example for the *orders* table omits the redundant foreign key and, for illustrative purposes, includes a comment. Despite the lack of a FOREIGN KEY constraint in the *master_customer* table, you still can't insert a *cust_id* that doesn't exist in the *master_customer* table because the reference to the *customer_location* table will prevent it.

```
CREATE TABLE orders  
(  
order_id int NOT NULL IDENTITY PRIMARY KEY,  
cust_id int NOT NULL,  
cust_loc smallint NOT NULL,  
--- Implied Foreign Key Reference of:  
--- (cust_id) REFERENCES master_customer (cust_id)  
CONSTRAINT FK_ORDER_CUST_LOC FOREIGN KEY (cust_id, cust_loc)  
REFERENCES customer_location (cust_id, cust_loc)  
)
```

Note that the table on which the foreign key is declared (the referencing table, which in this example is *orders*) isn't the only table that requires additional execution steps. When being updated, the table being referenced (in this case *customer_location*) must also have additional steps in its execution plan to ensure that an update of the columns being referenced won't break a relationship and create an orphan entry in the *orders* table. Without making any changes directly to the *customer_location* table, you can see a significant decrease in update or delete performance because of foreign key references added to other tables.

Practical Considerations for FOREIGN KEY Constraints

When using constraints, you should consider triggers, performance, and indexing. Let's take a look at the ramifications of each. Constraints are enforced before an AFTER triggered action is performed. If the constraint is violated, the statement will abort before the trigger fires.

The owner of a table isn't allowed to declare a foreign key reference to another table unless the owner of the other table has granted REFERENCES permission to the first table owner. Even if the owner of the first table is allowed to select from the table to be referenced, that owner must

have REFERENCES permission. This prevents another user from changing the performance of operations on your table without your knowledge or consent. You can grant any user REFERENCES permission even if you don't also grant SELECT permission, and vice-versa. The only exception is that the DBO, or any user who is a member of the *db_owner* role, has full default permissions on all objects in the database.

Performance When deciding on the use of foreign key relationships, you must weigh the protection provided against the corresponding performance overhead. Be careful not to add constraints that form logically redundant relationships. Excessive use of FOREIGN KEY constraints can severely degrade the performance of seemingly simple operations.

Indexing The columns specified in FOREIGN KEY constraints are often strong candidates for index creation. You should build the index with the same key order used in the PRIMARY KEY or UNIQUE constraint of the table that it references so that joins can be performed efficiently. Also be aware that a foreign key is often a subset of the table's primary key. In the *customer_location* table used in the preceding two examples, *cust_id* is part of the primary key as well as a foreign key in its own right. Given that *cust_id* is part of a primary key, it's already part of an index. In this example, *cust_id* is the lead column of the index, and building a separate index on it alone probably isn't warranted. However, if *cust_id* is *not* the lead column of the index B-tree, it might make sense to build an index on it.

Constraints

Constraint-Checking Solutions

Sometimes two tables reference one another, which creates a bootstrap problem. Suppose *Table1* has a foreign key reference to *Table2*, but *Table2* has a foreign key reference to *Table1*. Even before either table contains any data, you'll be prevented from inserting a row into *Table1* because the reference to *Table2* will fail. Similarly, you can't insert a row into *Table2* because the reference to *Table1* will fail.

ANSI SQL has a solution: *deferred constraints*, in which you can instruct the system to postpone constraint checking until the entire transaction is committed. Using this elegant remedy puts both INSERT statements into a single transaction that results in the two tables having correct references by the time COMMIT occurs. Unfortunately, no mainstream product currently provides the deferred option for constraints. The deferred option is part of the complete SQL-92 specification, which no product has yet fully implemented.

SQL Server 2000 provides *immediate* constraint checking; it has no deferred option. SQL Server offers three options for dealing with constraint checking: it allows you to add constraints after adding data, it lets you temporarily disable checking of foreign key references, and it allows you to use the *bcp* (bulk copy) program or BULK INSERT command to initially load data and avoid checking FOREIGN KEY constraints. (You can override this default option with *bcp* or the BULK INSERT command and force FOREIGN KEY constraints to be validated.) To add constraints after adding data, don't create constraints via the CREATE TABLE command. After adding the initial data, you can add constraints by using the ALTER TABLE command.

With the second option, the table owner can temporarily disable checking of foreign key references by using the ALTER TABLE *table* NOCHECK CONSTRAINT statement. Once data exists, you can reestablish the FOREIGN KEY constraint by using ALTER TABLE *table* CHECK CONSTRAINT. Note that when an existing constraint is reenabled using this method, SQL Server doesn't automatically check to see that all rows still satisfy the constraint. To do this, you can simply issue a *dummy update* by setting a column to itself for all rows, determining whether any constraint violations are raised, and then fixing them. (For example, you can issue UPDATE orders SET *cust_id* = *cust_id*.)

Finally you can use the *bcp* program or the BULK INSERT command to initially load data. The BULK INSERT command and the *bcp* program don't check any FOREIGN KEY constraints by default. You can use the CHECK_CONSTRAINTS option to override this behavior. BULK INSERT and *bcp* are faster than regular INSERT commands because they usually bypass normal integrity checks and most logging.

When you use ALTER TABLE to add (instead of reenabling) a new FOREIGN KEY constraint for a table in which data already exists, the existing data is checked by default. If constraint violations occur, the constraint isn't added. With large tables, such a check can be quite time-consuming. You do have an alternative—you can add a FOREIGN KEY constraint and omit the check. To do this, you specify the WITH NOCHECK option with ALTER TABLE. All subsequent operations will be checked, but existing data won't be checked. As in the case of reenabling a constraint, you can then perform a dummy update to flag any violations in the existing data. If you use this option, you should do the dummy update as soon as possible to ensure that all the data is clean. Otherwise, your users might see constraint error messages when they perform update operations on the preexisting data even if they haven't changed any values.

Restrictions on Dropping Tables

If you're dropping tables, you must drop all the referencing tables or drop the referencing FOREIGN KEY constraints before dropping the referenced table. For example, in the preceding example's *orders*, *customer_location*, and *master_customer* tables, the following sequence of DROP statements fails because a table being dropped is referenced by a table that still exists—that is, *customer_location* can't be dropped because the *orders* table references it, and *orders* isn't dropped until later:

```
DROP TABLE customer_location
DROP TABLE master_customer
DROP TABLE orders
```

Changing the sequence to the following works fine because *orders* is dropped first:

```
DROP TABLE orders
DROP TABLE customer_location
DROP TABLE master_customer
```

When two tables reference each other, you must first drop the constraints or set them to NOCHECK (both operations use ALTER TABLE) before the tables can be dropped. Similarly, a table that's being referenced can't be part of a TRUNCATE TABLE command. You must drop or disable the constraint or else simply drop and rebuild the table.

Self-Referencing Tables

A table can be self-referencing—that is, the foreign key can reference one or more columns in the same table. The following example shows an employee table in which a column for managers references another *employee* entry:

```
CREATE TABLE employee
(
  emp_id   int           NOT NULL PRIMARY KEY,
  emp_name varchar(30)  NOT NULL,
  mgr_id   int           NOT NULL REFERENCES employee(emp_id)
)
```

The *employee* table is a perfectly reasonable table. However, in this case, a single INSERT command that satisfies the reference is legal. For example, if the CEO of the company has an

emp_id of 1 and is also his own manager, the following INSERT will be allowed and can be a useful way to insert the first row in a self-referencing table:

```
INSERT employee VALUES (1, 'Chris Smith', 1)
```

Although SQL Server doesn't currently provide a deferred option for constraints, self-referencing tables add a twist that sometimes makes SQL Server use deferred operations internally. Consider the case of a nonqualified DELETE statement that deletes many rows in the table. After all rows are ultimately deleted, you can assume that no constraint violation will occur. However, violations might occur *during* the DELETE operation because some of the remaining referencing rows might be orphaned before they are actually deleted. SQL Server handles such *interim violations* automatically and without any user intervention. As long as the self-referencing constraints are valid at the end of the data modification statement, no errors are raised during processing.

To gracefully handle these interim violations, however, additional processing and worktables are required to hold the work-in-progress. This adds substantial overhead and can also limit the actual number of foreign keys that can be used. An UPDATE statement can also cause an interim violation. For example, if all employee numbers are to be changed by multiplying each by 1000, the following UPDATE statement would require worktables to avoid the possibility of raising an error on an interim violation:

```
UPDATE employee SET emp_id=emp_id * 1000, mgr_id=mgr_id * 1000
```

The additional worktables and the processing needed to handle the worktables are made part of the execution plan. Therefore, if the optimizer sees that a data modification statement *could* cause an interim violation, the additional temporary worktables will be created even if no such interim violations ever actually occur. These extra steps are needed only in the following situations:

A table is self-referencing (it has a FOREIGN KEY constraint that refers back to itself).

- A single data modification statement (UPDATE, DELETE, or INSERT based on a SELECT) is performed and can affect more than one row. (The optimizer can't determine *a priori*, based on the WHERE clause and unique indexes, whether more than one row could be affected.) Multiple data modification statements within the transaction don't apply—this condition must be a single statement that affects multiple rows.
- Both the referencing and referenced columns are affected (which is always the case for DELETE and INSERT operations, but might or might not be the case for UPDATE).

If a data modification statement in your application meets the preceding criteria, you can be sure that SQL Server is automatically using a limited and special-purpose form of deferred constraints to protect against interim violations.

CHECK Constraints

Enforcing *domain integrity* (that is, ensuring that only entries of expected types, values, or ranges can exist for a given column) is also important. SQL Server provides two ways to enforce domain integrity: CHECK constraints and rules. CHECK constraints allow you to define an expression for a table that must not evaluate to FALSE for a data modification statement to succeed. The constraint will allow the row if it evaluates to TRUE or to unknown. The constraint evaluates to unknown when NULL values are present, and this introduces three-value logic. SQL Server provides a similar mechanism to CHECK constraints, called rules, which are provided basically for backward compatibility reasons. Rules perform almost the same function as CHECK constraints, but they use different syntax and have fewer capabilities.

CHECK constraints make a table's definition more readable by including the domain checks in the DDL. Rules have a potential advantage in that they can be defined once and then bound to

multiple columns and tables (using *sp_bindrule* each time), while you must respecify a CHECK constraint for each column and table. But the extra binding step can also be a hassle, so this capability for rules is beneficial only if a rule will be used in many places. Although performance between the two approaches is identical, CHECK constraints are generally preferred over rules because they're a direct part of the table's DDL, they're ANSI-standard SQL, they provide a few more capabilities than rules (such as the ability to reference other columns in the same row or to call a system function), and perhaps most important, they're more likely than rules to be further enhanced in future releases of SQL Server.

CHECK constraints add additional steps to the execution plan to ensure that the expression doesn't evaluate to FALSE (which would result in the operation being aborted). Although steps are added to the execution plan for data modifications, these are typically much less expensive than the extra steps discussed earlier for FOREIGN KEY constraints. For foreign key checking, another table must be searched, requiring additional I/O. CHECK constraints deal only with some logical expression for the specific row already being operated on, so no additional I/O is required. Because additional processing cycles are used to evaluate the expressions, the system requires more CPU use. But if there's plenty of CPU power to spare, the effect might well be negligible. You can watch this by using Performance Monitor.

As with other constraint types, you can declare CHECK constraints on a single column or on multiple columns. You must declare a CHECK constraint that refers to more than one column as a separate element in the CREATE TABLE statement. Only single-column CHECK constraints can be defined right along with the column definition, and only one CHECK constraint can be defined with the column. All other CHECK constraints have to be defined as separate elements. However, keep in mind that one constraint can have multiple logical expressions that can be AND'ed or OR'ed together.

Some CHECK constraint features have often been underutilized by SQL Server database designers, including the ability to reference other columns in the same row, use system and niladic functions (which are evaluated at runtime), use combinations of expressions combined with AND or OR, and use CASE expressions. The following example shows a table with multiple CHECK constraints (as well as a PRIMARY KEY constraint and a FOREIGN KEY constraint) and showcases some of these features:

```
CREATE TABLE employee
(
emp_id      int          NOT NULL PRIMARY KEY
              CHECK (emp_id BETWEEN 0 AND 1000),

emp_name    varchar(30) NOT NULL CONSTRAINT no_nums
              CHECK (emp_name NOT LIKE '%[0-9]%),

mgr_id      int          NOT NULL REFERENCES employee(emp_id),

entered_date datetime  NULL CHECK (entered_date >=
              CURRENT_TIMESTAMP),

entered_by  int          CHECK (entered_by IS NOT NULL),
              CONSTRAINT valid_entered_by CHECK
              (entered_by = SUSER_ID(NULL) AND
              entered_by <> emp_id),

CONSTRAINT valid_mgr CHECK (mgr_id <> emp_id OR emp_id=1),

CONSTRAINT end_of_month CHECK (DATEPART(DAY, GETDATE()) < 28)
)
GO

EXEC sp_helpconstraint employee
```

GO
>>>>

Object Name

employee

constraint_type	constraint_name	constraint_keys
CHECK on column emp_id	CK__employee__emp_id__2C3393D0	([emp_id] >= 0 and [emp_id] <=1000)
CHECK on column entered_by	CK__employee__entered_by__300424B4	((not(entered_by is null)))
CHECK on column entered_date	CK__employee__entered_date__2F10007B	([entered_date] >= getdate())
CHECK Table Level end_of_month		(datepart(day, getdate()) <28)
FOREIGN KEY CHECK on column emp_name	FK__employee__mgr_id__2E1BDC42 no_nums	mgr_id ([emp_name] not like '%[0-9]%')
PRIMARY KEY (clustered) CHECK Table Level valid_entered_by	PK__employee__2B3F6F97	emp_id ([entered_by] = suser_id(null) and [entered_by] <> [emp_id])
CHECK Table Level valid_mgr		([mgr_id] <> [emp_id] or [emp_id] = 1)

Table is referenced by

pubs.dbo.employee: FK__employee__mgr_id__2E1BDC42
This example illustrates the following points:

- CHECK constraints can be expressed at the column level with abbreviated syntax (leaving naming to SQL Server), such as the check on *entered_date*; at the column level with an explicit name, such as the NO_NUMS constraint on *emp_name*; or as a table-level constraint, such as the VALID_MGR constraint.
- CHECK constraints can use regular expressions—for example, NO_NUMS ensures that a digit can never be entered as a character in a person's name.
- Expressions can be AND'ed and OR'ed together to represent more complex situations—for example, VALID_MGR. However, SQL Server won't check for logical correctness at the time a constraint is added to the table. Suppose you want to restrict the values in a *department_id* column to either negative numbers or values greater than 100. (Perhaps numbers between 0 and 100 are reserved.) You can add this column and constraint to the table using ALTER TABLE

```
ALTER TABLE employee
add department_no int CHECK
(department_no < 0 AND department_no > 100)
```

- Table-level CHECK constraints can refer to more than one column in the same row. For example, VALID_MGR means that no employee can be his own boss, except

- employee number 1, who is assumed to be the CEO. SQL Server currently has no provision that allows you to check a value from another row or from a different table.
- You can make a CHECK constraint prevent NULL values—for example, CHECK (*entered_by* IS NOT NULL). Generally, you simply declare the column NOT NULL.
 - A NULL column might make the expression logically unknown. For example, a NULL value for *entered_date* gives CHECK *entered_date* >= CURRENT_TIMESTAMP an unknown value. This doesn't reject the row, however. The constraint rejects the row only if the expression is clearly false, even if it isn't necessarily true.
 - You can use system functions, such as GETDATE, APP_NAME, DATALENGTH, and SUSER_ID, as well as scalar functions, such as SYSTEM_USER, CURRENT_TIMESTAMP, and USER, in CHECK constraints. This subtle feature is powerful and can be useful, for example, to ensure that a user can change only records that she has entered by comparing *entered_by* to the user's system ID, as generated by SUSER_ID (or by comparing *emp_name* to SYSTEM_USER). Note that scalar functions such as CURRENT_TIMESTAMP are provided for ANSI SQL conformance and simply map to an underlying SQL Server function, in this case GETDATE. So while the DDL to create the constraint on *entered_date* uses CURRENT_TIMESTAMP, *sp_helpconstraint* shows it as GETDATE, which is the underlying function. Either expression is valid, and they are equivalent in the CHECK constraint. The VALID_ENTERED_BY constraint ensures that the *entered_by* column can be set only to the currently connected user's ID, and it ensures that users can't update their own records.
 - A constraint defined as a separate table element can call a system function without referencing a column in the table. In the example preceding this list, the END_OF_MONTH CHECK constraint calls two date functions, DATEPART and GETDATE, to ensure that updates can't be made after day 27 of the month (which is when the business's payroll is assumed to be processed). The constraint never references a column in the table. Similarly, a CHECK constraint might call the APP_NAME function to ensure that updates can be made only from an application of a certain name, instead of from an ad hoc tool such as SQL Query Analyzer.

As with FOREIGN KEY constraints, you can add or drop CHECK constraints by using ALTER TABLE. When adding a constraint, by default the existing data is checked for compliance; you can override this default by using the WITH NOCHECK syntax. You can later do a dummy update to check for any violations. The table or database owner can also temporarily disable CHECK constraints by using NOCHECK in the ALTER TABLE statement.

Default Constraints

A default allows you to specify a constant value, NULL, or the run-time value of a system function if no known value exists or if the column is missing in an INSERT statement. Although you could argue that a default isn't truly a constraint (because a default doesn't enforce anything), you can create defaults in a CREATE TABLE statement using the CONSTRAINT keyword; therefore, henceforth defaults will be referred to as constraints. Defaults add little overhead, and you can use them liberally without too much concern about performance degradation.

SQL Server provides two ways of creating defaults. First, you can create a default (CREATE DEFAULT) and then bind the default to a column (*sp_bindefault*). This has been the case since the original SQL Server release. Second, you can use default constraints. These were introduced in 1995 with SQL Server 6.0 as part of the CREATE TABLE and ALTER TABLE statements, and they're based on the ANSI SQL standard (which includes such niceties as being able to use system functions). Using defaults is pretty intuitive. The type of default you use is a matter of preference; both perform the same function internally. Future enhancements are likely to be made to the ANSI-style implementation.

CHECK constraint discussion, now modified to include several defaults:

```
CREATE TABLE employee
(
emp_id      int      NOT NULL PRIMARY KEY DEFAULT 1000
            CHECK (emp_id BETWEEN 0 AND 1000),

emp_name    varchar(30) NULL DEFAULT NULL CONSTRAINT no_nums
            CHECK (emp_name NOT LIKE '%[0-9]%),

mgr_id      int      NOT NULL DEFAULT (1) REFERENCES
            employee(emp_id),

entered_date datetime NOT NULL CHECK (entered_date >=
            CONVERT(char(10), CURRENT_TIMESTAMP, 102))
            CONSTRAINT def_today DEFAULT
            (CONVERT(char(10), GETDATE(), 102)),

entered_by  int      NOT NULL DEFAULT SUSER_ID()
            CHECK (entered_by IS NOT NULL),

CONSTRAINT valid_entered_by CHECK (entered_by=SUSER_ID() AND
entered_by <> emp_id),

CONSTRAINT valid_mgr CHECK (mgr_id <> emp_id OR emp_id=1),

CONSTRAINT end_of_month CHECK (DATEPART(DAY, GETDATE()) < 28)
)
GO
```

```
EXEC sp_helpconstraint employee
GO
```

>>>

Object Name

Employee

constraint_type	constraint_name	constraint_keys
-----	-----	-----
CHECK on column emp_id	CK__employee__emp_id__2C3393D0 [emp_id] <= 1000)	([emp_id] >= 0
CHECK on column entered_by	CK__employee__entered_by__300424B4 [entered_by] is null)))))	((not(
DEFAULT on column entered_date	def_today convert(char(10), getdate(), 102))	(1000)
DEFAULT on column emp_id	DF__employee__emp_id__35BCFE0A	(1000)

```

DEFAULT on column DF__employee__emp_name__37A5467C (null)
emp_name
DEFAULT on column DF__employee__entered_by__3D5E1FD2 (suser_id())
entered_by
DEFAULT on column DF__employee__mgr_id__398D8EEE (1)
mgr_id
CHECK on column CK__employee__entered_date__2F10007B ([entered_date]
entered_date >= getdate())
CHECK Table Level end_of_month (datepart(day,
getdate()) <
28)
FOREIGN KEY FK__employee__mgr_id__2E1BDC42 mgr_id
CHECK on column no_nums ([emp_name]
emp_name not like
'%[0-9]%' )

PRIMARY KEY PK__employee__2B3F6F97 emp_id
(clustered)

CHECK Table Level valid_entered_by (
[entered_by] =
suser_id(null)
and
[entered_by]
<> [emp_id])

CHECK Table Level valid_mgr ([mgr_id] <>
[emp_id] or
[emp_id] = 1)

```

Table is referenced by

pubs.dbo.employee: FK__employee__mgr_id__2E1BDC42

The preceding code demonstrates the following about defaults:

A default constraint is always a single-column constraint because it pertains only to one column and can be defined only along with the column definition; you cannot define a default constraint as a separate table element. You can use the abbreviated syntax that omits the keyword CONSTRAINT and the specified name, letting SQL Server generate the name, or you can specify the name by using the more verbose CONSTRAINT name DEFAULT syntax.

- A default value can clash with a CHECK constraint. This problem appears only at runtime, not when you create the table or when you add the default using ALTER TABLE. For example, a column with a default of 0 and a CHECK constraint that states that the value must be greater than 0 can never insert or update the default value.
- Although you can assign a default to a column that has a PRIMARY KEY or a UNIQUE constraint, it doesn't make much sense to do so. Such columns must have unique values, so only one row could exist with the default value in that column. The preceding example sets a DEFAULT on a primary key column for illustration, but in general, this practice is unwise.

- You can write a constant value within parentheses, as in `DEFAULT (1)`, or without them, as in `DEFAULT 1`. A character or date constant must be enclosed in either single or double quotation marks.
- One tricky concept is knowing when a `NULL` is inserted into a column as opposed to a default value. A column declared `NOT NULL` with a default defined uses the default only under one of the following conditions:
 - The `INSERT` statement specifies its column list and omits the column with the default.

The `INSERT` statement specifies the keyword `DEFAULT` in the values list (whether the column is explicitly specified as part of the column list or implicitly specified in the values list and the column list is omitted, meaning "All columns in the order in which they were created"). If the values list explicitly specifies `NULL`, an error is raised and the statement fails; the default value isn't used. If the `INSERT` statement omits the column entirely, the default is used and no error occurs. (This behavior is in accordance with ANSI SQL.) The keyword `DEFAULT` can be used in the values list, and this is the only way the default value will be used if a `NOT NULL` column is specified in the column list of an `INSERT` statement (either, as in the following example, by omitting the column list—which means all columns—or by explicitly including the `NOT NULL` column in the columns list).

`INSERT EMPLOYEE VALUES (1, 'The Big Guy', 1, DEFAULT, DEFAULT)`

Table 6-10 summarizes the behavior of `INSERT` statements based on whether a column is declared `NULL` or `NOT NULL` and whether it has a default specified. It shows the result for the column for three cases:

- Omitting the column entirely (no entry)
- Having the `INSERT` statement use `NULL` in the values list
- Specifying the column and using `DEFAULT` in the values list

Table 2 `INSERT` behavior with defaults.

	<i>No Entry</i>		<i>Enter NULL</i>		<i>Enter DEFAULT</i>	
	<i>No Default</i>	<i>Default</i>	<i>No Default</i>	<i>Default</i>	<i>No Default</i>	<i>Default</i>
<code>NULL</code>	<code>NULL</code>	default	<code>NULL</code>	<code>NULL</code>	<code>NULL</code>	default
<code>NOT NULL</code>	error	default	error	error	Error	default

Declaring a default on a column that has the `IDENTITY` property doesn't make sense, and SQL Server will raise an error if you try it. The `IDENTITY` property acts as a default for the column. But the `DEFAULT` keyword cannot be used as a placeholder for an identity column in the values list of an `INSERT` statement. You can use a special form of `INSERT` statement if a table has a default value for every column (an identity column does meet this criteria) or allows `NULL`. The following statement uses the `DEFAULT VALUES` clause instead of a column list and values list:
`INSERT employee DEFAULT VALUES`

You can generate some test data by putting the `IDENTITY` property on a primary key column and declaring default values for all other columns and then repeatedly issuing an `INSERT` statement of this form within a `Transact-SQL` loop.

More About Constraints

This section offers some advice on working with constraints.

Constraint Names and System Catalog Entries

Earlier in this chapter, you learned about the cryptic-looking constraint names that SQL Server generates. Consider again the following simple CREATE TABLE statement:

```
CREATE TABLE customer
(
  cust_id    int      IDENTITY NOT NULL PRIMARY KEY,
  cust_name  varchar(30) NOT NULL
)
```

The constraint produced from this simple statement bears the nonintuitive name PK__customer__68E79C55. The advantage of explicitly naming your constraint rather than using the system-generated name is greater clarity. The constraint name is used in the error message or any constraint violation, so creating a name such as CUSTOMER_PK probably makes more sense to users than a name such as PK__customer__cust_i__0677FF3C. You should choose your own constraint names if such error messages are visible to your users. The first two characters (PK) show the constraint type—PK for PRIMARY KEY, UQ for UNIQUE, FK for FOREIGN KEY, and DF for DEFAULT. Next are two underscore characters, which are used as a separator. (You might be tempted to use one underscore to conserve characters and to avoid having to truncate as much. However, it's common to use a single underscore in a table name or a column name, both of which appear in the constraint name. Using two underscore characters distinguishes the kind of a name it is and where the separation occurs.)

Next comes the table name (customer), which is limited to 116 characters for a PRIMARY KEY constraint and slightly fewer characters for all other constraint names. For all constraints other than PRIMARY KEY, there are then two more underscore characters for separation followed by the next sequence of characters, which is the column name. The column name is truncated to five characters if necessary. If the column name has fewer than five characters in it, the length of the table name portion can be slightly longer.

And finally, the hexadecimal representation of the object ID for the constraint (68E79C55) comes after another separator. (This value is used as the id column of the sysobjects system table and the constid column of the sysconstraints system table.) Object names are limited to 128 characters in SQL Server 2000, so the total length of all portions of the constraint name must also be less than or equal to 128.

The hexadecimal value 0x68E79C55 is equal to the decimal value 1760009301, which is the value of constid in sysconstraints and of id in sysobjects.

These sample queries of system tables show the following:

A constraint is an object. A constraint has an entry in the sysobjects table in the xtype column of C, D, F, PK, or UQ for CHECK, DEFAULT, FOREIGN KEY, PRIMARY KEY, and UNIQUE, respectively.

Sysconstraints relates to sysobjects. The sysconstraints table is really just a view of the sysobjects system table. The constid column in the view is the object ID of the constraint, and the id column of sysconstraints is the object ID of the base table on which the constraint is declared. If the constraint is a column-level CHECK, FOREIGN KEY, or DEFAULT constraint, sysconstraints.colid has the column ID of the column. This colid in sysconstraints is related to the colid of syscolumns for the base table represented by id. A table-level constraint or any PRIMARY KEY/UNIQUE constraint (even if column level) always has 0 in this column.

To see the names and order of the columns in a PRIMARY KEY or UNIQUE constraint, you can query the sysindexes and syscolumns tables for the index being used to enforce the constraint. The name of the constraint and that of the index enforcing the constraint are the same, whether the name was user-specified or system-generated. The columns in the index key are somewhat cryptically encoded in the keys1 and keys2 fields of sysindexes. The easiest way to decode these values is to simply use the sp_helpindex system stored procedure; alternatively, you can use the code of that procedure as a template if you need to decode them in your own procedure.

Decoding the *status* Field

The status field of the sysconstraints view is a pseudo-bit-mask field packed with information. We could also call it a bitmap, because each bit has a particular meaning. If you know how to crack this column, you can essentially write your own sp_helpconstraint-like procedure. Note that the documentation is incomplete regarding the values of this column. One way to start decoding this column is to look at the definition of the sysconstraints view using the sp_helptext system procedure. The lowest four bits, obtained by AND'ing status with 0xF (status & 0xF), contain the constraint type. A value of 1 is PRIMARY KEY, 2 is UNIQUE, 3 is FOREIGN KEY, 4 is CHECK, and 5 is DEFAULT. The fifth bit is on (status & 0x10 <> 0) when the constraint is a nonkey constraint on a single column. The sixth bit is on (status & 0x20 <> 0) when the constraint is on multiple columns or is a PRIMARY KEY or UNIQUE constraint.

Some of the documentation classifies constraints as either table-level or column-level. This implies that any constraint defined on the line with a column is a column-level constraint and any constraint defined as a separate line in the table or added to the table with the ALTER TABLE command is a table-level constraint. However, this distinction does not hold true when you look at sysconstraints. Although it is further documented that the fifth bit is for a column-level constraint, you can see for yourself that this bit is on for any single column constraint except PRIMARY KEY and UNIQUE and that the sixth bit, which is documented as indicating a table-level constraint, is on for all multi-column constraints, as well as PRIMARY KEY and UNIQUE constraints. Some of the higher bits are used for internal status purposes, such as noting whether a nonclustered index is being rebuilt, and for other internal states. Table 6-11 shows some of the other bit-mask values you might be interested in:

Table 3. Bitmap values in the status column of sysconstraints.

<u>Bitmap Value</u>	<u>Description</u>
16	The constraint is a "column-level" constraint, which means that it's a single column constraint and isn't enforcing entity integrity
32	The constraint is a "table-level" constraint, which means that it's either a multi-column constraint, a PRIMARY KEY, or a UNIQUE constraint
512	The constraint is enforced by a clustered index.
1024	The constraint is enforced by a nonclustered index.
16384	The constraint has been disabled.
32767	The constraint has been enabled.
131072	SQL Server has generated a name for the constraint

Using this information, and not worrying about the higher bits used for internal status, you could use the following query to show constraint information for the employee table:

```
SELECT
OBJECT_NAME(constid) 'Constraint Name',
constid 'Constraint ID',
CASE (status & 0xF)
  WHEN 1 THEN 'Primary Key'
  WHEN 2 THEN 'Unique'
  WHEN 3 THEN 'Foreign Key'
  WHEN 4 THEN 'Check'
  WHEN 5 THEN 'Default'
  ELSE 'Undefined'
END 'Constraint Type',
CASE (status & 0x30)
  WHEN 0x10 THEN 'Column'
  WHEN 0x20 THEN 'Table'
  ELSE 'NA'
END 'Level'
FROM sysconstraints
WHERE id=OBJECT_ID('employee')
```

>>>

RDBMS Concepts and MS-SQL Server 2000
SQL Server Data Types

Constraint Name	ID	Constraint Type	Constraint Level
PK_employee__49C3F6B7 Primary Key	1237579447		Table
DF_employee__emp_id__4AB81AF0	1253579504	Default	Column
CK_employee__emp_id__4BAC3F29	1269579561	Check	Column
DF_employee__emp_name__4CA06362 1285579618		Default	Column
no_nums	1301579675	Check	Column
DF_employee__mgr_id__4E88ABD4	1317579732	Default	Column
FK_employee__mgr_id__4F7CD00D Foreign Key	1333579789		Column
CK_employee__entered_date__5070F446 1349579846		Check	Column
def_today	1365579903	Default	Column
DF_employee__entered_by__52593CB8	1381579960	Default	Column
CK_employee__entered_by__534D60F1	1397580017	Check	Column
valid_entered_by	1413580074	Check	Table
valid_mgr	1429580131	Check	Table
end_of_month	1445580188	Check	Table

Constraint Failures in Transactions and Multiple-Row Data Modifications

Many bugs occur in application code because the developers don't understand how failure of a constraint affects a multiple-statement transaction declared by the user. The biggest misconception is that any error, such as a constraint failure, automatically aborts and rolls back the entire transaction. On the contrary, after an error is raised, it's up to the transaction to either proceed and ultimately commit or to roll back. This feature provides the developer with the flexibility to decide how to handle errors. (The semantics are also in accordance with the ANSI SQL-92 standard for COMMIT behavior).

A simple transaction that tries to insert three rows of data. The second row contains a duplicate key and violates the PRIMARY KEY constraint. Some developers believe that this example wouldn't insert any rows because of the error that occurs in one of the statements; they think that this error will cause the entire transaction to be aborted. However, this doesn't happen—instead, the statement inserts two rows and then commits that change. Although the second INSERT fails, the third INSERT is processed because no error checking has occurred between the statements, and then the transaction does a COMMIT. Because no instructions were provided to take some other action after the error other than to proceed, SQL Server does just that. It adds the first and third INSERT statements to the table and ignores the second statement.

```
IF EXISTS (SELECT * FROM sysobjects WHERE name='show_error' AND
           type='U')
  DROP TABLE show_error
GO
```

```
CREATE TABLE show_error
(
  col1  smallint NOT NULL PRIMARY KEY,
  col2  smallint NOT NULL
)
GO
```

```
BEGIN TRANSACTION
```

```
INSERT show_error VALUES (1, 1)
INSERT show_error VALUES (1, 2)
INSERT show_error VALUES (2, 2)
```

```
COMMIT TRANSACTION
GO
```

```
SELECT * FROM show_error
GO
```

```
Server: Msg 2627, Level 14, State 1, Line 1
Violation of PRIMARY KEY constraint 'PK__show_error__6EF57B66'.
Cannot insert duplicate key in object 'show_error'.
```

The statement has been terminated.

```
col1  col2
----  ----
1      1
```

2 2

Here's a modified version of the transaction. This example does some simple error checking using the system function @@ERROR and rolls back the transaction if any statement results in an error. In this example, no rows are inserted because the transaction is rolled back.

```
IF EXISTS (SELECT * FROM sysobjects WHERE name='show_error'
  AND type='U')
  DROP TABLE show_error
GO
```

```
CREATE TABLE show_error
(
col1  smallint NOT NULL PRIMARY KEY,
col2  smallint NOT NULL
)
GO
```

```
BEGIN TRANSACTION
INSERT show_error VALUES (1, 1)
IF @@ERROR <> 0 GOTO TRAN_ABORT
INSERT show_error VALUES (1, 2)
if @@ERROR <> 0 GOTO TRAN_ABORT
INSERT show_error VALUES (2, 2)
if @@ERROR <> 0 GOTO TRAN_ABORT
COMMIT TRANSACTION
GOTO FINISH
```

```
TRAN_ABORT:
ROLLBACK TRANSACTION
```

```
FINISH:
```

```
GO
SELECT * FROM show_error
GO
```

```
Server: Msg 2627, Level 14, State 1, Line 1
Violation of PRIMARY KEY constraint 'PK__show_error__70DDC3D8':
  Cannot insert duplicate key in object 'show_error'.
```

The statement has been terminated.

```
col1  col2
----  ----
(0 row(s) affected)
```

Because many developers have handled transaction errors incorrectly and because it can be tedious to add an error check after every command, SQL Server includes a SET statement that aborts a transaction if it encounters any error during the transaction. (Transact-SQL has no WHENEVER statement, although such a feature would be useful for situations like this.) Using SET XACT_ABORT ON causes the entire transaction to be aborted and rolled back if any error is encountered. The default setting is OFF, which is consistent with ANSI-standard behavior. By setting the option XACT_ABORT ON, we can now rerun the example that does no error checking, and no rows will be inserted:

```
IF EXISTS (SELECT * FROM sysobjects WHERE name='show_error'
```

```
    AND type='U')
    DROP TABLE show_error
GO

CREATE TABLE show_error
(
col1  smallint NOT NULL PRIMARY KEY,
col2  smallint NOT NULL
)
GO

SET XACT_ABORT ON
BEGIN TRANSACTION

INSERT show_error VALUES (1, 1)
INSERT show_error VALUES (1, 2)
INSERT show_error VALUES (2, 2)

COMMIT TRANSACTION
GO

SELECT * FROM show_error
GO
```

```
Server: Msg 2627, Level 14, State 1, Line 1
Violation of PRIMARY KEY constraint 'PK__show_error__72C60C4A'.
Cannot insert duplicate key in object 'show_error'.
```

```
col1  col2
----  ----
```

```
(0 row(s) affected)
```

A final comment about constraint errors and transactions: a single data modification statement (such as an UPDATE statement) that affects multiple rows is automatically an atomic operation, even if it's not part of an explicit transaction. If such an UPDATE statement finds 100 rows that meet the criteria of the WHERE clause but one row fails because of a constraint violation, no rows will be updated.

The modification of a given row will fail if any constraint is violated or if a trigger aborts the operation. As soon as a failure in a constraint occurs, the operation is aborted, subsequent checks for that row aren't performed, and no trigger fires for the row. Hence, the order of these checks can be important, as the following list shows

1. Defaults are applied as appropriate.
2. NOT NULL violations are raised.
3. CHECK constraints are evaluated.
4. FOREIGN KEY checks of *referencing* tables are applied.
5. FOREIGN KEY checks of *referenced* tables are applied.
6. UNIQUE/PRIMARY KEY is checked for correctness.
7. Triggers

Joins

You gain much more power when you join tables, which typically results in combining columns of matching rows to project and return a *virtual table*. Usually, joins are based on the primary and foreign keys of the tables involved, although the tables aren't required to explicitly declare keys. The *pubs* database contains a table of authors (*authors*) and a table of book titles (*titles*). An obvious query would be, "Show me the titles that each author has written and sort the results alphabetically by author. I'm interested only in authors who live outside California." Neither the *authors* table nor the *titles* table alone has all this information. Furthermore, a many-to-many relationship exists between authors and titles; an author might have written several books, and a book might have been written by multiple authors. So an intermediate table, *titleauthor*, exists expressly to associate authors and titles, and this table is necessary to correctly join the information from *authors* and *titles*. To join these tables, you must include all three tables in the FROM clause of the SELECT statement, specifying that the columns that make up the keys have the same values:

```
SELECT
'Author'=RTRIM(au_lname) + ', ' + au_fname,
'Title'=title
FROM authors AS A JOIN titleauthor AS TA
  ON A.au_id=TA.au_id      -- JOIN CONDITION
JOIN titles AS T
  ON T.title_id=TA.title_id -- JOIN CONDITION
WHERE A.state <> 'CA'
ORDER BY 1
```

Here's the output:

Author	Title
Blotchet-Halls, Reginald	Fifty Years in Buckingham Palace Kitchens
DeFrance, Michel	The Gourmet Microwave
del Castillo, Innes	Silicon Valley Gastronomic Treats
Panteley, Sylvia	Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean
Ringer, Albert	Is Anger the Enemy?
Ringer, Albert	Life Without Fear
Ringer, Anne	The Gourmet Microwave
Ringer, Anne	Is Anger the Enemy?

Before discussing join operations further, let's study the preceding example. The author's last and first names have been concatenated into one field. The RTRIM (right trim) function is used to strip off any trailing whitespace from the *au_lname* column. Then we add a comma and a space and concatenate on the *au_fname* column. This column is then *aliased* as simply *Author* and is returned to the calling application as a single column

The RTRIM function isn't needed for this example. Because the column is of type *varchar*, trailing blanks won't be present. RTRIM is shown for illustration purposes only

Another important point is that the ORDER BY 1 clause indicates that the results should be sorted by the first column in the result set. It's more typical to use the column name rather than its number, but using the column number provides a convenient shorthand when the column is an expression and hence isn't present in the base table or view (virtual table) being queried.

Instead of using ORDER BY 1, you can repeat the same expression used in the select list and specify `ORDER BY RTRIM(au_lname) + ', ' + au_fname` instead. Alternatively, SQL server

provides a feature supported by the ANSI SQL-99 standard that allows sorting by columns not included in the select list. So even though you don't individually select the columns `au_lname` or `au_fname`, you can nonetheless choose to order the query based on these columns by specifying columns `ORDER BY au_lname, au_fname`. We'll see this in the next example. Notice also that the query contains comments (`-- JOIN CONDITION`). A double hyphen (`--`) signifies that the rest of the line is a comment (similar to `//` in C++). You can also use the C-style `/* comment block */`, which allows blocks of comment lines.

Comments can be nested, but you should generally try to avoid this. You can easily introduce a bug by not realizing that a comment is nested within another comment and misreading the code.

Now let's examine the join in the example above. The ON clauses specify how the tables relate and set the join criteria, stating that `au_id` in `authors` must equal `au_id` in `titleauthor`, and `title_id` in `titles` must equal `title_id` in `titleauthor`. This type of join is referred to as an *equijoin*, and it's the most common type of join operation. To remove ambiguity, you must qualify the columns. You can do this by specifying the columns in the form `table.column`, as in `authors.au_id = titleauthor.au_id`. The more compact and common way to do this, however, is by specifying a *table alias* in the FROM clause, as was done in this example. By following the `titles` table with the word `AS` and the letter `T`, the `titles` table will be referred to as `T` anywhere else in the query where the table is referenced. Typically, such an alias consists of one or two letters, although it can be much longer (following the same rules as identifiers).

After a table is aliased, it must be referred to by the alias, so now we can't refer to `authors.au_id` because the `authors` table has been aliased as `A`. We must use `A.au_id`. Note also that the `state` column of `authors` is referred to as `A.state`. The other two tables don't contain a `state` column, so qualifying it with the `A.` prefix isn't necessary; however, doing so makes the query more readable—and less prone to subsequent bugs.

The join is accomplished using the ANSI JOIN SQL syntax, which was introduced in SQL Server version 6.5. Many examples and applications continue to use an old-style JOIN syntax, which is shown below. (The term "old-style JOIN" is actually used by the SQL-92 specification.) The ANSI JOIN syntax is based on ANSI SQL-92. The main differences between the two types of join formulations are

- The ANSI JOIN actually uses the keyword JOIN.
- The ANSI JOIN segregates join conditions from search conditions.

The ANSI JOIN syntax specifies the JOIN conditions in the ON clauses (one for each pair of tables), and the search conditions are specified in the WHERE clause—for example, `WHERE state <> 'CA'`. Although slightly more verbose, the explicit JOIN syntax is more readable. There's no difference in performance; behind the scenes, the operations are the same. Here's how you can respecify the query using the old-style JOIN syntax:

```
SELECT
```

```
'Author'=RTRIM(au_lname) + ', ' + au_fname,  
'Title'=title  
FROM authors A, titles T, titleauthor TA  
WHERE  
A.au_id=TA.au_id AND T.title_id=TA.title_id -- JOIN CONDITIONS  
AND A.state <> 'CA'  
ORDER BY 1
```

This query produces the same output and the same execution plan as the previous query.

One of the most common errors that new SQL users make when using the old-style JOIN syntax is not specifying the join condition. Omitting the WHERE clause is still a valid SQL request and causes a result set to be returned. However, that result set is likely not what the user wanted. In the query above, omitting the WHERE clause would return the Cartesian product of the three tables: it would generate every possible combination of rows between them. Although in a few unusual cases you might want all permutations, usually this is just a user error. The number of rows returned can be huge and typically doesn't represent anything meaningful.

The Cartesian product of the three small tables here (*authors*, *titles*, and *titleauthor* each have less than 26 rows) generates 10,350 rows of (probably) meaningless output.

Using the ANSI JOIN syntax, it's impossible to accidentally return a Cartesian product of the tables—and that's one reason to use ANSI JOINS almost exclusively. The ANSI JOIN syntax requires an ON clause for specifying how the tables are related. In cases where you actually do want a Cartesian product, the ANSI JOIN syntax allows you to use a CROSS JOIN operator, which we'll examine in more detail later.

You might want to try translating some of them into queries using old-style JOIN syntax, because you should be able to recognize both forms. If you have to read SQL code written earlier than version 7, you're bound to come across queries using this older syntax.

The most common form of join is an *equijoin*, which means that the condition linking the two tables is based on equality. An equijoin is sometimes referred to as an *inner join* to differentiate it from an *outer join*, which I'll discuss shortly. Strictly speaking, an inner join isn't quite the same as an equijoin; an inner join can use an operator such as less than (<) or greater than (>). So all equijoins are inner joins but not all inner joins are equijoins. To make this distinction clear in your code, you can use the INNER JOIN syntax in place of JOIN

```
FROM authors AS A INNER JOIN titleauthor TA ON A.au_id=TA.au_id
```

Other than making the syntax more explicit, there's no difference in the semantics or the execution. By convention, the modifier INNER generally isn't used.

ANSI SQL-92 also specifies the natural join operation, in which you don't have to specify the tables' column names. By specifying syntax such as *FROM authors NATURAL JOIN titleauthor*,

the system automatically knows how to join the tables without your specifying the column names to match. SQL Server doesn't yet support this feature.

The ANSI specification calls for the natural join to be resolved based on identical column names between the tables. Perhaps a better way to do this would be based on a declared primary key-foreign key relationship, *if it exists*. Admittedly, declared key relationships have issues, too, because there's no restriction that only one such foreign key relationship be set up. Also, if the natural join were limited to only such relationships, all joins would have to be known in advance—as in the old CODASYL days.

The FROM clause in this example shows an alternative way to specify a table alias—by omitting the AS. The use of AS preceding the table alias, as used in the previous example, conforms to ANSI SQL-92. From SQL Server's standpoint, the two methods are equivalent (stating *FROM authors AS A* is identical to stating *FROM authors A*). Commonly, the AS formulation is used in ANSI SQL-92 join operations and the formulation that omits AS is used with the old-style join formulation. However, you can use either formulation—it's strictly a matter of preference.

Outer Joins

Inner joins return only rows from the respective tables that meet the conditions specified in the ON clause. In other words, if a row in the first table doesn't match any rows in the second table, that row isn't returned in the result set. In contrast, outer joins preserve some or all of the unmatched rows. To illustrate how easily subtle semantic errors can be introduced, let's refer back to the previous two query examples, in which we want to see the titles written by all authors not living in California. The result omits two writers who do not, in fact, live in California. Are the queries wrong? No! They perform exactly as written—we didn't specify that authors who currently have no titles in the database should be included. The results of the query are as we requested. In fact, the *authors* table has four rows that have no related row in the *titleauthor* table, which means these "authors" actually haven't written any books. Two of these authors don't contain the value CA in the *state* column, as the following result set shows.

au_id	Author	state
341-22-1782	Smith, Meander	KS
527-72-3246	Greene, Morningstar	TN
724-08-9931	Stringer, Dirk	CA
893-72-1158	McBaden, Heather	CA

The *titles* table has one row for which there is no author in our database, as shown here. (Later, you'll see the types of queries used to produce these two result sets.)

title_id	title
MC3026	The Psychology of Computer Cooking

To find out authors who live outside of California," the query would use an outer join so that authors with no matching titles would be selected:

```
SELECT
'Author'=RTRIM(au_lname) + ', ' + au_fname,
'Title'=title
FROM
(
-- JOIN CONDITIONS
-- FIRST join authors and titleauthor
(authors AS A
FULL OUTER JOIN titleauthor AS TA ON A.au_id=TA.au_id
)
-- The result of the previous join is then joined to titles
FULL OUTER JOIN titles AS T ON TA.title_id=T.title_id
)
WHERE
state <> 'CA' OR state IS NULL
ORDER BY 1
```

Here's the output:

Here's the output:

Author	Title
NULL	The Psychology of Computer Cooking
Blotchet-Halls, Reginald	Fifty Years in Buckingham Palace

DeFrance, Michel	Kitchens
del Castillo, Innes	The Gourmet Microwave
Greene, Morningstar	Silicon Valley Gastronomic Treats
Panteley, Sylvia	NULL
	Onions, Leeks, and Garlic: Cooking
	Secrets of the Mediterranean
Ringer, Albert	Is Anger the Enemy?
Ringer, Albert	Life Without Fear
Ringer, Anne	The Gourmet Microwave
Ringer, Anne	Is Anger the Enemy?
Smith, Meander	NULL

The query demonstrates a *full outer join*. Rows in the *authors* and *titles* tables that don't have a corresponding entry in *titleauthor* are still presented, but with a NULL entry for the *title* or *author* column. (The data from the *authors* table is requested as a comma between the last names and the first names. Because we have the option `CONCAT_NULL_YIELDS_NULL` set to ON, the result is NULL for the *author* column. If we didn't have that option set to ON, SQL Server would have returned the single comma between the nonexistent last and first names.) A full outer join preserves nonmatching rows from both the lefthand and righthand tables. In the example above, the *authors* table is presented first, so it is the lefthand table when joining to *titleauthor*. The result of that join is the lefthand table when joining to *titles*.

ANSI Outer Joins

The outer join formulations here are standard ANSI SQL-92. However, this is one area in which there aren't enough good examples. Using OUTER JOIN with more than two tables introduces some obscure issues; in fact, few products provide support for full outer joins.

You can generate missing rows from one or more of the tables by using either a *left outer join* or a *right outer join*. So if we want to preserve all authors and generate a row for all authors who have a missing title, but we don't want to preserve titles that have no author, we can reformulate the query using LEFT OUTER JOIN, as shown below. This join preserves entries only on the lefthand side of the join. Note that the left outer join of the *authors* and *titleauthor* columns generates two such rows (for Greene and Smith). The result of the join is the lefthand side of the join to *titles*; therefore, the LEFT OUTER JOIN must be specified again to preserve these rows with no matching *titles* rows.

```
SELECT
'Author'=RTRIM(au_lname) + ', ' + au_fname,
'Title'=title
FROM
(
(authors as A
LEFT OUTER JOIN titleauthor AS TA ON A.au_id=TA.au_id
)
LEFT OUTER JOIN titles AS T ON TA.title_id=T.title_id
)
```

Here's the output:

Author	Title
-----	-----
Blotchet-Halls, Reginald	Fifty Years in Buckingham Palace

	Kitchens
DeFrance, Michel	The Gourmet Microwave
del Castillo, Innes	Silicon Valley Gastronomic Treats
Greene, Morningstar	NULL
Panteley, Sylvia	Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean
Ringer, Albert	Is Anger the Enemy?
Ringer, Albert	Life Without Fear
Ringer, Anne	The Gourmet Microwave
Ringer, Anne	Is Anger the Enemy?
Smith, Meander	NULL

The query produces the same rows as the full outer join, except for the row for *The Psychology of Computer Cooking*. Because we specified only LEFT OUTER JOIN, there was no request to preserve *titles* (righthand) rows with no matching rows in the result of the join of *authors* and *titleauthor*.

You must use care with OUTER JOIN operations because the order in which tables are joined affects which rows are preserved and which aren't. In an inner join, the symmetric property holds (if A equals B, then B equals A) and no difference results, whether something is on the left or the right side of the equation, and no difference results in the order in which joins are specified. This is definitely *not* the case for OUTER JOIN operations.

Consider the following two queries and their results:

QUERY 1

```
SELECT
'Author'= RTRIM(au_lname) + ', ' + au_fname,
'Title'=title
FROM (titleauthor AS TA
RIGHT OUTER JOIN authors AS A ON (A.au_id=TA.au_id))

FULL OUTER JOIN titles AS T ON (TA.title_id=T.title_id)
WHERE
A.state <> 'CA' or A.state is NULL
ORDER BY 1
```

Author	Title
-----	-----
NULL	The Psychology of Computer Cooking
Blotchet-Halls, Reginald	Fifty Years in Buckingham Palace Kitchens
DeFrance, Michel	The Gourmet Microwave
del Castillo, Innes	Silicon Valley Gastronomic Treats
Greene, Morningstar	NULL
Panteley, Sylvia	Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean
Ringer, Albert	Is Anger the Enemy?
Ringer, Albert	Life Without Fear
Ringer, Anne	The Gourmet Microwave
Ringer, Anne	Is Anger the Enemy?
Smith, Meander	NULL

This query produces results semantically equivalent to the previous FULL OUTER JOIN formulation, although we've switched the order of the *authors* and *titleauthor* tables. This query

and the previous one preserve both authors with no matching titles and titles with no matching authors. This might not be obvious because RIGHT OUTER JOIN is clearly different than FULL OUTER JOIN. However, in this case we know it's true because a FOREIGN KEY constraint exists on the *titleauthor* table to ensure that there can never be a row in the *titleauthor* table that doesn't match a row in the *authors* table, and the FOREIGN KEY columns in *titleauthor* are defined to not allow NULL values. So we can be confident that the *titleauthor* RIGHT OUTER JOIN to *authors* can't produce any fewer rows than would a FULL OUTER JOIN.

But if we modify the query ever so slightly by changing the join order again, look what happens:

QUERY 2

```
SELECT
'Author'=rtrim(au_lname) + ', ' + au_fname,
'Title'=title
FROM (titleauthor AS TA
FULL OUTER JOIN titles AS T ON TA.title_id=T.title_id)
RIGHT OUTER JOIN authors AS A ON A.au_id=TA.au_id
WHERE
A.state <> 'CA' or A.state is NULL
ORDER BY 1
```

Author	Title
Blotchet-Halls, Reginald	Fifty Years in Buckingham Palace Kitchens
DeFrance, Michel	The Gourmet Microwave
del Castillo, Innes	Silicon Valley Gastronomic Treats
Greene, Morningstar	NULL
Panteley, Sylvia	Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean
Ringer, Albert	Is Anger the Enemy?
Ringer, Albert	Life Without Fear
Ringer, Anne	The Gourmet Microwave
Ringer, Anne	Is Anger the Enemy?
Smith, Meander	NULL

At a glance, Query 2 looks equivalent to Query 1, although the join order is slightly different. But notice how the results differ. Query 2 didn't achieve the goal of preserving the *titles* rows without corresponding *authors*, and the row for *The Psychology of Computer Cooking* is again excluded. This row would have been preserved in the first join operation:

```
FULL OUTER JOIN titles AS T ON TA.title_id=T.title_id
```

But then the row is discarded because the second join operation preserves only authors without matching titles:

```
RIGHT OUTER JOIN authors AS A ON A.au_id=TA.au_id
```

Because the title row for *The Psychology of Computer Cooking* is on the lefthand side of this join operation and only a RIGHT OUTER JOIN operation is specified, this title is discarded.

Just as INNER is an optional modifier, so is OUTER. Hence, LEFT OUTER JOIN can be equivalently specified as LEFT JOIN, and FULL OUTER JOIN can be equivalently expressed as

FULL JOIN. However, although INNER is seldom used by convention and is usually only implied, OUTER is almost always used by convention when specifying any type of outer join.

Because join order matters, use the parentheses and indentation carefully when specifying OUTER JOIN operations. Indentation, of course, is always optional, and use of parentheses is often optional. But as this example shows, it's easy to make mistakes that result in your queries returning the wrong answers. As is true with almost all programming, simply getting into the habit of using comments, parentheses, and indentation often results in such bugs being noticed and fixed by a developer or database administrator before they make their way into your applications.

The Obsolete *= OUTER JOIN Operator

Prior to version 6.5, SQL Server had limited outer-join support in the form of the special operators *= and =*. Many people assume that the LEFT OUTER JOIN syntax is simply a synonym for *=, but this isn't the case. LEFT OUTER JOIN is semantically different from and superior to *=.

For inner joins, the symmetric property holds, so the issues with old-style JOIN syntax don't exist. You can use either new-style or old-style syntax with inner joins. For outer joins, you should consider the *= operator obsolete and move to the OUTER JOIN syntax as quickly as possible—the *= operator might be dropped entirely in future releases of SQL Server.

ANSI's OUTER JOIN syntax, which was adopted in SQL Server version 6.5, recognizes that for outer joins, the conditions of the join must be evaluated separately from the criteria applied to the rows that are joined. ANSI gets it right by separating the JOIN criteria from the WHERE criteria. The old SQL Server *= and =* operators are prone to ambiguities, especially when three or more tables, views, or subqueries are involved. Often the results aren't what you'd expect, even though you might be able to explain them. But sometimes you simply can't get the result you want. These aren't implementation bugs; more accurately, these are inherent limitations in trying to apply the outer-join criteria in the WHERE clause.

Who's Writing the Queries?

If you expect end users to understand the semantics of outer joins or deal correctly with NULL, you might as well just give them a random data generator. These concepts are tricky, and it's your job to insulate your users from them. Later in this chapter, you'll see how you can use views to accomplish this.

When *= was introduced, no ANSI specification existed for OUTER JOIN or even for INNER JOIN. Just the old-style join existed, with operators such as =* in the WHERE clause. So the designers quite reasonably tried to fit an outer-join operator into the WHERE clause, which was the only place where joins were ever stated. However, efforts to resolve this situation helped spur the ANSI SQL committee's specification of proper OUTER JOIN syntax and semantics. Implementing outer joins correctly is difficult, and SQL Server is one of the few mainstream products that has done so.

To illustrate the semantic differences and problems with the old *= syntax, a series of examples will be discussed using both new and old outer-join syntax. The following is essentially the same outer-join query shown earlier, but this one returns only a count. It correctly finds the 11 rows, preserving both authors with no titles and titles with no authors.

```
-- New style OUTER JOIN correctly finds 11 rows
SELECT COUNT(*)
FROM
```

```
(
-- FIRST join authors and titleauthor
  (authors AS A
  LEFT OUTER JOIN titleauthor AS TA ON A.au_id=TA.au_id
  )
-- The result of the previous join is then joined to titles
  FULL OUTER JOIN titles AS T ON TA.title_id=T.title_id
)
WHERE
state <> 'CA' OR state IS NULL
```

There's really no way to write this query—which does a full outer join—using the old syntax, because the old syntax simply isn't expressive enough. Here's what looks to be a reasonable try—but it generates several rows that you wouldn't expect:

```
-- Attempt with old-style join. Really no way to do FULL OUTER
-- JOIN. This query finds 144 rows--WRONG!!
SELECT COUNT(*)
FROM
  authors A, titleauthor TA, titles T
WHERE
A.au_id *= TA.au_id
AND
TA.title_id =* T.title_id
AND
(state <> 'CA' OR state IS NULL)
```

Now let's examine some issues with the old-style outer join using a simple example of only two tables, *Customers* and *Orders*:

```
CREATE TABLE Customers
(
  Cust_ID int PRIMARY KEY,
  Cust_Name char(20)
)

CREATE TABLE Orders
(
  OrderID int PRIMARY KEY,
  Cust_ID int REFERENCES Customers(Cust_ID)
)
GO

INSERT Customers VALUES (1, 'Cust 1')
INSERT Customers VALUES (2, 'Cust 2')
INSERT Customers VALUES (3, 'Cust 3')
INSERT Orders VALUES (10001, 1)
INSERT Orders VALUES (20001, 2)
GO
```

At a glance, in the simplest case, the new-style and old-style syntax appear to work the same. Here's the new syntax:


```
SELECT
'Customers.Cust_ID'=Customers.Cust_ID, Customers.Cust_Name,
'Orders.Cust_ID'=Orders.Cust_ID
FROM Customers LEFT JOIN Orders
ON Customers.Cust_ID=Orders.Cust_ID
```

Here's the output:

Customers.Cust_ID	Cust_Name	Orders.Cust_ID
1	Cust 1	1
2	Cust 2	2
3	Cust 3	NULL

And here's the old-style syntax:

```
SELECT 'Customers.Cust_ID'=Customers.Cust_ID, Customers.Cust_Name,
'Orders.Cust_ID'=Orders.Cust_ID
FROM Customers, Orders WHERE Customers.Cust_ID *= Orders.Cust_ID
```

Here's the output:

Customers.Cust_ID	Cust_Name	Orders.Cust_ID
1	Cust 1	1
2	Cust 2	2
3	Cust 3	NULL

But as soon as you begin to add restrictions, things get tricky. What if you want to filter out *Cust 2*? With the new syntax it's easy, but remember not to filter out the row with NULL that the outer join just preserved!

```
SELECT 'Customers.Cust_ID'=Customers.Cust_ID, Customers.Cust_Name,
'Orders.Cust_ID'=Orders.Cust_ID
FROM Customers LEFT JOIN Orders
ON Customers.Cust_ID=Orders.Cust_ID
WHERE Orders.Cust_ID <> 2 OR Orders.Cust_ID IS NULL
```

Here's the output:

Customers.Cust_ID	Cust_Name	Orders.Cust_ID
1	Cust 1	1
3	Cust 3	NULL

Now try to do this query using the old-style syntax and filter out *Cust 2*:

```
SELECT 'Customers.Cust_ID'=Customers.Cust_ID, Customers.Cust_Name,
'Orders.Cust_ID'=Orders.Cust_ID
FROM Customers, Orders
WHERE Customers.Cust_ID *= Orders.Cust_ID
AND (Orders.Cust_ID <> 2 OR Orders.Cust_ID IS NULL)
```

Here's the output:

Customers.Cust_ID	Cust_Name	Orders.Cust_ID
1	Cust 1	1
2	Cust 2	NULL
3	Cust 3	NULL

Notice that this time, we don't get rid of *Cust 2*. The check for NULL occurs before the JOIN, so the outer-join operation puts *Cust 2* back. This result might be less than intuitive, but at least we can explain and defend it. That's not always the case, as you'll see in a moment.

If you look at the preceding query, you might think that we should have filtered out *Customers.Cust_ID* rather than *Orders.Cust_ID*. How did we miss that? Surely this query will fix the problem:

```
SELECT 'Customers.Cust_ID'=Customers.Cust_ID, Customers.Cust_Name,
'Orders.Cust_ID'=Orders.Cust_ID
FROM Customers, Orders
WHERE Customers.Cust_ID *= Orders.Cust_ID
AND (Customers.Cust_ID <> 2 OR Orders.Cust_ID IS NULL)
```

Here's the output:

Customers.Cust_ID	Cust_Name	Orders.Cust_ID
1	Cust 1	1
2	Cust 2	NULL
3	Cust 3	NULL

Oops! Same result. The problem here is that *Orders.Cust_ID IS NULL* is now being applied *after* the outer join, so the row is presented again. If we're careful and understand exactly how the old outer join is processed, we can get the results we want with the old-style syntax for this query. We need to understand that the *OR Orders.Cust_ID IS NULL* puts back *Cust_ID 2*, so just take that out. Here is the code:

```
SELECT 'Customers.Cust_ID'=Customers.Cust_ID, Customers.Cust_Name,
'Orders.Cust_ID'=Orders.Cust_ID
FROM Customers, Orders
WHERE Customers.Cust_ID *= Orders.Cust_ID
AND Customers.Cust_ID <> 2
```

Here's the output:

Customers.Cust_ID	Cust_Name	Orders.Cust_ID
1	Cust 1	1
3	Cust 3	NULL

Finally! This is the result we want. And if you really think about it, the semantics are even understandable (although different from the new style). Besides the issues of joins with more than two tables and the lack of a full outer join, we also can't effectively deal with subqueries and views (virtual tables). For example, let's try creating a view with the old-style outer join:

```
CREATE VIEW Cust_old_OJ AS
(SELECT Orders.Cust_ID, Customers.Cust_Name
```

```
FROM Customers, Orders
WHERE Customers.Cust_ID *= Orders.Cust_ID)
```

A simple select from the view looks fine:

```
SELECT * FROM Cust_old_OJ
```

And it gives this output:

Cust_ID	Cust_Name
1	Cust 1
2	Cust 2
NULL	Cust 3

But restricting from this view doesn't seem to make sense:

```
SELECT * FROM Cust_old_OJ WHERE Cust_ID <> 2
AND Cust_ID IS NOT NULL
```

The output shows NULLs in the *Cust_ID* column, even though we tried to filter them out:

Cust_ID	Cust_Name
1	Cust 1
NULL	Cust 2
NULL	Cust 3

If we expand the view to the full select and we realize that *Cust_ID* is *Orders.Cust_ID*, not *Customers.Cust_ID*, perhaps we can understand why this happened. But we still can't filter out those rows! In contrast, if we create the view with the new syntax and correct semantics, it works exactly as expected:

```
CREATE VIEW Cust_new_OJ AS
(SELECT Orders.Cust_ID, Customers.Cust_Name
FROM Customers LEFT JOIN Orders
ON Customers.Cust_ID=Orders.Cust_ID )
GO
```

```
SELECT * FROM Cust_new_OJ WHERE Cust_ID <> 2 AND Cust_ID IS NOT NULL
```

Here's what was expected:

Cust_ID	Cust_Name
1	Cust 1

The new syntax performed the outer join and then applied the restrictions in the WHERE clause to the result. In contrast, the old style applied the WHERE clause to the tables being joined and then performed the outer join, which can reintroduce NULL rows. This is why the results often seemed bizarre. However, if that behavior is what you want, you could apply the criteria in the JOIN clause instead of in the WHERE clause.

The following example uses the new syntax to mimic the old behavior. The WHERE clause is shown here simply as a placeholder to make clear that the statement *Cust_ID <> 2* is in the JOIN section, not in the WHERE section.

```
SELECT 'Customers.Cust_ID'=Customers.Cust_ID, Customers.Cust_Name,  
       'Orders.Cust_ID'=Orders.Cust_ID  
FROM Customers LEFT JOIN Orders  
     ON Customers.Cust_ID=Orders.Cust_ID  
     AND Orders.Cust_ID <> 2  
WHERE 1=1
```

Here's the output:

Customers.Cust_ID	Cust_Name	Orders.Cust_ID
1	Cust 1	1
2	Cust 2	NULL
3	Cust 3	NULL

As you can see, the row for *Cust 2* was filtered out from the *Orders* table before the join, but because it was NULL, it was reintroduced by the OUTER JOIN operation.

With the improvements in outer-join support, you can now use outer joins where you couldn't previously. A bit later, you'll see how to use an outer join instead of a correlated subquery in a common type of query.

Cross Joins

In addition to INNER JOIN, OUTER JOIN, and FULL JOIN, the ANSI JOIN syntax allows a CROSS JOIN. Earlier, you saw that the advantage of using the ANSI JOIN syntax was that you wouldn't *accidentally* create a Cartesian product. However, in some cases, creating a Cartesian product might be exactly what you want to do. SQL Server allows you to specify a CROSS JOIN with no ON clause to produce a Cartesian product.

For example, one use for CROSS JOINS is to generate sample or test data. For example, to generate 10,000 names for a sample *employees* table, you don't have to come up with 10,000 individual INSERT statements. All you need to do is build a *first_names* table and a *last_names* table with 26 names each (perhaps one starting with each letter of the English alphabet), and a *middle_initials* table with the 26 letters. When these three small tables are joined using the CROSS JOIN operator, the result is well over 10,000 unique names to insert into the *employees* table. The SELECT statement used to generate these names looks like this:

```
SELECT first_name, middle_initial, last_name  
FROM first_names CROSS JOIN middle_initials  
     CROSS JOIN last_names
```

To summarize the five types of ANSI JOIN operations, consider two tables, *TableA* and *TableB*:

INNER JOIN (default)

TableA INNER JOIN TableB ON join_condition

The INNER JOIN returns rows from either table only if they have a corresponding row in the other table. In other words, the INNER JOIN disregards any rows in which the specific join condition, as specified in the ON clause, isn't met.

LEFT OUTER JOIN

TableA LEFT OUTER JOIN TableB ON join_condition

The LEFT OUTER JOIN returns all rows for which a connection exists between *TableA* and *TableB*; in addition, it returns all rows from *TableA* for which no corresponding row exists in *TableB*. In other words, it preserves unmatched rows from *TableA*. *TableA* is sometimes called the *preserved table*. In result rows containing unmatched rows from *TableA*, any columns selected from *TableB* are returned as NULL.

RIGHT OUTER JOIN

TableA RIGHT OUTER JOIN TableB ON join_condition

The RIGHT OUTER JOIN returns all rows for which a connection exists between *TableA* and *TableB*; in addition, it returns all rows from *TableB* for which no corresponding row exists in *TableA*. In other words, it preserves unmatched rows from *TableB*, and in this case *TableB* is the preserved table. In result rows containing unmatched rows from *TableB*, any columns selected from *TableA* are returned as NULL.

FULL OUTER JOIN

TableA FULL OUTER JOIN TableB ON join_condition

The FULL OUTER JOIN returns all rows for which a connection exists between *TableA* and *TableB*. In addition, it returns all rows from *TableA* for which no corresponding row exists in *TableB*, with any values selected from *TableB* returned as NULL. In addition, it returns all rows from *TableB* for which no corresponding row exists in *TableA*, with any values selected from *TableA* returned as NULL. In other words, FULL OUTER JOIN acts as a combination of LEFT OUTER JOIN and RIGHT OUTER JOIN.

CROSS JOIN

TableA CROSS JOIN TableB

The CROSS JOIN returns all rows from *TableA* combined with all rows from *TableB*. No ON clause exists to indicate any connecting column between the tables. A CROSS JOIN returns a Cartesian product of the two tables.

Subqueries

SQL Server has an extremely powerful capability for nesting queries that provides a natural and efficient way to express WHERE clause criteria in terms of the results of other queries. You can express most joins as *subqueries*, although this method is often less efficient than performing a join operation. For example, to use the *pubs* database to find all employees of the New Moon Books publishing company, you can write the query as either a join (using ANSI join syntax) or as a subquery.

Here's the query as a join (equijoin, or inner join):

```
SELECT emp_id, lname
FROM employee JOIN publishers ON employee.pub_id=publishers.pub_id
WHERE pub_name='New Moon Books'
```

This is the query as a subquery:

```
SELECT emp_id, lname
FROM employee
WHERE employee.pub_id IN
  (SELECT publishers.pub_id
   FROM publishers WHERE pub_name='New Moon Books')
```

You can write a join (equijoin) as a subquery (subselect), but the converse isn't necessarily true. The equijoin offers an advantage in that the two sides of the equation equal each other and the order doesn't matter. In certain types of subqueries, it does matter which query is the nested query. However, if the query with the subquery can be rephrased as a semantically equivalent JOIN query, the optimizer will do the conversion internally and the performance will be the same whether you write your queries as joins or with subqueries.

Relatively complex operations are simple to perform when you use subqueries. For example, earlier you saw that the *pubs* sample database has four rows in the *authors* table that have no related row in the *titleauthor* table (which prompted our outer-join discussion).

The following simple subquery returns those four *author* rows:

```
SELECT 'Author ID'=A.au_id,
       'Author'=CONVERT(varchar(20), RTRIM(au_lname) + ', '
       + RTRIM(au_fname)), state
FROM authors A
WHERE A.au_id NOT IN
  (SELECT B.au_id FROM titleauthor B)
```

Here's the output:

Author ID	Author	state
341-22-1782	Smith, Meander	KS
527-72-3246	Greene, Morningstar	TN
724-08-9931	Stringer, Dirk	CA
893-72-1158	McBadden, Heather	CA

The IN operation is commonly used for subqueries, either to find matching values (similar to a join) or to find nonmatching values by negating it (NOT IN), as shown above. Using the IN predicate is actually equivalent to = ANY. If you wanted to find every row in *authors* that had at least one entry in the *titleauthor* table, you could use either of these queries.

Here's the query using IN:

```
SELECT 'Author ID'=A.au_id,
       'Author'=CONVERT(varchar(20), RTRIM(au_lname) + ', '
       + RTRIM(au_fname)), state
FROM authors A
```

```
WHERE A.au_id IN
      (SELECT B.au_id FROM titleauthor B)
```

This is the query using equivalent formulation with = ANY:

```
SELECT 'Author ID'=A.au_id,
       'Author'=CONVERT(varchar(20), RTRIM(au_lname) + ', '
       + RTRIM(au_fname)), state
FROM authors A
WHERE A.au_id=ANY
      (SELECT B.au_id FROM titleauthor B)
```

Here's the output:

Author ID	Author	state
172-32-1176	White, Johnson	CA
213-46-8915	Green, Marjorie	CA
238-95-7766	Carson, Cheryl	CA
267-41-2394	O'Leary, Michael	CA
274-80-9391	Straight, Dean	CA
409-56-7008	Bennet, Abraham	CA
427-17-2319	Dull, Ann	CA
472-27-2349	Gringlesby, Burt	CA
486-29-1786	Locksley, Charlene	CA
648-92-1872	Blotchet-Halls, Regi	OR
672-71-3249	Yokomoto, Akiko	CA
712-45-1867	del Castillo, Innes	MI
722-51-5454	DeFrance, Michel	IN
724-80-9391	MacFeather, Stearns	CA
756-30-7391	Karsen, Livia	CA
807-91-6654	Panteley, Sylvia	MD
846-92-7186	Hunter, Sheryl	CA
899-46-2035	Ringer, Anne	UT
998-72-3567	Ringer, Albert	UT

Each of these formulations is equivalent to testing the value of *au_id* in the *authors* table to the *au_id* value in the first row in the *titleauthor* table, and then OR'ing it to a test of the *au_id* value of the second row, and then OR'ing it to a test of the value of the third row, and so on. As soon as one row evaluates to TRUE, the expression is TRUE, and further checking can stop because the row in *authors* qualifies. However, it's an easy mistake to conclude that NOT IN must be equivalent to <> ANY, and some otherwise good discussions of the SQL language have made this exact mistake. More significantly, some products have also erroneously implemented it as such. Although IN is equivalent to = ANY, NOT IN is instead equivalent to <> ALL, not to <> ANY.

Careful reading of the ANSI SQL-92 specifications also reveals that NOT IN is equivalent to <> ALL but is not equivalent to <> ANY. Section 8.4 of the specifications shows that R NOT IN T is equivalent to NOT (R = ANY T). Furthermore, careful study of section 8.7 <quantified comparison predicate> reveals that NOT (R = ANY T) is TRUE if and only if R <> ALL T is TRUE. In other words, NOT IN is equivalent to <> ALL.

By using NOT IN, you're stating that *none* of the corresponding values can match. In other words, *all* of the values must not match (<> ALL), and if even one does match, it's FALSE. With <> ANY, as soon as one value is found to be not equivalent, the expression is TRUE. This, of course, is also the case for every row of *authors*: rows in *titleauthor* will always exist for other *au_id* values, and hence all *authors* rows will have at least one nonmatching row in *titleauthor*. That is, every

row in *authors* will evaluate to TRUE for a test of <> ANY row in *titleauthor*.

The following query using <> ALL returns the same four rows as the earlier one that used NOT IN:

```
SELECT 'Author ID'=A.au_id,  
       'Author'=CONVERT(varchar(20), RTRIM(au_lname) + ', '  
       + RTRIM(au_fname)), state  
FROM authors A  
WHERE A.au_id <> ALL  
      (SELECT B.au_id FROM titleauthor B)
```

Here is the output:

Author ID	Author	state
341-22-1782	Smith, Meander	KS
527-72-3246	Greene, Morningstar	TN
724-08-9931	Stringer, Dirk	CA
893-72-1158	McBadden, Heather	CA

If you had made the mistake of thinking that because IN is equivalent to = ANY, then NOT IN is equivalent to <> ANY, you would have written the query as follows. This returns all 23 rows in the *authors* table!

```
SELECT 'Author ID'=A.au_id,  
       'Author'=CONVERT(varchar(20), RTRIM(au_lname) + ', '  
       + RTRIM(au_fname)), state  
FROM authors A  
WHERE A.au_id <> ANY  
      (SELECT B.au_id FROM titleauthor B)
```

Here's the output:

Author ID	Author	state
172-32-1176	White, Johnson	CA
213-46-8915	Green, Marjorie	CA
238-95-7766	Carson, Cheryl	CA
267-41-2394	O'Leary, Michael	CA
274-80-9391	Straight, Dean	CA
341-22-1782	Smith, Meander	KS
409-56-7008	Bennet, Abraham	CA
427-17-2319	Dull, Ann	CA
472-27-2349	Gringlesby, Burt	CA
486-29-1786	Locksley, Charlene	CA
527-72-3246	Greene, Morningstar	TN
648-92-1872	Blotchet-Halls, Regi	OR
672-71-3249	Yokomoto, Akiko	CA
712-45-1867	del Castillo, Innes	MI
722-51-5454	DeFrance, Michel	IN
724-08-9931	Stringer, Dirk	CA
724-80-9391	MacFeather, Stearns	CA
756-30-7391	Karsen, Livia	CA
807-91-6654	Panteley, Sylvia	MD

846-92-7186	Hunter, Sheryl	CA
893-72-1158	McBadden, Heather	CA
899-46-2035	Ringer, Anne	UT
998-72-3567	Ringer, Albert	UT

The examples just shown use IN, NOT IN, ANY, and ALL to compare values to a set of values from a subquery. This is common. However, it's also common to use expressions and compare a set of values to a single, scalar value. For example, to find *titles* whose royalties exceed the average of all royalty values in the *roysched* table by 25 percent or more, you can use this simple query:

```
SELECT titles.title_id, title, royalty
FROM titles
WHERE titles.royalty >=
  (SELECT 1.25 * AVG(roysched.royalty) FROM roysched)
```

This query is perfectly good because the aggregate function AVG (*expression*) stipulates that the subquery must return exactly one value and no more. Without using IN, ANY, or ALL (or their negations), a subquery that returns more than one row will result in an error. If you incorrectly rewrote the query as follows, without the AVG function, you'd get run-time error 512:

```
SELECT titles.title_id, title, royalty
FROM titles
WHERE titles.royalty >=
  (SELECT 1.25 * roysched.royalty FROM roysched)
```

Here is the output:

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

It is significant that this error is a run-time error and not a syntax error: in the SQL Server implementation, if that subquery didn't produce more than one row, the query would be considered valid and would execute.

The subquery in the following code returns only one row, so the query is valid and returns four rows:

```
SELECT titles.title_id, royalty, title
FROM titles
WHERE titles.royalty >=
  (SELECT 1.25*roysched.royalty FROM roysched
   WHERE roysched.title_id='MC3021' AND lorange=0)
```

Here is the output:

title_id	royalty	title
BU2075	24	You Can Combat Computer Stress!
MC3021	24	The Gourmet Microwave
PC1035	16	But Is It User Friendly?
TC4203	14	Fifty Years in Buckingham Palace Kitchens

However, this sort of query can be dangerous, and you should avoid it or use it only when you know that a PRIMARY KEY or UNIQUE constraint will ensure that the subquery returns only one

value. The query here appears to work, but it's a bug waiting to happen. As soon as another row is added to the *roysched* table—say, with a *title_id* of MC3021 and a *lorange* of 0—the query returns an error. No constraint exists to prevent such a row from being added.

You might argue that SQL Server should determine whether a query formulation could conceivably return more than one row regardless of the data at the time and then disallow such a subquery formulation. The reason it doesn't is that such a query might be quite valid when the database relationships are properly understood, so the power shouldn't be limited to try to protect naïve users. Whether you agree with this philosophy or not, it's consistent with SQL in general—and you should know by now that you can easily write a perfectly legal, syntactically correct query that answers a question in a way that's entirely different from what you thought you were asking!

Correlated Subqueries

You can use powerful correlated subqueries to compare specific rows of one table to a condition in a matching table. For each row otherwise qualifying in the main (or top) query, the subquery is evaluated. Conceptually, a correlated subquery is similar to a loop in programming, although it's entirely without procedural constructs such as *do-while* or *for*. The results of each execution of the subquery must be correlated to a row of the main query. In the next example, for every row in the *titles* table that has a price of \$19.99 or less, the row is compared with each *sales row* for stores in California for which the revenue (*price* × *qty*) is greater than \$250. In other words, "Show me titles with prices of under \$20 for which any single sale in California was more than \$250."

```
SELECT T.title_id, title
FROM titles T
WHERE price <= 19.99
AND T.title_id IN (

SELECT S.title_id FROM sales S, stores ST
  WHERE S.stor_id=ST.stor_id
  AND ST.state='CA' AND S.qty*T.price > 250
  AND T.title_id=S.title_id)
```

Here's the result:

title_id	title
BU7832	Straight Talk About Computers
PS2091	Is Anger the Enemy?
TC7777	Sushi, Anyone?

Correlated subquery, like many subqueries, could have been written as a join (here using the old-style JOIN syntax):

```
SELECT T.title_id, T.title
FROM sales S, stores ST, titles T
WHERE S.stor_id=ST.stor_id
AND T.title_id=S.title_id
AND ST.state='CA'
AND T.price <= 19.99
AND S.qty*T.price > 250
```

It becomes nearly impossible to create alternative joins when the subquery isn't doing a simple IN or when it uses aggregate functions. For example, suppose we want to find titles that lag in sales

for each store. This could be defined as "Find any title for every store in which the title's sales in that store are below 80 percent of the average of sales for all stores that carry that title and ignore titles that have no price established (that is, the price is NULL)." An intuitive way to do this is to first think of the main query that will give us the gross sales for each title and store, and then for each such result, do a subquery that finds the average gross sales for the title for all stores. Then we correlate the subquery and the main query, keeping only rows that fall below the 80 percent standard.

For clarity, notice the two distinct queries, each of which answers a separate question. Then notice how they can be combined into a single correlated query to answer the specific question posed here. All three queries use the old-style JOIN syntax.

```
-- This query computes gross revenues by
-- title for every title and store
SELECT T.title_id, S.stor_id, ST.stor_name, city, state,
       T.price*S.qty
FROM titles AS T, sales AS S, stores AS ST
WHERE T.title_id=S.title_id AND S.stor_id=ST.stor_id
```

```
-- This query computes 80% of the average gross revenue for each
-- title for all stores carrying that title:
SELECT T2.title_id, .80*AVG(price*qty)
```

```
FROM titles AS T2, sales AS S2
WHERE T2.title_id=S2.title_id
GROUP BY T2.title_id
```

```
-- Correlated subquery that finds store-title combinations whose
-- revenues are less than 80% of the average of revenues for that
-- title for all stores selling that title
SELECT T.title_id, S.stor_id, ST.stor_name, city, state,
       Revenue=T.price*S.qty
FROM titles AS T, sales AS S, stores AS ST
WHERE T.title_id=S.title_id AND S.stor_id=ST.stor_id
AND T.price*S.qty <
  (SELECT 0.80*AVG(price*qty)
   FROM titles T2, sales S2
   WHERE T2.title_id=S2.title_id
   AND T.title_id=T2.title_id )
```

And the answer is (from the third query):

title_id	stor_id	stor_name	city state	Revenue
BU1032	6380	Eric the Read Books	Seattle WA	99.95
MC3021	8042	Bookbeat	Portland OR	44.85
PS2091	6380	Eric the Read Books	Seattle WA	32.85
PS2091	7067	News & Brews	Los Gatos CA	109.50
PS2091	7131	Doc-U-Mat: Quality Laundry and Books	Remulade WA	219.00

When the newer ANSI JOIN syntax was first introduced, it wasn't obvious how to use it to write a correlated subquery. It could be that the creators of the syntax forgot about the correlated subquery case, because using the syntax seems like a hybrid of the old and the new: the correlation is still done in the WHERE clause rather than in the JOIN clause.

Examine the two equivalent formulations of the above query using the ANSI JOIN syntax:

```
SELECT T.title_id, S.stor_id, ST.stor_name, city, state,
       Revenue=T.price*S.qty
FROM titles AS T JOIN sales AS S ON T.title_id=S.title_id
     JOIN stores AS ST ON S.stor_id=ST.stor_id
WHERE T.price*S.qty <
      (SELECT 0.80*AVG(price*qty)
       FROM titles T2 JOIN sales S2 ON T2.title_id=S2.title_id
       WHERE T.title_id=T2.title_id )
```

```
SELECT T.title_id, S.stor_id, ST.stor_name, city, state,
       Revenue=T.price*S.qty
FROM titles AS T JOIN sales AS S
     ON T.title_id=S.title_id
     AND T.price*S.qty <
      (SELECT 0.80*AVG(T2.price*S2.qty)
       FROM sales AS S2 JOIN titles AS T2
       ON T2.title_id=S2.title_id
       WHERE T.title_id=T2.title_id)
     JOIN stores AS ST ON S.stor_id=ST.stor_id
```

To completely avoid the old-style syntax with the join condition in the WHERE clause, we could write this using a subquery with a GROUP BY in the FROM clause (by creating a derived table). However, although this gets around having to use the old syntax, it might not be worth it. The query is much less intuitive than either of the preceding two formulations, and it takes twice as many logical reads to execute it.

```
SELECT T.title_id, S.stor_id, ST.stor_name, city, state,
       Revenue = T.price * S.qty
FROM
titles AS T JOIN sales AS S
ON T.title_id = S.title_id
JOIN stores AS ST
ON S.stor_id = ST.stor_id
JOIN
(SELECT T2.title_id, .80 * AVG(price * qty) AS avg_val
FROM titles AS T2 JOIN sales AS S2
ON T2.title_id = S2.title_id
GROUP BY T2.title_id) AS AV ON T.title_id = AV.title_id
AND T.price * S.qty < avg_val
```

Often, correlated subqueries use the EXISTS statement, which is the most convenient syntax to use when multiple fields of the main query are to be correlated to the subquery. (In practice, EXISTS is seldom used other than with correlated subqueries.) EXISTS simply checks for a nonempty set. It returns (internally) either TRUE or NOT TRUE (which we won't refer to as FALSE, given the issues of three-valued logic and NULL). Because no column value is returned and the only thing that matters is whether any rows are returned, convention dictates that a column list isn't specified.

A common use for EXISTS is to answer a query such as "Show me the titles for which no stores have sales."

```
SELECT T.title_id, title FROM titles T
WHERE NOT EXISTS
(SELECT 1
FROM sales S
WHERE T.title_id=S.title_id )
```

Here is the output:

title_id title

MC3026 The Psychology of Computer Cooking
PC9999 Net Etiquette

Conceptually, this query is pretty straightforward. The subquery, a simple equijoin, finds all matches of *titles* and *sales*. Then NOT EXISTS correlates *titles* to those matches, looking for *titles* that don't have even a single row returned in the subquery.

Another common use of EXISTS is to determine whether a table is empty. The optimizer knows that as soon as it gets a single hit using EXISTS, the operation is TRUE and further processing is unnecessary. For example, here's how you determine whether the authors table is empty:

```
SELECT 'Not Empty' WHERE EXISTS (SELECT * FROM authors)
```

Here's an outer-join formulation for the problem described earlier: "Show me the titles for which no stores have sales."

```
SELECT T1.title_id, title FROM titles T1  
LEFT OUTER JOIN sales S ON T1.title_id=S.title_id  
WHERE S.title_id IS NULL
```

Depending on your data and indexes, the outer-join formulation might be faster or slower than a correlated subquery. But before deciding to write your query one way or the other, you might want to come up with a couple of alternative formulations and then choose the one that's fastest in your situation.

In this example, for which little data exists, both solutions run in subsecond elapsed time. But the outer-join query requires fewer than half the number of logical I/Os than does the correlated subquery. With more data, that difference would be significant.

This query works by joining the *stores* and *titles* tables and by preserving the titles for which no store exists. Then, in the WHERE clause, it specifically chooses *only* the rows that it preserved in the outer join. Those rows are the ones for which a title had no matching store.

At other times, a correlated subquery might be preferable to a join, especially if it's a self-join back to the same table or some other exotic join. Here's an example. Given the following table (and assuming that the *row_num* column is guaranteed unique), suppose we want to identify the rows for which *col2* and *col3* are duplicates of another row:

```
row_num col2 col3
```

```
-----
```

```
1 C D  
2 A A  
4 C B  
5 C C  
6 B C  
7 C A  
8 C B  
9 C D  
10 D D
```

We can do this in two standard ways. The first way uses a self-join. In a self-join, the table (or view) is used multiple times in the FROM clause and is aliased at least once. Then it can be treated as an entirely different table and you can compare columns between two "instances" of the same table. A self-join to find the rows having duplicate values for *col2* and *col3* is easy to understand:

```
SELECT DISTINCT A.row_num, A.col2, A.col3  
FROM match_cols AS A, match_cols AS B
```

```
WHERE A.col2=B.col2 AND A.col3=B.col3 AND A.row_num <> B.row_num
ORDER BY A.col2, A.col3
```

row_num	col2	col3
4	C	B
8	C	B
1	C	D
9	C	D

But in this case, a correlated subquery using aggregate functions provides a considerably more efficient solution, especially if many duplicates exist:

```
SELECT A.row_num, A.col2, A.col3 FROM match_cols AS A
WHERE EXISTS (SELECT B.col2, B.col3 FROM match_cols AS B
WHERE B.col2=A.col2
AND B.col3=A.col3
GROUP BY B.col2, B.col3 HAVING COUNT(*) > 1)
ORDER BY A.col2, A.col3
```

This correlated subquery has another advantage over the self-join example—the *row_num* column doesn't need to be unique to solve the problem at hand.

You can take a correlated subquery a step further to ask a seemingly simple question that's surprisingly tricky to answer in SQL: "Show me the stores that have sold every title." Even though it seems like a reasonable request, relatively few people can come up with the correct SQL query, especially if I throw in the restrictions that you aren't allowed to use an aggregate function like `COUNT(*)` and that the solution must be a single `SELECT` statement (that is, you're not allowed to create temporary tables or the like).

The previous query already revealed two titles that no store has sold, so we know that with the existing dataset, no stores can have sales for all titles. For illustrative purposes, let's add sales records for a hypothetical store that does, in fact, have sales for every title. Following that, we'll see the query that finds all stores that have sold every title (which we know ahead of time is only the phony one we're entering here):

```
-- The phony store
INSERT stores (stor_id, stor_name, stor_address, city, state, zip)
VALUES ('9999', 'WE SUPPLY IT ALL', 'One Main St', 'Poulsbo',
'WA', '98370')

-- By using a combination of hard-coded values and selecting every
-- title, generate a sales row for every title
INSERT sales (stor_id, title_id, ord_num, ord_date, qty, payterms)
SELECT '9999', title_id, 'PHONY1', GETDATE(), 10, 'Net 60'
FROM titles

-- Find stores that supply every title
SELECT ST.stor_id, ST.stor_name, ST.city, ST.state
FROM stores ST
WHERE NOT EXISTS
(SELECT * FROM titles T1
WHERE NOT EXISTS
(SELECT * FROM titles T1
WHERE NOT EXISTS
```

```
(SELECT * FROM sales S
WHERE S.title_id=T1.title_id AND ST.stor_id=S.stor_id)
)
```

Here's the result:

```
stor_id stor_name          city state
-----
9999   WE SUPPLY IT ALL    Poulsbo WA
```

Although this query might be difficult to think of immediately, you can easily understand why it works. In English, it says, "Show me the store(s) such that no titles exist that the store doesn't sell." This query consists of the two subqueries that are applied to each store. The bottommost subquery produces all the titles that the store has sold. The upper subquery is then correlated to that bottom one to look for any titles that are *not* in the list of those that the store has sold. The top query returns any stores that aren't in this list. This type of query is known as a *relational division*, and unfortunately, it isn't as easy to express as we'd like. Although the query shown is quite understandable, once you have a solid foundation in SQL, it's hardly intuitive. As is almost always the case, you could probably use other formulations to write this query.

If you think of the query in English as "Find the stores that have sold as many unique titles as there are total unique titles," you'll find the following formulation somewhat more intuitive:

```
SELECT ST.stor_id, ST.stor_name
FROM stores ST, sales SA, titles T
WHERE SA.stor_id=ST.stor_id AND SA.title_id=T.title_id
GROUP BY ST.stor_id,ST.stor_name
HAVING COUNT(DISTINCT SA.title_id)=(SELECT COUNT(*) FROM titles T1)
```

The following formulation runs much more efficiently than either of the previous two. The syntax is similar to the preceding one but its approach is novel because it's just a standard subquery, not a join or a correlated subquery. You might think it's an illegal query, since it does a GROUP BY and a HAVING without an aggregate in the select list of the first subquery. But that's OK, both in terms of what SQL Server allows and in terms of the ANSI specification

```
-- Find stores that have sold every title
SELECT stor_id, stor_name FROM stores WHERE stor_id IN
(SELECT stor_id FROM sales GROUP BY stor_id
HAVING COUNT(DISTINCT title_id)=(SELECT COUNT(*) FROM titles)
)
```

Here's a formulation that uses a derived table—a feature that allows you to use a subquery in a FROM clause. This capability lets you alias a virtual table returned as the result set of a SELECT statement, and then lets you use this result set as if it were a real table. This query also runs efficiently.

```
SELECT ST.stor_id, ST.stor_name
FROM stores ST,
(SELECT stor_id, COUNT(DISTINCT title_id) AS title_count
FROM sales
GROUP BY stor_id
) as SA
WHERE ST.stor_id=SA.stor_id AND SA.title_count=
(SELECT COUNT(*) FROM titles)
```

Functions

Introduction

The SQL language includes many functions that you can use to summarize data from a column within a table. Collectively, the functions that enable you to summarize data are referred to as *aggregate* functions. You might also hear aggregate functions referred to as *group* functions because they operate on groups of rows to provide you with a single result.

Use the following syntax with aggregate functions:

```
USE database
SELECT FUNCTION(expression)
FROM table
```

You will typically replace *expression* with a column name. You can optionally include an AS clause after the function so that SQL Server can display a heading for the column in the result set. If you do not specify an alias when you use a function, SQL Server does not display a column heading. When you use an aggregate function in your SELECT statement, you cannot include other columns in the SELECT clause unless you use a GROUP BY clause.

The following query shows you how to find the highest rental price for a movie in the movie table:

```
USE movies
SELECT MAX(rental_price) AS `Highest Rental Fee`
FROM movie
```

It enables you to count the number of rows in the customer table in order to determine the total number of customers:

```
USE movies
SELECT COUNT(*)
FROM customer
```

Aggregate functions and data types

You will most often use the aggregate functions on numeric data, but you can use some of these functions against character data.

You can use the MAX() function against character-based columns as follows:

```
USE movies
SELECT MAX(title)
FROM movie
```


In this example, SQL Server returns the movie title of 'Young Frankenstein' because alphabetically, it is the last movie title in the table. You cannot use the MIN() and MAX() functions against the bit data type. Some of the other considerations for aggregate functions and data types include:

- You can use the COUNT function against all data types. COUNT is the only aggregate function you can use against text, ntext, and image data types.
- You can use only the int, smallint, decimal, numeric, float, real, money, and smallmoney data types in the SUM() and AVG() functions.

Null values

Other than the COUNT function, the aggregate functions ignore null values in columns. All of the aggregate functions base their calculations on the premise that the values in columns are significant only if those values are not null. If you count the number of rows based on a column with null values (such as COUNT(zip), SQL Server skips any rows that have null values in that column. If you use COUNT(*), you will get the actual row count-even if a row has nothing but null values in all columns.

Before you begin:

You are logged on to Windows NT as user#. You have created a database named movies and tables within it named movie, category, customer, rental, and rental_detail. You have defined primary key, foreign key, default, and check constraints on the tables, and created nonclustered indexes based on your primary keys. You have imported data into the tables. You have created database diagrams for both the movies and pubs databases.

1. Write a query to find the average price of movies with a G rating.

```
USE movies
SELECT AVG(rental_price) AS `Average Rental Fee`
FROM movie
WHERE rating = `G`
```

2. What query would you use to find the title and price of the highest priced movie? (Hint:You must use a subquery to find this information.)

```
USE movies
SELECT title, rental_price
FROM movie
WHERE rental_price = (SELECT MAX(rental_price) FROM movie)
```

Using group by to group the results of aggregate functions

You use the GROUP BY clause to divide the rows of a table into groups and then display summary results for a specific column. You should not use the GROUP BY clause on a column in which multiple rows have null values because SQL Server will treat all of the rows with null values in that column as a group.

You might want to count the number of movies you have in stock for each rating (G, PG, and so on). To find this information, you must use a GROUP BY clause to group the movies by rating and then count the number of movies in each group. Use the following syntax:

```
USE movies
SELECT rating, COUNT(movie_num)
FROM movie
GROUP BY rating
```

In this example, you can use COUNT(movie_num) because the structure of the movie table does not permit null values in the movie_num column.

The GROUP BY clause requires a one-to-one relationship between the columns you specify in the SELECT statement (other than the aggregate function) and the GROUP BY clause. For example, the following query is invalid because the rating column is not included in the SELECT clause:

```
USE movies
SELECT COUNT(movie_num)
FROM movie
GROUP BY rating
```

This query is also invalid because the rating column is not referenced in a GROUP BY clause:

```
USE movies
SELECT rating, COUNT(movie_num)
FROM movie
```

You can use a GROUP BY clause to enable you to calculate the total rental fee collected for each invoice in the rental_detail table:

```
USE movies
SELECT invoice_num, SUM(rental_price)
FROM rental_detail
GROUP BY invoice_num
```

Using a WHERE clause

You can also add a WHERE clause to a query that contains an aggregate function in order to restrict the groups on which the aggregate function performs its calculations. If you use a WHERE clause, SQL Server groups only the rows that meet the condition you specify. For example, if you want to see the average price of movies with a PG or R rating, you could use the following query:

```
USE movies
SELECT rating, AVG(rental_price)
FROM movie
WHERE rating = `PG` OR rating = `R`
GROUP BY rating
```

The WHERE clause must precede the GROUP BY clause. If you reverse the order of these clauses, you will get a syntax error.

Using group by with having

If you want to restrict the rows returned by a query in which you are using an aggregate function and a GROUP BY clause, you can use a HAVING clause instead of a WHERE clause. The HAVING clause offers you a distinct advantage over a WHERE clause because it enables you to use aggregate functions to restrict the rows returned in the result set.

You could use the following query to display the rating and average price of all movies for each rating as long as the average price of those movies is greater than \$2.50:

```
USE movies
SELECT rating, AVG(rental_price)
FROM movie
GROUP BY rating
HAVING AVG(rental_price) >= 2.50
```

One other difference between a WHERE clause and a HAVING clause is that the WHERE clause restricts the groups of rows on which the aggregate function calculates its results; in contrast, the aggregate function calculates values for all groups of rows but only displays those that meet the HAVING clause's criteria in the result set.

1. Design a query based on the movies database that shows the total rental price collected for each invoice in the rental_detail table.

```
USE movies
SELECT invoice_num, SUM(rental_price) AS `Total Rental Price`
FROM rental_detail
GROUP BY invoice_num
```

2. Design a query on the movies database that shows all invoices and their total rental price where the total price was more than \$4.00. (You should get 22 rows in the result set.)

```
USE movies
SELECT invoice_num, SUM(rental_price) AS `Total Rental Price`
FROM rental_detail
GROUP BY invoice_num
HAVING SUM(rental_price) > 4
```

3. Design a query that lists the category and average rental price of movies in the Comedy category. (The category_num for Comedy is 1.)

```
SELECT category_num, AVG(rental_price) AS 'Average Rental Price'  
FROM movie  
GROUP BY category_num  
HAVING category_num = 1
```

Using group by with the WITH ROLLUP operator

You can use the WITH ROLLUP operator to enable SQL Server to calculate totals for the columns you include in a GROUP BY clause. You will typically use the WITH ROLLUP operator on queries that include at least two columns in the GROUP BY clause. The WITH ROLLUP operator then adds a totals row for each unique value in the first column of the GROUP BY clause.

Consider the following query:

```
USE northwind  
SELECT orderid, productid, SUM(quantity) AS `Total Quantity`  
FROM [order details]  
GROUP BY orderid, productid WITH ROLLUP  
ORDER BY orderid, productid
```

This query displays the order ID, product ID, and the total quantity of each product purchased for each order ID. In addition, because the query includes the WITH ROLLUP operator, SQL Server Query Analyzer not only totals the quantity purchased for each product of each order, but also generates a total for each order. SQL Server adds a row to the result set to display the running totals. You can identify running totals because SQL Server Query Analyzer indicates them by NULL in the result set. (In this example, the running total for all products for a particular orderid displays NULL in the productid column.)

Using group by with the WITH CUBE operator

In contrast to the WITH ROLLUP operator, the WITH CUBE operator calculates totals for all combinations of the columns in your GROUP BY clause (instead of calculating totals just for the first column).

Using group by with the WITH CUBE operator

In contrast to the WITH ROLLUP operator, the WITH CUBE operator calculates totals for all combinations of the columns in your GROUP BY clause (instead of calculating totals just for the first column).

```
USE northwind  
SELECT orderid, productid, SUM(quantity) AS `Total Quantity`  
FROM [order details]  
GROUP BY orderid, productid WITH CUBE  
ORDER BY orderid, productid
```

In this example, SQL Server calculates not only the total quantity of products sold in each order, but also the total quantity of all products sold (regardless of order ID and product ID), as well as the total quantity of each product sold (regardless of the order ID). SQL Server displays NULL in the columns with calculations resulting from the WITH CUBE operator. In this example, you will see additional rows in the result set over what you saw when you used the WITH ROLLUP operator

Using the GROUPING function

You can use the GROUPING function to add a column to indicate whether a column has been used to return the result set. Further, you can use the GROUPING function with either the CUBE or ROLLUP operators. Using the GROUPING function makes it easier for you to see which columns have been used to generate the total rows.

Use the GROUPING function along with the WITH CUBE operator, use the following syntax:

```
USE northwind
SELECT orderid, GROUPING(orderid), productid, GROUPING(productid), SUM(quantity) AS
`Total Quantity`
FROM [order details]
GROUP BY orderid, productid
WITH CUBE
ORDER BY orderid, productid
```

In this example, SQL Server will display a `1` in the column it is using to group the results on, and a `0` if it is not using the column. The GROUPING thus enables you to identify which columns SQL Server used to generate its summary rows.

Using Compute and Compute by Clause

SQL Server supports both the COMPUTE and COMPUTE BY clauses for generating summary rows in the result set. Both of these keywords are not included in the ANSI-SQL standard. You can use these keywords for generating a result set, but you should not use them in any applications.

You can use COMPUTE to display the total quantity of all products purchased in the order details table by using the following query:

```
USE northwind
SELECT orderid, productid, quantity
FROM [order details]
ORDER BY orderid, productid
COMPUTE SUM(quantity)
```

This query adds a row to the end of the result set with the total quantity of all products purchased.

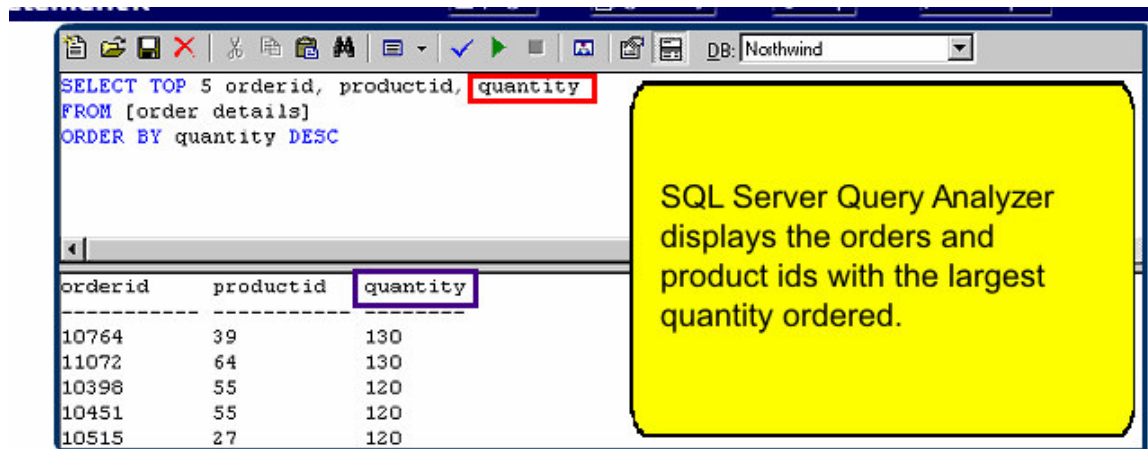
In contrast, you can use the COMPUTE BY clause to calculate a total quantity purchased in each order by using the following query:

```
USE northwind
SELECT orderid, productid, quantity
FROM [order details]
```

```
ORDER BY orderid, productid
COMPUTE SUM(quantity)BY orderid
```

Display the top n rows in a result set

You can use the TOP *n* or TOP *n* PERCENT keywords to specify that you want SQL Server to return only a specific number of rows in the result set. You must include an ORDER BY clause so that SQL Server can determine the top *x* rows.



1. By using the movies database, write a query to list the titles of the top three movies based on rental price. (Hint: make sure you use the DISTINCT keyword so that you get three unique titles.)

```
USE movies
SELECT DISTINCT TOP 3 title, rental_price
FROM movie
ORDER BY rental_price DESC
```

2. Use the movies database to write a query listing the top three invoice numbers based on total money spent renting movies.

```
USE movies
SELECT TOP 3 invoice_num, SUM(rental_price)
FROM rental_detail
GROUP BY invoice_num
ORDER BY SUM(rental_price) DESC
```

3. Use the pubs database to write a query listing the titles and prices of the top five most expensive books.

```
USE pubs
SELECT TOP 5 title, price
FROM titles
ORDER BY price DESC
```

Index

Indexes are the other significant user-defined, on-disk data structure besides tables. An index provides fast access to data when the data can be searched by the value that is the index key.

In this chapter, you'll understand the physical organization of index pages for both types of SQL Server indexes, clustered and nonclustered, the various options available when you create and re-create indexes, when, and why to rebuild your indexes, SQL Server 2000's online index defragmentation utility and about a tool for determining whether your indexes need defragmenting.

Indexes allow data to be organized in a way that allows optimum performance when you access or modify it. SQL Server does not *need* indexes to successfully retrieve results for your SELECT statements or to find and modify the specified rows in your data modification statements. As your tables get larger, the value of using proper indexes becomes obvious. You can use indexes to quickly find data rows that satisfy conditions in your WHERE clauses, to find matching rows in your JOIN clauses, or to efficiently maintain uniqueness of your key columns during INSERT and UPDATE operations. In some cases, you can use indexes to help SQL Server sort, aggregate, or group your data or to find the first few rows as indicated in a TOP clause.

It is the job of the query optimizer to determine which indexes, if any, are most useful in processing a specific query. The final choice of which indexes to use is one of the most important components of the query optimizer's execution plan. In this chapter, we'll understand what indexes look like and how they can speed up your queries.

Index Organization

Think of the indexes that you see in your everyday life—those in books and other documents. Suppose you're trying to write a SELECT statement in SQL Server using the CASE expression, and you're using two SQL Server documents to find out how to write the statement. One document is the *Microsoft SQL Server 2000 Transact-SQL Language Reference Manual*. The other is this book, *Inside Microsoft SQL Server 2000*. You can quickly find information in either book about CASE, even though the two books are organized differently.

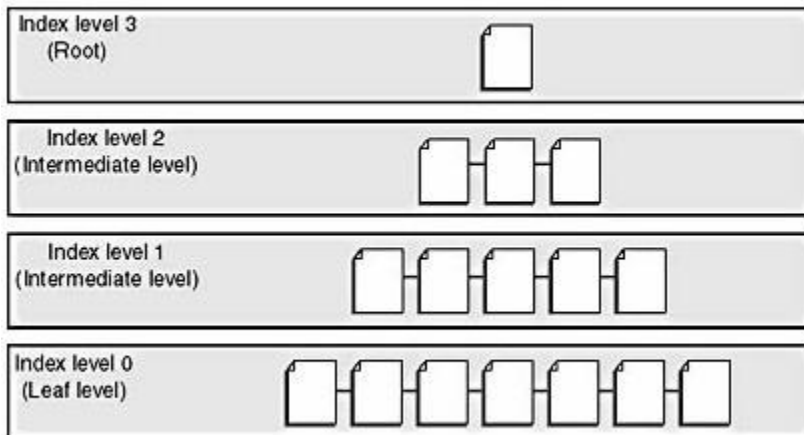
In the T-SQL language reference, all the information is organized alphabetically. You know that CASE will be near the front, so you can just ignore the last 80 percent of the book. Keywords are shown at the top of each page to tell you what topics are on that page. So you can quickly flip through just a few pages and end up at a page that has BREAK as the first keyword and CEILING as the last keyword, and you know that CASE will be on this page. If CASE is not on this page, it's not in the book. And once you find the entry for CASE, right between BULK INSERT and CAST, you'll find all the information you need, including lots of helpful examples.

Next, you try to find CASE in this book. There are no helpful key words at the top of each page, but there's an index at the back of the book and all the index entries are organized alphabetically. So again, you can make use of the fact that CASE is near the front of the alphabet and quickly find it between "cascading updates" and "case-insensitive searches." However, unlike in the reference manual, once you find the word CASE, there are no nice neat examples right in front of you. Instead, the index gives you pointers. It tells you what pages to look at—it might list two or three pages that point you to various places in the book. If you look up SELECT in the back of the book, however, there might be dozens of pages listed, and if you look up SQL Server, there might be hundreds.

These searches through the two books are analogous to using clustered and nonclustered indexes. In a clustered index, the data is actually stored in order, just as the reference manual has all the main topics in order. Once you find the data you're looking for, you're done with the search. In a nonclustered index, the index is a completely separate structure from the data itself. Once you find what you're looking for in the index, you have to follow pointers to the actual data. A nonclustered index in SQL Server is very much like the index in the back of this book. If there is only one page reference listed in the index, you can quickly find the information. If a dozen pages are listed, it's quite a bit slower. If hundreds of pages are listed, you might think there's no point in using the index at all. Unless you can narrow down your topic of interest to something more specific, the index might not be of much help.

Both clustered and nonclustered indexes in SQL Server store their information using standard B-trees, as shown in figure 21. A B-tree provides fast access to data by searching on a key value of the index. B-trees cluster records with similar keys. The *B* stands for *balanced*, and balancing the tree is a core feature of a B-tree's usefulness. The trees are managed, and branches are grafted as necessary, so that navigating down the tree to find a value and locate a specific record takes only a few page accesses. Because the trees are balanced, finding any record requires about the same amount of resources, and retrieval speed is consistent because the index has the same depth throughout.

Figure 21 A B-Tree for SQL Server Index



An index consists of a tree with a root from which the navigation begins, possible intermediate index levels, and bottom-level leaf pages. You use the index to find the correct leaf page. The number of levels in an index will vary depending on the number of rows in the table and the size of the key column or columns for the index. If you create an index using a large key, fewer entries will fit on a page, so more pages (and possibly more levels) will be needed for the index. On a qualified select, update, or delete, the correct leaf page will be the lowest page of the tree in which one or more rows with the specified key or keys reside. A qualified operation is one that affects only specific rows that satisfy the conditions of a WHERE clause, as opposed to accessing the whole table. In any index, whether clustered or nonclustered, the leaf level contains every key value, in key sequence.

Clustered Indexes

The leaf level of a clustered index contains the data pages, not just the index keys. Another way to say this is that the data itself is part of the clustered index. A clustered index keeps the data in a table ordered around the key. The data pages in the table are kept in a doubly linked list called the *page chain*. The order of pages in the page chain, and the order of rows on the data pages, is the order of the index key or keys. Deciding which key to cluster on is an important performance consideration. When the index is traversed to the leaf level, the data itself has been *retrieved*, not simply *pointed to*.

Because the actual page chain for the data pages can be ordered in only one way, a table can have only one clustered index. The query optimizer strongly favors a clustered index because such an index allows the data to be found directly at the leaf level. Because it defines the actual order of the data, a clustered index allows especially fast access for queries looking for a range of values. The query optimizer detects that only a certain range of data pages must be scanned.

Most tables should have a clustered index. If your table will have only one index, it generally should be clustered. Many documents describing SQL Server indexes will tell you that the clustered index physically stores the data in sorted order. This can be misleading if you think of physical storage as the disk itself. If a clustered index had to keep the data on the actual disk in a particular order, it could be prohibitively expensive to make changes. If a page got too full and had to be split in two, all the data on all the succeeding pages would have to be moved down. Sorted order in a clustered index simply means that the data page chain is in order. If SQL Server follows the page chain, it can access each row in clustered index order, but new pages can be added by simply adjusting the links in the page chain. In SQL Server 2000, all clustered indexes are unique. If you build a clustered index without specifying the *unique* keyword, SQL Server forces uniqueness by adding a *uniqueifier* to the rows when necessary. This uniqueifier is a 4-byte value added as an additional sort key to only the rows that have duplicates of their primary sort key.

Nonclustered Indexes

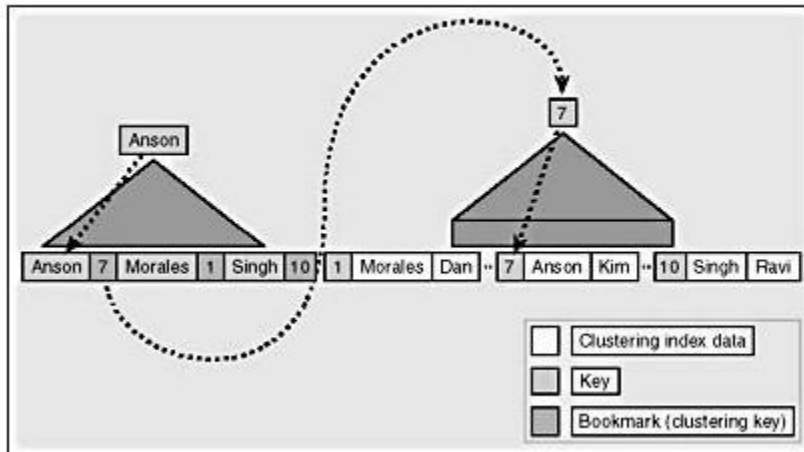
In a nonclustered index, the lowest level of the tree (the leaf level) contains a bookmark that tells SQL Server where to find the data row corresponding to the key in the index. A bookmark can take one of two forms. If the table has a clustered index, the bookmark is the clustered index key for the corresponding data row. If the table is a heap (in other words, it has no clustered index), the bookmark is a RID, which is an actual row locator in the form File#:Page#:Slot#. (In contrast, in a clustered index, the leaf page *is* the data page.)

The presence or absence of a nonclustered index doesn't affect how the data pages are organized, so you're not restricted to having only one nonclustered index per table, as is the case with clustered indexes. Each table can include as many as 249 nonclustered indexes, but you'll usually want to have far fewer than that.

When you search for data using a nonclustered index, the index is traversed and then SQL Server retrieves the record or records pointed to by the leaf-level indexes. For example, if you're looking for a data page using an index with a depth of three—a root page, one intermediate page, and the leaf page—all three index pages must be traversed. If the leaf level contains a clustered index key, all the levels of the clustered index then have to be traversed to locate the specific row. The clustered index will probably also have three levels, but in this case remember that the leaf level is the data itself. There are two additional index levels separate from the data, typically one less than the number of levels needed for a nonclustered index. The data page still must be retrieved, but because it has been exactly identified there's no need to scan the entire table. Still, it takes six logical I/O operations to get one data page. You can see that a nonclustered index is a win only if it's highly selective.

Figure 22 illustrates this process without showing you the individual levels of the B-trees. To find the first name for the employee named Anson, which has a nonclustered index on the last name and a clustered index on the employee ID. The nonclustered index uses the clustered keys as its bookmarks. Searching the index for Anson, SQL Server finds that the associated clustered index key is 7. It then traverses the clustered index looking for the row with a key of 7, and it finds Kim as the first name in the row I'm looking for.

Figure22 SQL server traverse both clusteres and non clustered Indexes to find the first name for the employee names Anson



Creating an Index

The typical syntax for creating an index is straightforward:

```
CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED] INDEX index_name
ON table_name (column_name [ASC | DESC][,...n])
```

When you create an index, you must specify a name for it. You must also specify the table on which the index will be built, and then one or more columns. For each column, you can specify that the leaf level will store the key values sorted in either ascending (ASC) or descending (DESC) order. The default is ascending. You can specify that SQL Server must enforce uniqueness of the key values by using the keyword UNIQUE. If you don't specify UNIQUE, duplicate key values will be allowed. You can specify that the index be either clustered or nonclustered. Nonclustered is the default.

CREATE INDEX has some additional options available for specialized purposes. You can add a WITH clause to the CREATE INDEX command:

```
[WITH
[FILLFACTOR = fillfactor]
[,...] [PAD_INDEX]
[,...] IGNORE_DUP_KEY]
[,...] DROP_EXISTING]
[,...] STATISTICS_NORECOMPUTE]
```

```
[[,] SORT_IN_TEMPDB]  
]
```

FILLFACTOR is probably the most commonly used of these options. FILLFACTOR lets you reserve some space on each leaf page of an index. In a clustered index, since the leaf level contains the data, you can use FILLFACTOR to control how much space to leave in the table itself. By reserving free space, you can later avoid the need to split pages to make room for a new entry. But remember that FILLFACTOR is not maintained; it indicates only how much space is reserved with the existing data at the time the index is built. If you need to, you can use the DBCC DBREINDEX command to rebuild the index and reestablish the original FILLFACTOR specified.

If you plan to rebuild all of a table's indexes, simply specify the clustered index with DBCC DBREINDEX. Doing so internally rebuilds the entire table and all nonclustered indexes.

FILLFACTOR isn't usually specified on an index-by-index basis, but you can specify it this way for fine-tuning. If FILLFACTOR isn't specified, the serverwide default is used. The value is set for the server via `sp_configure, fillfactor`. This value is 0 by default, which means that leaf pages of indexes are made as full as possible. FILLFACTOR generally applies only to the index's leaf page (the data page for a clustered index). In specialized and high-use situations, you might want to reserve space in the intermediate index pages to avoid page splits there, too. You can do this by specifying the PAD_INDEX option, which uses the same value as FILLFACTOR.

The DROP_EXISTING option specifies that a given index should be dropped and rebuilt as a single transaction. This option is particularly useful when you rebuild clustered indexes. Normally, when a clustered index is dropped, every nonclustered index has to be rebuilt to change its bookmarks to RIDs instead of the clustering keys. Then, if a clustered index is built (or rebuilt), all the nonclustered indexes need to be rebuilt again to update the bookmarks. The DROP_EXISTING option of the CREATE INDEX command allows a clustered index to be rebuilt without having to rebuild the nonclustered indexes twice. If you are creating the index on the exact same keys that it had previously, the nonclustered indexes do not need to be rebuilt at all. If you are changing the key definition, the nonclustered indexes are rebuilt only once, after the clustered index is rebuilt.

You can ensure the uniqueness of an index key by defining the index as UNIQUE or by defining a PRIMARY KEY or UNIQUE constraint. If an UPDATE or INSERT statement would affect multiple rows, and if even one row is found that would cause duplicate keys defined as unique, the entire statement is aborted and no rows are affected. Alternatively, when you create the unique index, you can use the IGNORE_DUP_KEY option so that a duplicate key error on a multiple-row INSERT won't cause the entire statement to be rolled back. The nonunique row will be discarded, and all other rows will be inserted or updated. IGNORE_DUP_KEY doesn't allow the uniqueness of the index to be violated; instead, it makes a violation in a multiple-row data modification nonfatal to all the nonviolating rows.

The SORT_IN_TEMPDB option allows you to control where SQL Server performs the sort operation on the key values needed to build an index. The default is that SQL Server uses space from the filegroup on which the index is to be created. While the index is being built, SQL Server scans the data pages to find the key values and then builds leaf-level index rows in internal sort buffers. When these sort buffers are filled, they are written to disk. The disk heads for the database can then move back and forth between the base table pages and the work area where the sort buffers are being stored. If, instead, your CREATE INDEX command includes the option SORT_IN_TEMPDB, performance can be greatly improved, particularly if your *tempdb* database is on a separate physical disk from the database you're working with, with its own controller. You can optimize head movement because two separate heads read the base table pages and

manage the sort buffers. You can get even more improvement in index creation speed if your *tempdb* is on a faster disk than your user database and you use the SORT_IN_TEMPDB option. As an alternative to using the SORT_IN_TEMPDB option, you can create separate filegroups for a table and its indexes. (That is, the table is on one filegroup and its indexes are on another.) If the two filegroups are on different disks with their own controllers, you can also minimize the disk head movement.

Constraints and Indexes

When you declare a PRIMARY KEY or UNIQUE constraint, a unique index is created on one or more columns, just as if you had used the CREATE INDEX command. The names of indexes that are built to support these constraints are the same as the constraint names. In terms of internal storage and maintenance of indexes, there is no difference between unique indexes created using the CREATE INDEX command and indexes created to support constraints. The query optimizer makes decisions based on the presence of the unique index rather than on the fact that a column was declared as a primary key. How the index got there in the first place is irrelevant to the query optimizer.

When you create a table that includes PRIMARY KEY or UNIQUE constraints, you can specify whether the associated index will be clustered or nonclustered, and you can also specify the fillfactor. Since the fillfactor applies only at the time the index is created, and since there is no data when you first create the table, it might seem that specifying the fillfactor at that time is completely useless. However, if after the table is populated you decide to use DBCC DBREINDEX to rebuild all your indexes, you can specify a fillfactor of 0 to indicate that SQL Server should use the fillfactor that was specified when the index was created. You can also specify a fillfactor when you use ALTER TABLE to add a PRIMARY KEY or UNIQUE constraint to a table, and if the table already had data in it, the fillfactor value is applied when you build the index to support the new constraint.

If you check the documentation for CREATE TABLE and ALTER TABLE, you'll see that the SORT_IN_TEMPDB option is not available for either command. It really doesn't make sense to specify a sort location when you first create the table because there's nothing to sort. However, the fact that you can't specify this alternate location when you add a PRIMARY KEY or UNIQUE constraint to a table with existing data seems like an oversight. Also note that SORT_IN_TEMPDB is not an option when you use DBCC DBREINDEX. Again, there's no reason why it couldn't have been included, but it isn't available in this release.

The biggest difference between indexes created using the CREATE INDEX command and indexes that support constraints is in how you can drop the index. The DROP INDEX command allows you to drop only indexes that were built with the CREATE INDEX command. To drop indexes that support constraints, you must use ALTER TABLE to drop the constraint. In addition, to drop a PRIMARY KEY or UNIQUE constraint that has any FOREIGN KEY constraints referencing it, you must first drop the FOREIGN KEY constraint. This can leave you with a window of vulnerability while you redefine your constraints and rebuild your indexes. While the FOREIGN KEY constraint is gone, an INSERT statement can add a row to the table that violates your referential integrity.

One way to avoid this problem is to use DBCC DBREINDEX, which drops and rebuilds all your indexes on a table in a single transaction, without requiring the auxiliary step of removing FOREIGN KEY constraints. Alternatively, you can use the CREATE INDEX command with the DROP_EXISTING option. In most cases, you cannot use this command on an index that supports a constraint. However, there is an exception.

The following command attempts to rebuild the index on the *title_id* column of the *titles* table in the *pubs* database:

```
CREATE CLUSTERED INDEX UPKCL_titleidind ON titles(title_id)
  WITH DROP_EXISTING
```

SQL Server returned this error message to me:

```
Server: Msg 1907, Level 16, State 1, Line 1
Cannot re-create index 'UPKCL_titleidind'. The new index definition
does not match the constraint being enforced by the existing index.
```

How would you have known that this index supported a constraint? First of all, the name includes *UPKCL*, which is a big clue. However, the output of *sp_helpindex* tells us only the names of the indexes, the property (clustered or unique), and the columns the index is on. It doesn't tell us if the index supports a constraint. However, if we execute *sp_help* on a table, the output will tell us that *UPKCL_titleidind* is a PRIMARY KEY constraint. The error message indicates that we can't use the *DROP_EXISTING* clause to rebuild this index because the new definition doesn't match the current index. We can use this command as long as the properties of the new index are exactly the same as the old. In this case, we can rephrase the command as follows so that the *CREATE INDEX* is successful:

```
CREATE UNIQUE CLUSTERED INDEX UPKCL_titleidind ON titles(title_id)
  WITH DROP_EXISTING
```

The Structure of Index Pages

Index pages are structured much like data pages. As with all other types of pages in SQL Server, index pages have a fixed size of 8 KB, or 8192 bytes. Index pages also have a 96-byte header, but just like in data pages, there is an offset array at the end of the page with two bytes for each row to indicate the offset of that row on the page. However, if you use *DBCC PAGE* with style 1 to print out the individual rows on an index page, the slot array is not shown. If you use style 2, it will only print out all the bytes on a page with no attempt to separate the bytes into individual rows. You can then look at the bottom of the page and notice that each set of two bytes does refer to a byte offset on the page. Each index has a row in the *sysindexes* table, with an *indid* value of either 1, for a clustered index, or a number between 2 and 250, indicating a nonclustered index. An *indid* value of 255 indicates LOB data (*text*, *ntext* or *image* information). The *root* column value contains a file number and page number where the root of the index can be found. You can then use *DBCC PAGE* to examine index pages, just as you do for data pages.

The header information for index pages is almost identical to the header information for data pages. The only difference is the value for *type*, which is 1 for data and 2 for index. The header of an index page also has nonzero values for *level* and *indexid*. For data pages, these values are both always 0.

There are basically three different kinds of index pages: leaf level for nonclustered indexes, node (nonleaf) level for clustered indexes, and node level for nonclustered indexes. There isn't really a separate structure for leaf level pages of a clustered index because those are the data pages, which we've already seen in detail. There is, however, one special case for leaf-level clustered index pages.

Clustered Index Rows with a Uniqueifier

If your clustered index was not created with the UNIQUE property, SQL Server adds a 4-byte field when necessary to make each key unique. Since clustered index keys are used as bookmarks to identify the base rows being referenced by nonclustered indexes, there needs to be a unique way to refer to each row in a clustered index. SQL Server adds the uniqueifier only when necessary—that is, when duplicate keys are added to the table. We'll use the same table which we initially used to illustrate the row structure of a table with all fixed-length columns. We'll create an identical table with a different name, then add a clustered (nonunique) index to the table:

Clustered Index Rows with a Uniqueifier

If your clustered index was not created with the UNIQUE property, SQL Server adds a 4-byte field when necessary to make each key unique. Since clustered index keys are used as bookmarks to identify the base rows being referenced by nonclustered indexes, there needs to be a unique way to refer to each row in a clustered index. SQL Server adds the uniqueifier only when necessary—that is, when duplicate keys are added to the table. We'll use the same table which we initially used to illustrate the row structure of a table with all fixed-length columns. We'll create an identical table with a different name, then add a clustered (nonunique) index to the table:

```
USE pubs
GO
CREATE TABLE Clustered_Dupes
  (Col1 char(5) NOT NULL,
   Col2 int NOT NULL,
   Col3 char(3) NULL,
   Col4 char(6) NOT NULL,
   Col5 float NOT NULL)
GO
CREATE CLUSTERED INDEX CI_dupes_col1 ON Clustered_Dupes(col1)
```

The rows in *syscolumns* are without the clustered index. However, if you look at the *sysindexes* row for this table, you'll notice something different.

```
SELECT first, indid, keycnt, name FROM sysindexes
WHERE id = object_id ('Clustered_Dupes')
```

RESULT:

first	indid	keycnt	name
-----	-----	-----	-----
0x000000000000	1	2	CI_dupes_col1

The column called *keycnt*, which indicates the number of keys an index has, is 2. If we had created this index using the UNIQUE qualifier, the *keycnt* value would be 1. If we had looked at the *sysindexes* row before adding a clustered index, when the table was still a heap, the row for the table would have had a *keycnt* value of 0. Add the initial row and then look at *sysindexes* to find the first page of the table:

```
INSERT Clustered_Dupes VALUES ('ABCDE', 123, null, 'CCCC', 4567.8)
GO
```


Without this bit on, TagA would have a value of 0x10. The "extra" variable-length portions of the second two rows are shaded in the figure. You can see that 8 extra bytes are added when we have a duplicate row. In this case, the first 4 extra bytes are added because the uniqueifier is considered a variable-length column. Since there were no variable-length columns before, SQL Server adds 2 bytes to indicate the number of variable-length columns present. These bytes are at offsets 33-34 in these rows with the duplicate keys and have the value of 1. The next 2 bytes, (offsets 35-36) indicate the position where the first variable length column ends. In both these rows, the value is 0x29, which converts to 41 decimal. The last 4 bytes (offsets 37-40) are the actual uniqueifier. In the second row, which has the first duplicate, the uniqueifier is 1. The third row has a uniqueifier of 2.

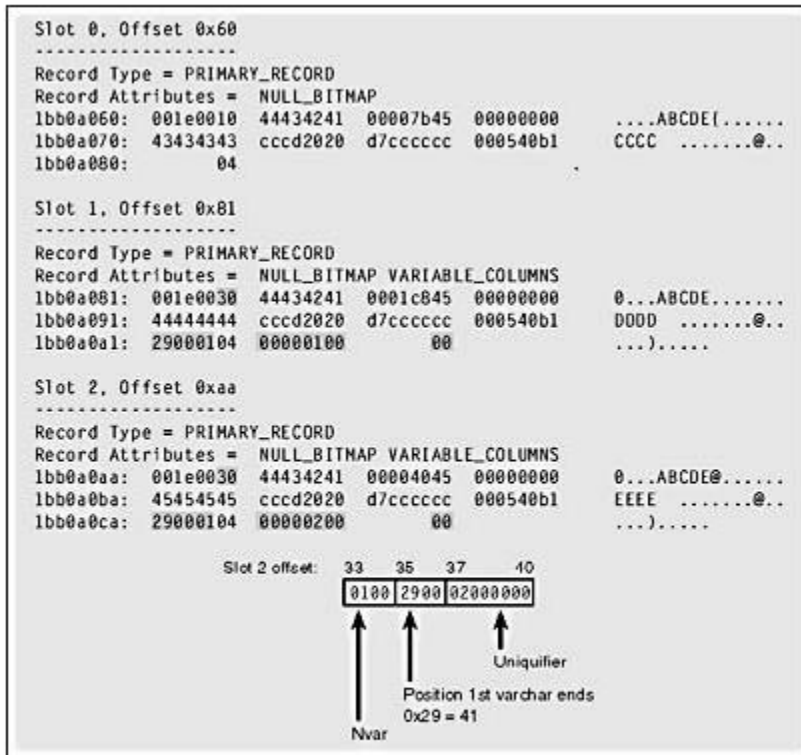


Figure 8-4. Three data rows containing duplicate values in the clustered key column.

Index Space Requirements

Now you understand the structure and number of bytes in individual index rows, but you really need to be able to translate that into overall index size. In general, the size of an index is based on the size of the index keys, which determines how many index rows can fit on an index page and the number of rows in the table.

B-Tree Size

When we talk about index size, we usually mean the size of the index tree. The clustered index does include the data, but since you still have the data even if you drop the clustered index, we're usually just interested in how much additional space the nonleaf levels require. A clustered index's node levels typically take up very little space. You have one index row for each page of table data, so the number of index pages at the level above the leaf (level 1) is the bytes available per page divided by the index key row size, and this quotient divided *into* the number of data pages. You can do a similar computation to determine the pages at level 2. Once you reach a level with only one page, you can stop your computations because that level is the root.

Consider a table of 10,000 pages and 500,000 rows with a clustered key of a 5-byte fixed-length character. As you saw in Figure 8-5, the index key row size is 12 bytes, so we can fit 674 (8096 bytes available on a page / 12 bytes per row) rows on an index page. Since we need index rows to point to all 10,000 pages, we'll need 15 index pages (10000 / 674) at level 1. Now, all these index pages need pointers at level 2, and since one page can contain 15 index rows to point to these 15 pages, level 2 is the root. If our 10,000-page table also has a 5-byte fixed-length character nonclustered index, the leaf-level rows (level 0) will be 11 bytes long as they will each contain a clustered index key (5 bytes), a nonclustered index key (5 bytes) and 1 byte for the status bits. The leaf level will contain every single nonclustered key value along with the corresponding clustered key values. An index page can hold 736 index rows at this leaf level. We need 500,000 index rows, one for each row of data, so we need 680 leaf pages. If this nonclustered index is unique, the index rows at the higher levels will be 12 bytes each, so 674 index rows will fit per page, and we'll need two pages at level 1, with level 2 as the one root page.

So how big are these indexes compared to the table? For the clustered index, we had 16 index pages for a 10,000-page table, which is less than 1 percent of the size of the table. We frequently use 1 percent as a ballpark estimate for the space requirement of a clustered index, even though you can see that in this case it's an overly large estimate. On the other hand, our nonclustered index needed 683 pages for the 10,000-page table, which is about 6 percent additional space. For nonclustered indexes, it is much harder to give a good ballpark figure. Nonclustered index keys are frequently much larger, or even composite, so it's not unusual to have key sizes of over 100n bytes. In that case, we'd need a lot more leaf level pages, and the total nonclustered index size could be 30 or 40 percent of the size of the table. Sometime we can see nonclustered indexes that are as big or bigger than the table itself. Once you have two or three nonclustered indexes, you need to double the space to support these indexes. Remember that SQL Server allows you to have up to 249 nonclustered indexes! Disk space is cheap, but is it that cheap? You still need to plan your indexes carefully.

Actual vs. Estimated Size

The actual space used for tables or indexes, as opposed to the ballpark estimates I just described, is stored in the *sysindexes* table. A column called *rowcnt* contains the number of data rows in the table. A column called *dpages* contains, for a clustered index or a heap, the number of data pages used; for all other indexes, *dpages* contains the number of index pages used. For LOB data (*text*, *ntext*, or *image*), with an *indid* value of 255, *dpages* is 0. *sysindexes* also has a column called *reserved*, which, for a clustered index or a heap contains the number of data pages reserved; for a nonclustered index, it contains the number of pages reserved for the index itself; and for LOB data, *reserved* is a count of the pages allocated.

Finally, there is a column in *sysindexes* called *used*. For clustered indexes or heaps, it contains the total number of pages used for all index and table data. For LOB data, *used* is the number of LOB pages. For nonclustered indexes, *used* is the count of index pages. Because no specific value is just the number of nonleaf, or nondata, pages in a clustered index, you must do

additional computations to determine the total number of index pages for a table, not including the data.

The stored procedure *sp_spaceused* examines these values and reports the total size used by a table. Keep in mind that these values in *sysindexes* are not updated every time a table or index is modified. In particular, immediately after you create a table and after some bulk operations, *sp_spaceused* might not accurately reflect the total space because the values in *sysindexes* are not accurate.

You can force SQL Server to update the *sysindexes* data in two ways. The simplest way to force the space used values to be updated is to ask for this when you execute the *sp_spaceused* procedure. The procedure takes two optional parameters: the first is called *@objname* and is a table name in the current database, and the second is called *@updateusage*, which can have a value of TRUE or FALSE. If you specify TRUE, the data in *sysindexes* will be updated. The default is FALSE.

Forcing SQL Server to update the space used data for a table called *charge*:

```
exec sp_spaceused charge, true
```

Here's the result:

```
name  rows  reserved  data  index_size  unused
-----
charge 100000 10328 KB  5384 KB  4696 KB  248 KB
```

Remember that after the first 8 pages are allocated to a table, SQL Server grants all future allocations in units of 8 pages, called *extents*. The sum of all the allocations is the value of *reserved*. It might be that only a few of the 8 pages in some of the extents actually have data in them and are counted as used, so that the *reserved* value is frequently higher than the *used* value. The *data* value is computed by looking at the *data* column in *sysindexes* for the clustered index or heap and adding any LOB pages. The *index_size* value is computed by taking the *used* value from *sysindexes* for the clustered index or heap, which includes all index pages for all indexes. Because this value includes the number of data pages, which are at the leaf level of the clustered index, we must subtract the number of data pages from used pages to get the total number of index pages. The procedure *sp_spaceused* does all of its space reporting in KB, which we can divide by 8 to get the number of pages. The *unused* value is then the leftover amount after we subtract the *data* value and the *index_size* value from the *reserved* value.

You can also use the DBCC UPDATEUSAGE command to update the space information for every table and index in *sysindexes*. Or, by supplying additional arguments, you can have the values updated for only a single table, view, or index. In addition, DBCC UPDATEUSAGE gives you a report of the changes it made. You might want to limit the command to only one table because the command works by actually counting all the rows and all the pages in each table it's interested in. For a huge database, this can take a lot of time. In fact, *sp_spaceused* with the *@updateusage* argument set to TRUE actually calls DBCC UPDATEUSAGE and suppresses the report of changes made.

The following command updates the space used information for every table in the current database:

```
DBCC UPDATEUSAGE (0)
```

Here's some of the output I received when I ran this in the *Northwind* database:

DBCC UPDATEUSAGE: sysindexes row updated for table 'syscolumns' (index ID 2):

USED pages: Changed from (2) to (5) pages.

RSVD pages: Changed from (2) to (5) pages.

DBCC UPDATEUSAGE: sysindexes row updated for table 'Orders' (index ID 2):

USED pages: Changed from (2) to (5) pages.

RSVD pages: Changed from (2) to (5) pages.

DBCC UPDATEUSAGE: sysindexes row updated for table 'Orders' (index ID 3):

USED pages: Changed from (2) to (5) pages.

RSVD pages: Changed from (2) to (5) pages.

DBCC UPDATEUSAGE: sysindexes row updated for table 'Order Details' (index ID 3):

USED pages: Changed from (2) to (6) pages.

RSVD pages: Changed from (2) to (6) pages.

You can use this command to update both the *used* and *reserved* values in *sysindexes*. It's great to have a way to see how much space is being used by a particular table or index, but what if we're getting ready to build a database and we want to know how much space our data and indexes will need? For planning purposes, it would be nice to know this information ahead of time.

If you've already created your tables and indexes, even if they're empty, SQL Server can get column and datatype information from the *syscolumns* system table, and it should be able to see any indexes you've defined in the *sysindexes* table. There shouldn't be a need for you to type all this information into a spreadsheet. The CD also includes a set of stored procedures you can use to calculate estimated space requirements when the table and indexes have already been built. The main procedure is called *sp_EstTableSize*, and it requires only two parameters: the table name and the anticipated number of rows. The procedure calculates the storage requirements for the table and all indexes by extracting information from the *sysindexes*, *syscolumns*, and *systypes* tables. The result is only an estimate when you have variable-length fields. The procedure has no way of knowing whether a variable-length field will be completely filled, half full, or mostly empty in every row, so it assumes that variable-length columns will be filled to the maximum size. If you know that this won't be the case with your data, you can create a second table that more closely matches the expected data. For example, if you have a *varchar(1000)* field that you must set to the maximum because a few rare entries might need it but 99 percent of your rows will use only about 100 bytes, you can create a second table with a *varchar(100)* field and run *sp_EstTableSize* on that table.

Managing an Index

SQL Server maintains your indexes automatically. As you add new rows, it automatically inserts them into the correct position in a table with a clustered index, and it adds new leaf-level rows to your nonclustered indexes that will point to the new rows. When you remove rows, SQL Server automatically deletes the corresponding leaf-level rows from your nonclustered indexes. Here you'll see some specific examples of the changes that take place within an index as data modification operations take place.

Types of Fragmentation

In general, you need to do very little in terms of index maintenance. However, indexes can become fragmented. The fragmentation can be of two types. Internal fragmentation occurs when space is available within your index pages—that is, when the indexes are not making the most

efficient use of space. External fragmentation occurs when the logical order of pages does not match the physical order, or when the extents belonging to a table are not contiguous.

Internal fragmentation means that the index is taking up more space than it needs to. Scanning the entire table involves more read operations than if no free space were available on your pages. However, internal fragmentation is sometimes desirable—in fact, you can request internal fragmentation by specifying a low fillfactor value when you create an index. Having room on a page means that there is space to insert more rows without having to split a page. Splitting is a relatively expensive operation and can lead to external fragmentation because when a page is split, a new page must be linked into the indexes page chain, and usually the new page is not contiguous to the page being split.

External fragmentation is truly bad only when SQL Server is doing an ordered scan of all or part of a table or index. If you're seeking individual rows through an index, it doesn't matter where those rows are physically located—SQL Server can find them easily. If SQL Server is doing an unordered scan, it can use the IAM pages to determine which extents need to be fetched, and the IAM pages list the extents in disk order, so the fetching can be very efficient. Only if the pages need to be fetched in logical order, according to their index key values, do you need to follow the page chain. If the pages are heavily fragmented, this operation is more expensive than if there were no fragmentation.

Detecting Fragmentation

You can use the DBCC SHOWCONTIG command to report on the fragmentation of an index. Here's the syntax:

```
DBCC SHOWCONTIG
[ ( { table_name | table_id | view_name | view_id }
  [ , index_name | index_id ]
)
]
[ WITH { ALL_INDEXES
      | FAST [ , ALL_INDEXES ]
      | TABLERESULTS [ , { ALL_INDEXES } ]
      [ , { FAST | ALL_LEVELS } ]
}
]
```

You can ask to see the information for all the indexes on a table or view or just one index, and you can specify the object and index by name or by ID number. Alternatively, you can use the ALL_INDEXES option, which provides an individual report for each index on the object, regardless of whether a specific index is specified.

Here's some sample output from running a basic DBCC SHOWCONTIG on the *order_details* table in the *Northwind* database:

```
DBCC SHOWCONTIG scanning 'Order Details' table...
Table: 'Order Details' (325576198); index ID: 1, database ID: 6
TABLE level scan performed.
- Pages Scanned.....: 9
- Extents Scanned.....: 6
- Extent Switches.....: 5
- Avg. Pages per Extent.....: 1.5
- Scan Density [Best Count:Actual Count].....: 33.33% [2:6]
```

- Logical Scan Fragmentation: 0.00%
- Extent Scan Fragmentation: 16.67%
- Avg. Bytes Free per Page.....: 673.2
- Avg. Page Density (full).....: 91.68%

By default, DBCC SHOWCONTIG scans the page chain at the leaf level of the specified index and keeps track of the following values:

- Average number of bytes free on each page (*Avg. Bytes Free per Page*)
- Number of pages accessed (*Pages scanned*)
- Number of extents accessed (*Extents scanned*)
- Number of times a page had a lower page number than the previous page in the scan (This value for *Out of order pages* is not displayed, but it is used for additional computations.)
- Number of times a page in the scan was on a different extent than the previous page in the scan (*Extent Switches*)

SQL Server also keeps track of all the extents that have been accessed, and then it determines how many gaps are in the used extents. An extent is identified by the page number of its first page. So, if extents 8, 16, 24, 32, and 40 make up an index, there are no gaps. If the extents are 8, 16, 24, and 40, there is one gap. The value in DBCC SHOWCONTIG's output called *Extent Scan Fragmentation* is computed by dividing the number of gaps by the number of extents, so in this example the *Extent Scan Fragmentation* is $\frac{1}{4}$, or 25 percent. A table using extents 8, 24, 40, and 56 has 3 gaps, and its *Extent Scan Fragmentation* is $\frac{3}{4}$, or 75 percent. The maximum number of gaps is the number of extents - 1, so *Extent Scan Fragmentation* can never be 100 percent.

The value in DBCC SHOWCONTIG's output called *Logical Scan Fragmentation* is computed by dividing the number of *Out Of Order Pages* by the number of pages in the table. This value is meaningless in a heap.

You can use either the *Extent Scan Fragmentation* value or the *Logical Scan Fragmentation* value to determine the general level of fragmentation in a table. The lower the value, the less fragmentation there is. Alternatively, you can use the value called *Scan Density*, which is computed by dividing the optimum number of extent switches by the actual number of extent switches. A high value means that there is little fragmentation. *Scan Density* is not valid if the table spans multiple files, so all in all it is less useful than the other values.

You can use DBCC SHOWCONTIG to have the report returned in a format that can be easily inserted into a table. If you have a table with the appropriate columns and datatypes, you can execute the following:

```
INSERT INTO CONTIG_TABLE
EXEC ('dbcc showcontig ([order details])
with all_indexes, tableresults')
```

Here's a subset of the results:

ObjectName	IndexId	Level	Pages	Rows
-----	-----	-----	-----	-----
Order Details	1	0	9	2155
Order Details	2	0	4	2155
Order Details	3	0	4	2155

Many more columns are returned if you specify `TABLERESULTS`. In addition to the values that the nontabular output returns, you also get the values listed in Table 8-3.

Table 4 Additional columns returned when `TABLERESULTS` is specified.

Column	Meaning
<i>ObjectName</i>	Name of the table or view processed.
<i>ObjectId</i>	ID of the object processed.
<i>IndexName</i>	Name of the index processed (NULL for a heap).
<i>IndexId</i>	ID of the index (0 for a heap).
<i>Level</i>	Level of the index. Level 0 is the leaf (or data) level of the index. The level number increases moving up the tree toward the index root. The level is 0 for a heap. By default, only level 0 is reported. If you specify <code>ALL_LEVELS</code> along with <code>TABLERESULTS</code> , you get a row for each level of the index.
<i>Pages</i>	Number of pages comprising that level of the index or the entire heap.
<i>Rows</i>	Number of data or index records at that level of the index. For a heap, this is the number of data records in the entire heap.
<i>MinimumRecordSize</i>	Minimum record size in that level of the index or the entire heap.
<i>MaximumRecordSize</i>	Maximum record size in that level of the index or the entire heap.
<i>AverageRecordSize</i>	Average record size in that level of the index or the entire heap.
<i>ForwardedRecords</i>	Number of forwarded records in that level of the index or the entire heap.
<i>Extents</i>	Number of extents in that level of the index or the entire heap.

One last option available to `DBCC SHOWCONTIG` is the `FAST` option. The command takes less time to run when you specify this option, as you might imagine, because it gathers only data that is available from the IAM pages and the nonleaf levels of the index, and it returns only these values:

- Pages Scanned
- Extent Switches
- Scan Density
- Logical Scan Fragmentation

Since the level above the leaf has pointers to every page, SQL Server can determine all the page numbers and determine the *Logical Scan Fragmentation*. In fact, it can also use the IAM pages to determine the *Extent Scan Fragmentation*. However, the purpose of the `FAST` option is to determine whether a table would benefit from online defragmenting, and since online defragmenting cannot change the *Extent Scan Fragmentation*, there is little benefit in reporting it.

Removing Fragmentation

Several methods are available for removing fragmentation from an index. First, you can rebuild the index and have SQL Server allocate all new contiguous pages for you. You can do this by using a simple `DROP INDEX` and `CREATE INDEX`, but I've already discussed some reasons why this is not optimal. In particular, if the index supports a constraint, you can't use the `DROP INDEX` command. Alternatively, you can use `DBCC DBREINDEX`, which can rebuild all the indexes on a table in a single operation, or you can use the `drop_existing` clause along with `CREATE INDEX`.

The drawback of these methods is that the table is unavailable while the index is being rebuilt. If you are rebuilding only nonclustered indexes, there is a shared lock on the table, which means that no modifications can be made, but other processes can `SELECT` from the table. Of course, they cannot take advantage of the index you're rebuilding, so the query might not perform as well

as it should. If you're rebuilding the clustered index, SQL Server takes an exclusive lock, and no access is allowed at all.

SQL Server 2000 allows you to defragment an index without completely rebuilding it. In this release, DBCC INDEXDEFRAG reorders the leaf level pages into physical order as well as logical order, but using only the pages that are already allocated to the leaf level. It basically does an in-place ordering, similar to a sorting technique called bubble-sort. This can reduce the logical fragmentation to 0 to 2 percent, which means that an ordered scan through the leaf level will be much faster. In addition, DBCC INDEXDEFRAG compacts the pages of an index, based on the original fillfactor, which is stored in the *sysindexes* table. This doesn't mean that the pages always end up with the original fillfactor, but SQL Server uses that as a goal. The process does try to leave at least enough space for one average-size row after the defragmentation takes place. In addition, if a lock cannot be obtained on a page during the compaction phase of DBCC INDEXDEFRAG, SQL Server skips the page and doesn't go back to it. Any empty pages created as a result of this compaction are removed.

```
DBCC INDEXDEFRAG(0, 'charge', 1)
```

Here's the output:

Pages Scanned	Pages Moved	Pages Removed
-----	-----	-----
673	668	12

The algorithm SQL Server uses for DBCC INDEXDEFRAG finds the next physical page in a file belonging to the leaf level and the next logical page in the leaf level with which to swap it. It finds the next physical page by scanning the IAM pages for that index. The single-page allocations are not included in this release. Pages on different files are handled separately. The algorithm finds the next logical page by scanning the leaf level of the index. After each page move, all locks and latches are dropped and the key of the last page moved is saved. The next iteration of the algorithm uses the key to find the next logical page. This allows other users to update the table and index while DBCC INDEXDEFRAG is running.

Suppose the leaf level of an index consists of these pages, in this order:

```
47 22 83 32 12 90 64
```

The first step is to find the first physical page, which is 12, and the first logical page, which is 47. These pages are then swapped, using a temporary buffer as a holding area. After the first swap, the leaf level looks like this:

```
12 22 83 32 47 90 64
```

The next physical page is 22, which is the same as the next logical page, so no work is done. The next physical page is 32, which is swapped with the next logical page, 83, to look like this:

```
12 22 32 83 47 90 64
```

After the next swap of 47 with 83, the leaf level looks like this:

```
12 22 32 47 83 90 64
```

The 64 is swapped with 83:

```
12 22 32 47 64 90 83
```

And finally, 83 and 90 are swapped:

12 22 32 47 64 83 90

Keep in mind that DBCC INDEXDEFRAG uses only pages that are already part of the index leaf level. No new pages are allocated. In addition, the defragmenting can take quite a while on a large table. You get a report every five minutes on the estimated percentage completed.

Using an Index

Hopefully, you're aware of at least some of the places that indexes can be useful in your SQL Server applications. I'll list a few of the situations in which you can benefit greatly from indexes

Looking for Rows

The most straightforward use of an index is to help SQL Server find one or more rows in a table that satisfy a certain condition. For example, if you're looking for all the customers who live in a particular state (for example, your WHERE clause is WHERE state = 'WI'), you can use an index on the *state* column to find those rows more quickly. SQL Server can traverse the index from the root, comparing index key values to 'WI', to determine whether 'WI' exists in the table and then find the data associated with that value.

Joining

A typical join tries to find all the rows in one table that match rows in another table. Of course, you can have joins that aren't looking for exact matching, but you're looking for some sort of relationship between tables, and equality is by far the most common relationship. A query plan for a join frequently starts with one of the tables, finds the rows that match the search conditions, and then uses the join key in the qualifying rows to find matches in the other table. An index on the join column in the second table can be used to quickly find the rows that match.

Sorting

A clustered index stores the data logically in sorted order. The data pages are linked together in order of the clustering keys. If you have a query to ORDER BY the clustered keys or by the first column of a composite clustered key, SQL Server does not have to perform a sort operation to return the data in sorted order.

If you have a nonclustered index on the column you're sorting by, the sort keys themselves are in order at the leaf level. For example, consider the following query:

```
SELECT firstname, lastname, state, zipcode
FROM customers
ORDER BY zipcode
```

The nonclustered index will have all the zip codes in sorted order. However, we want the data pages themselves to be accessed to find the *firstname* and *lastname* values associated with the *zipcode*. The query optimizer will decide if it's faster to traverse the nonclustered index leaf level

and from there access each data row or to just perform a sort on the data. If you're more concerned with getting the first few rows of data as soon as possible and less concerned with the overall time to return all the results, you can force SQL Server to use the nonclustered index with the FAST hint.

Inverse Indexes

SQL Server allows you to sort in either ascending or descending order, and if you have a clustered index on the sort column, SQL Server can use that index and avoid the actual sorting. The pages at the leaf level of the clustered index are doubly linked, so SQL Server can use a clustered index on *lastname* to satisfy this query

```
SELECT * FROM customers ORDER BY lastname DESC
```

as easily as it can use the same index for this query:

```
SELECT * FROM customers ORDER BY lastname
```

SQL Server 2000 allows you to create descending indexes. Why would you need them if an ascending index can be scanned in reverse order? The real benefit is when you want to sort with a combination of ascending and descending order. For example, take a look at this query:

```
SELECT * FROM employees  
ORDER BY lastname, salary DESC
```

The default sort order is ascending, so the query wants the names ordered alphabetically by last name, and within duplicate last names, the rows should be sorted with the highest salary first and lowest salary last. The only kind of index that can help SQL Server avoid actually sorting the data is one with the *lastname* and *salary* columns sorted in opposite orders, such as this one:

```
CREATE CLUSTERED INDEX name_salary_idx  
ON employees (lastname, salary DESC)
```

If you execute *sp_help* or *sp_helpindex* on a table, SQL Server will indicate whether the index is descending by placing a minus (-) after the index key. Here's a subset of some *sp_helpindex* output:

index_name	index_description	index_keys
-----	-----	-----
lastname_salary_idx	nonclustered located on PRIMARY	LastName, Salary(-)

Grouping

One way that SQL Server can perform a GROUP BY operation is by first sorting the data by the grouping column. For example, if you want to find out how many customers live in each state, you can write a query with a GROUP BY state clause. A clustered index on state will have all the rows with the same value for state in logical sequence, so the sorting and grouping operations will be very fast.

Maintaining Uniqueness

Creating a unique index (or defining a PRIMARY KEY or UNIQUE constraint that builds a unique index) is by far the most efficient method of guaranteeing that no duplicate values are entered into a column. By traversing an index tree to determine where a new row should go, SQL Server can detect within a few page reads that a row already has that value. Unlike all the other uses of indexes described in this section, using unique indexes to maintain uniqueness isn't just one option among others. Although SQL Server might not always traverse a particular unique index to determine where to try to insert a new row, SQL Server will always use the existence of a unique index to verify whether a new set of data is acceptable.

Summary

We now know how SQL Server indexes organize the data on disk and help you access your data more quickly than if no indexes existed. Indexes are organized as B-trees, which means that you will always traverse through the same number of index levels when you traverse from the root to any leaf page. To use an index to find a single row of data, SQL Server never has to read more pages than there are levels in an appropriate index.

You also learned about all the options available when you create an index, how to determine the amount of space an index takes up, and how to predict the size of an index that doesn't have any data yet.

Indexes can become fragmented in SQL Server 2000, but the performance penalty for fragmentation is usually much less than in earlier versions of SQL Server. When you want to defragment your indexes, you have several methods to choose from, one of which allows the index to continue to be used by other operations, even while the defragmentation operation is going on.

Finally, you learned about some of the situations in which you can get the greatest performance gains by having the appropriate indexes on your tables.

Views

Introduction

You can use a view to save almost any SELECT statement as a separate database object. This SELECT statement enables you to create a result set that you can use just as you would any table. In a sense, you can think of a view as a "virtual" table. Views do not actually contain data—they simply consist of SELECT statements for extracting data from the actual tables in your database.

The tables on which you base a view are referred to as *base tables*. You can use a view to create a subset of a base table by selecting only some of its columns, or you can use a view to display columns from multiple base tables by using a join statement.

Why Use Views ?

One of the best advantages of views is that you can give your server's users permissions only to the view itself and not the underlying data. Thus, a view provides you with additional security. You can also use views to enable users to see some but not all columns in a table—so if a table contains a column with sensitive information, you can use a view to restrict users from seeing that column. For example, if you have an employee table that contains employee names, addresses, and salaries, you can create a view to enable users to see the employee names and addresses, but not salaries.

You can also use views as a way to hide a complex database design. If you have normalized the design of your database such that data is spread out over multiple tables, it can be difficult for users to learn how to retrieve data from those tables. By using views, you can relieve users from having to learn how to write SQL statements to join the tables.

Creating a View

You create a view by using the CREATE VIEW Transact-SQL statement. You can include a total of 1,024 columns in a view. You cannot combine the CREATE VIEW with other SQL statements in the same batch. If you want to use other statements (such as USE *database*) with the CREATE VIEW statement, you must follow those statements with the GO keyword.

Use the following syntax to create a view:

```
USE database
GO
CREATE VIEW view_name AS
SELECT column_list
FROM table_name
```

Replace *view_name* with the name you want to assign to the view. You should come up with a naming convention for your views that makes it easier for you to differentiate between tables and views. For example, you might try using "view" as part of all of your view names. Replace *column_list* with the list of columns you want to include in the view, and *table_name* with the name of the table on which you want to base the view.

If you want to create a view that consists only of each customer's name and phone number, use the following syntax:

```
USE movies
GO
```

```
CREATE VIEW dbo.CustView AS
SELECT lname, fname, phone
FROM customer
```

You can optionally specify a list of column names so that SQL Server will use these names for the columns in the view instead of the column names from the table in the SELECT portion of the statement. For example, in the following query, the (lname, fname) clause assigns these names to the columns in the view instead of the names au_lname, au_fname:

```
USE pubs
GO
CREATE VIEW dbo.PracticeView
(lname, fname)
AS
SELECT au_lname, au_fname
FROM authors
```

Restrictions

You cannot include the ORDER BY, COMPUTE, or COMPUTE BY clauses in the SELECT statement you use to create a view. In addition, you cannot use the SELECT INTO keywords. Your view cannot refer to temporary tables. For example, the following SQL statement is invalid:

```
CREATE VIEW dbo.TestView AS
SELECT col1, col2
FROM #temp_table
```

Permissions

If your users have permissions to the database in which you create the view, they will inherit permissions to the view itself. However, if your users do not inherit permissions to the view, you must assign them permissions or they will not be able to access the view. You do not have to give users permissions to the base tables on which you create a view—you just have to give users permissions to the view itself provided you are both the owner of the table and the view.

Ownership

The views that you create depend on the base tables (or other views). SQL Server refers to objects that depend on other objects as *dependent*. Objects can have either the same or different owners. If the same owner owns both the view and the table, that owner (typically you) needs only to assign users permissions to the view. Likewise, when users access your view, SQL Server need check users permissions only for that view.

If you (or another user with sufficient permissions) create a view based on a table for which you are not the owner, SQL Server considers the ownership chain to be "broken." Each object's owner can change users' permissions; thus, SQL Server must check users' permissions for the

view and all objects on which the view depends. Checking users' permissions for each object hurts your server's performance. Microsoft recommends that you do not break the ownership chain (meaning create views with different owners from the base tables) in order to avoid degrading the performance of your server.

To avoid breaking the ownership chain, you should explicitly specify the owner of the view when you create it. You should typically make the database owner (dbo) user the owner of all views along with all of the other objects in a database.

You make the dbo user the owner of a view by using the following syntax:

```
CREATE VIEW dbo.view_name AS
SELECT column_list
FROM table_name
```

Nested views

SQL Server enables you to create a view based on another view (this is also called a *nested view*). However, nested views can be much more difficult to troubleshoot because you must search through multiple view definitions to find a problem. For this reason, Microsoft recommends that you create separate views instead.

Creating a View based on Joined Tables

You can create a view based on joined tables by using a table join as part of your SELECT statement. You can use the following syntax to create a view based on a table join:

```
CREATE VIEW view_name
(column_list)
AS
SELECT columns
FROM table1 JOIN table2
ON join_condition
```

If you want to create a view that contains the title of each book in the pubs database, along with the author's royalty percentage for that book, you could use the following query:

```
USE pubs
GO
CREATE VIEW dbo.TitleRoyaltyView
AS
SELECT t.title, r.royalty
FROM titles AS t JOIN roysched AS r
ON t.title_id JOIN r.title_id
```

You should base views only on inner joins, not outer joins. While SQL Server enables you to specify an outer join in the SELECT statement, you will get unpredictable results.

SQL Server frequently returns null values for the inner table in the outer join.

Steps to create views from joined tables.....

Step 1:

The screenshot shows the SQL Server Query Analyzer interface. The main window displays the following SQL code:

```
USE movies
GO
CREATE VIEW dbo.R_MovieView
AS
SELECT title,category_num
FROM movie
WHERE rating='R'
```

Below the code, a message states: "The command(s) completed successfully." At the bottom, it shows "Query batch completed." and "Exec time: 0:00".

On the right side of the window, there is a vertical navigation pane with links for Step 1, Step 2, Step 3, Step 4, Step 5, and Step 6. Step 1 is currently selected.

A callout box titled "Creating Views from Joined Tables" is overlaid on the bottom right of the screenshot. It contains the following text:

Use the suggested query shown to create a view of title and category number of movies with an 'R' rating named R_MovieView.

Execute the query.

Step 2:

SQL Server Query Analyzer - [Query - User1.movies.DOMAIN\User1 - (untitl...]

File Edit View Query Window Help

DB: movies

```
USE movies
GO
CREATE VIEW dbo.R_MovieView
AS
SELECT title,category_num
FROM movie
WHERE rating='R'
SELECT *
FROM R_MovieView
```

title	category_num
Alien	3
Aliens	
All Quiet on the Western Front	
Amadeus	
American Beauty	

Results

Query batch completed. Exec time: 0:00

Creating Views from Joined Tables
Enter the SELECT statement to display the view.
Highlight the query and click on the Execute Query button.

Step 3:

SQL Server Query Analyzer - [Query - User1.movies.DOMAIN\User1 - (untitl...]

File Edit View Query Window Help

DB: movies

```
CREATE VIEW dbo.MovieCategoryView
AS
SELECT m.movie_num,m.title,c.description
FROM movie AS m JOIN category AS c
ON m.category_num=c.category_num
```

The command(s) completed successfully.

Results

Query batch completed. Exec time: 0:00

Creating Views from Joined Tables
Use the suggested query shown to create a view from joined tables containing the movie_num, title, and category description columns named MovieCategoryView.
Execute the query.

Step 4:

The screenshot shows the SQL Server Query Analyzer interface. The query editor contains the following SQL code:

```
CREATE VIEW dbo.MovieCategoryView
AS
SELECT m.movie_num,m.title,c.description
FROM movie AS m JOIN category AS c
ON m.category_num=c.category_num
SELECT DISTINCT*
FROM MovieCategoryView
ORDER BY title
```

The results pane displays the following data:

movie_num	title	descripti
147	12 Angry Men	Drama
170	2001: A Space Odyssey	Science F
185	African Queen The	
180	Alien	
192	Aliens	
215	All Quiet on the Western	

The status bar at the bottom indicates "Query batch completed." and "Exec time: 0:00".

Creating Views from Joined Tables
Enter the SELECT statement to verify the view and display the results in order by title, each title displayed only once.
Highlight and execute the query.

Step 1
Step 2
Step 3
Step 4
Step 5
Step 6

Step 5:

The screenshot shows the SQL Server Query Analyzer interface. The query editor contains the following SQL code:

```
CREATE VIEW dbo.RentalsView
AS
SELECT c.fname,c.lname,r.invoice_num,r.rental_date
FROM customer AS c JOIN rental AS r
ON c.cust_num=r.cust_num
```

The results pane displays the message: "The command(s) completed successfully."

The status bar at the bottom indicates "Query batch completed." and "Exec time: 0:00".

Creating Views from Joined Tables
Use the suggested query shown to create a view from joined tables of customer's first name, last name, invoice number, and rental date named RentalsView.
Execute the query.

Step 1
Step 2
Step 3
Step 4
Step 5
Step 6

Step 6:

The screenshot shows the SQL Server Query Analyzer interface. The main window displays the following SQL code:

```
CREATE VIEW dbo.RentalsView
AS
SELECT c.fname,c.lname,r.invoice_num,r.rental_date
FROM customer AS c JOIN rental AS r
ON c.cust_num=r.cust_num
SELECT fname+' '+lname AS Name, invoice_num,rental date
FROM RentalsView
ORDER BY lname, fname
```

The results pane shows the following data:

Name	invoice_num	rental_date
DonAlfred	630	1999-02-2
DonAlfred	631	1999-06-3
DonAlfred		
RhondaArdoin		
DougAtene		
DougAtene		

At the bottom of the window, a status bar indicates "Query batch completed." and "Exec time: 0:00".

Creating Views from Joined Tables
Use the SELECT statement to verify the view. List first and last names in a single column and sort by customer name.
Highlight and execute the query.

Step 1
Step 2
Step 3
Step 4
Step 5
Step 6

Displaying view definitions

You can use the following system views to display information about a database's views.

System view	Based on system table	Enables you to view
information_schema.tables	sysobjects	View names
information_schema.view_table_usage	sysdepends	Base object names
information_schema.views	syscomments	View definition
information_schema.view_column_usage	syscolumns	Columns defined in a view

To view a list of views defined in a database, you can use the following syntax:

```
SELECT *  
FROM information_schema.tables  
WHERE table_type = 'view'
```

To view the SELECT statement that makes up a view, use the `sp_helptext` stored procedure. Use the following syntax:

```
sp_helptext view_name
```

Preventing users from displaying view definitions

SQL Server stores a view's definition in the `syscomments` table; however, you should not delete the definition from this table as a technique for hiding the view definition. Instead, you should use encryption. (Although you can delete the definition of the view and it will still work, Microsoft recommends that you not delete it from `syscomments` to avoid problems when you upgrade to future versions of SQL Server.)

You can optionally add the `WITH ENCRYPTION` operator to prevent users from reading a view's definition, as follows:

```
CREATE VIEW view_name  
WITH ENCRYPTION  
AS  
select_statement
```

Modifying a view

You can alter a view by either dropping and re-creating it, or by using the `ALTER VIEW` statement. If you drop a view, you must re-create any permissions assignments when you re-create the view. In contrast, if you change a view by using the `ALTER VIEW` statement, the view

retains whatever permissions you had assigned to your users. You can use the following syntax to change an existing view:

```
ALTER VIEW view_name  
(column_list)  
AS  
select_statement
```

If you created the view with the WITH ENCRYPTION operator, you must include that option in the ALTER VIEW statement.

Dropping a view

You drop a view by using the DROP VIEW statement. When you delete a view, SQL Server automatically deletes the view definition and any permissions you have assigned to users for it. If you delete a table that is referenced by a view, SQL Server does not automatically drop the view. Thus, you must manually drop the view. You can use the sp_depends stored procedure to determine if a table has any dependent views by using the following syntax:

```
sp_depends object_name
```

You must be the owner of a view to delete it. However, if you are a member of the sysadmins server role or the database owner database role, you can drop a view that is owned by another user by specifying the owner's name in the DROP VIEW statement.

Use the following syntax to delete a view:

```
DROP VIEW [owner.]view_name
```

Using views to work with data

You can insert, update, and delete rows from a table by using a view. Note that views do not contain the actual data in a table—instead, views are simply windows to the data in the table. Note: If you have configured any of the columns in the tables on which the view is based to not permit nulls, and these columns are not contained in the view, you will not be able to insert rows into the table. Depending on the update statement, you might not be able to change the table either. You cannot modify the data in more than one table through a view. If a view is based on joined tables, you can modify the data in only one of the joined tables, not both. If you want to modify the data in both tables on which a view is based, you will need to write separate statements for modifying the data in each table.

Because views are essentially windows to your tables, you cannot insert, update, or delete rows if your statements will violate data integrity. For example, the rental and rental_detail tables in the movies database are linked together in a primary key to foreign key relationship based on the invoice_num column. Thus, you cannot change a value in the invoice_num column in either table, nor can you change it through a view.

Use the following syntax to insert data into a table by using a view:

```
INSERT INTO view_name  
VALUES(value_list)
```

Replace the *value_list* with a list of values you want to insert into the columns contained in the view.

Use the following syntax to update data through a view:

```
UPDATE view_name
SET column_name = value
WHERE condition
```

Likewise, you can use the following syntax to delete rows through a view:

```
DELETE FROM view_name
WHERE condition
```

1. What are some advantages of using views?

Views help provide additional security as you can give your server's users permissions only to the view itself and not the underlying data. With views you can restrict users from seeing sensitive information. Views can also be used as a way to hide a complex database design.

2. You would like to prevent anyone from reading the statement you used to build a view. What should you do?

You can prevent users from displaying a view definition by encrypting it. You encrypt a view definition by adding the WITH ENCRYPTION clause after the CREATE VIEW statement as follows:

```
CREATE VIEW view_name
WITH ENCRYPTION
AS
SELECT statement
```

Programming with Transact-SQL

Microsoft Transact-SQL allows you to use standard SQL as a programming language to write logic that can execute within the database engine. This extremely powerful capability is one of the keys to the success and growth of Microsoft SQL Server. Transact-SQL simplifies application development and reduces the amount of conversation necessary between the client application and the server by allowing more code to be processed at the server.

The SELECT statement, which is the foundation for effective use of Transact-SQL. Before reading any further, be sure you have a good understanding of the concepts presented in that chapter. These extensions allow you to easily write complex routines entirely in SQL, and they provide the basis for the additional power of Transact-SQL over standard SQL. Later in this chapter, you'll see some extensive examples that sometimes use Transact-SQL in nonintuitive ways, but it will be far from exhaustive coverage of what is possible using the Transact-SQL language.

Transact-SQL as a Programming Language

Is Transact-SQL really a programming language? On the one hand, Microsoft doesn't claim that Transact-SQL is an alternative to C, C++, Microsoft Visual Basic, COBOL, Fortran, or such 4GL development environments as Sybase PowerBuilder or Borland Inprise Delphi. You'd be hard-pressed to think of a substantial application that you could write entirely in Transact-SQL, and every one of those other languages or tools exists to do exactly that. Also, Transact-SQL offers no user interface and no file or device I/O, and the programming constructs are simple and limited.

On the other hand, you can argue that Transact-SQL is a specialized language that is best used in addition to one of those other languages or tools. It allows SQL Server to be programmed to execute complex tasks such as declaring and setting variables, branching, looping, and error checking, without the need for code to be written in another language. You can write reusable routines that you subsequently invoke and pass variables to. You can introduce bugs if you make an error in logic or syntax. Your Transact-SQL code can quickly get so complicated that you'll be glad you can use the same debugger to debug Transact-SQL routines that you use for C, C++, and Java development.

With Transact-SQL routines, conditional logic executes within the SQL Server engine and even within an executing SQL statement. This can greatly improve performance—the alternative would be message passing between the client process and the server. In today's increasingly networked world, reducing round-trip conversations is a key to developing and deploying efficient applications. Some might claim that the emergence of the Internet makes client/server computing irrelevant. But in fact, the typically slow network conditions found in Internet applications make client/server computing *more important*—operations should be written so that the client/server conversation is minimized. The best way to do this is to use the programming aspects of Transact-SQL to let whole routines execute remotely, without the need for intermediate processing at the client. The corresponding performance gains are phenomenal. In addition, by having SQL Server, rather than the application, handle the database logic, applications can often better insulate themselves from change. An application can execute a procedure simply by calling it and passing some parameters, and then the database structure or the procedure can change radically while the application remains entirely unaffected. As long as the inputs and outputs of the procedure are unchanged, the application is shielded from underlying changes to the database. The ability to encapsulate the database routines can result in a more efficient development environment for large projects in which the application programmers are often distinct from the database experts.

Programming at Multiple Levels

In the early days of database management systems, a single computer handled all processing. This was what we now call the *one-tier model*. The advent of client/server computing introduced the classic *two-tier model*: a server receiving requests from and providing data to a separate client. The newest approach, the *three-tier model*, places an intermediate layer between the database server and the client application. Figure 24 shows the two-tier client/server model, and Figure 25 shows the three-tier model, which some programmers find preferable for some applications.

Figure 24 Two Tier Data Model figure 25 Three tier Data Model

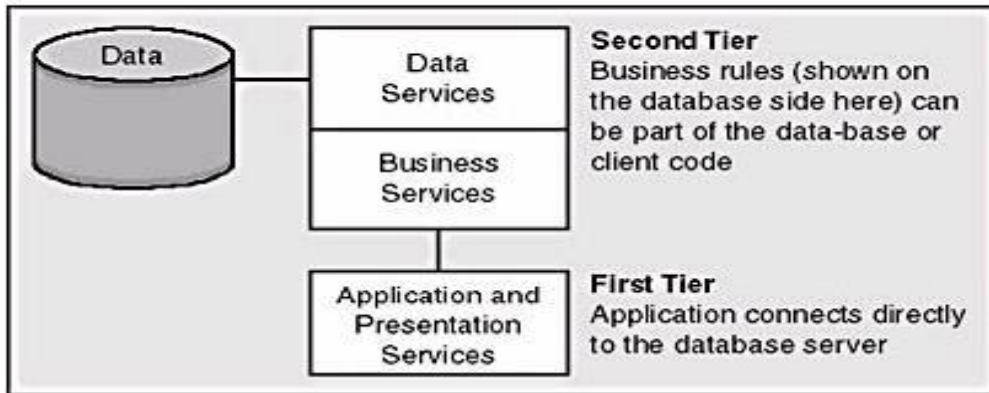
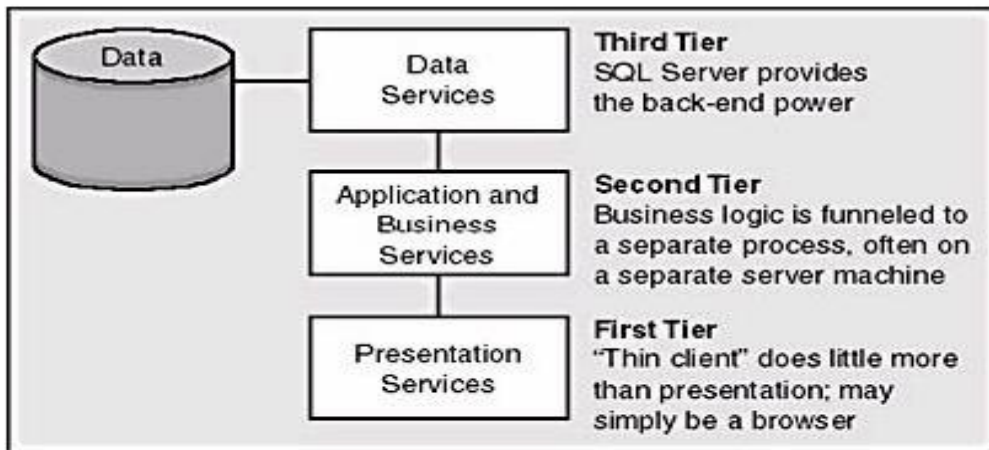


Figure 10-1. The two-tier model.



The two-tier model has a client (first tier) and a server (second tier); the client handles application and presentation logic, and the server handles data services and business services. The two-tier model uses the so-called *fat client*—the client does a large amount of application processing locally. Many solutions are deployed using this topology; certainly the lion's share of today's non-Internet client/server solutions are two-tier.

In the three-tier model, a *thin client* handles mostly presentation tasks. Supporters of the three-tier approach point out that this model allows the client computer to be less powerful; allows a

variety of client operating environments (obviously the case with Internet browsers); and reduces the complexity of installing, configuring, and maintaining software at the client. The client connects to an application server that handles the application process. The application server then connects to the database server, handling data services. The application server is typically a more powerful machine than the client, and it handles all the issues of application code maintenance, configuration, operations, and so on.

Presumably, far fewer application servers exist than clients; therefore, server costs and issues should be reduced in the three-tier model. (The three tiers are logical concepts; the actual number of computers involved might be more or less than three.) Typically, you're part of a three-tier model when you use the Internet with a browser and access a Web page residing on an Internet server that performs database work. An example of a three-tier application is a mail-order company's Web page that checks current availability by using ODBC calls to access its inventory in SQL Server.

Both two-tier and three-tier solutions have advantages and disadvantages, and each can be viewed as a potential solution to a given problem. The specifics of the problem and the constraints imposed by the environment (including costs, hardware, and expertise of both the development staff and the user community) should drive the solution's overall design. However, some people claim that a three-tier solution means that such capabilities as stored procedures, rules, constraints, or triggers are no longer important or shouldn't be used. In a three-tier solution, the middle tier doing application logic is still a client from the perspective of the database services (that is, SQL Server). Consequently, for that tier, it's still preferable that remote work be done close to the data. If the data can be accessed by servers other than the application server, integrity checking must be done at the database server as well as in the application. Otherwise, you've left a gaping hole through which the integrity checks in the application can be circumvented.

The client sends the commands and processes the results, and it makes no difference to SQL Server whether an application server process does this or whether the process that the end user is running directly does it. The benefits of the programmable server provide much more efficient network use and better abstraction of data services from the application and application server.

Three-Tier Solution That Works

Perhaps the single most successful three-tier solution available today is R/3 from SAP. The financial and operations capabilities that R/3 provides use a three-tier model, with a terminal-like interface connected to the R/3 application server that then accesses Microsoft SQL Server. The R/3 application server wisely and heavily uses stored procedures that provide both performance gains and development efficiency.

Transact-SQL Programming Constructs

Transact-SQL extends standard SQL by adding many useful programming constructs. These constructs will look familiar to developers experienced with C/C++, Basic, Visual Basic, Fortran, Java, Pascal, and similar languages. The Transact-SQL extensions are wonderful additions to the original SQL standard. Before the extensions were available, requests to the database were always simple, single statements. Any conditional logic had to be provided by the calling application. With a slow network or via the Internet, this would be disastrous, requiring many trips across the network—more than are needed by simply letting conditions be evaluated at the server and allowing subsequent branching of execution.

Variables

Without variables, a programming language wouldn't be particularly useful. Transact-SQL is no exception. Variables that you declare are *local variables*, which have scope and visibility only within the batch or stored procedure in which you declare them. (The next chapter discusses the details of batches and stored procedures.) In code, the single @ character designates a local variable. SQL Server 2000 introduces the ability to store and retrieve session context information, which allows information to be available throughout a connection; in any stored procedure, function, or trigger; and across batches. You can use this session context information like a session variable, with a scope broader than that of SQL Server's local variables but not as broad as that of a true global variable.

Transact-SQL has no true global variables that allow information to be shared across connections. Older documentation refers to certain parameterless system functions as *global variables* because they're designated by @@, which is similar to the designation for local variables.

User-declared global variables would be a nice enhancement that we might see in a future release. Until then, temporary tables provide a decent alternative for sharing values among connections.

Local Variables

You declare local variables at the beginning of a batch or a stored procedure. You can subsequently assign values to local variables with the SELECT statement or the SET statement, using an equal (=) operator. The values you assign using the SET statement can be constants, other variables, or expressions. When you assign values to a variable using a SELECT statement, you typically select the values from a column in a table. The syntax for declaring and using the variables is identical, regardless of the method you use to assign the values to the variable. In the following example, a variable is declared, assigned it a value using the SET statement, and then used that variable in a WHERE clause:

```
DECLARE @limit money
SET @limit = $10
SELECT * FROM titles
WHERE price <= @limit
```

However, you typically use SELECT when the values to be assigned are found in a column of a table. A SELECT statement used to assign values to one or more variables is called an *assignment SELECT*. You can't combine the functionality of the assignment SELECT and a "regular" SELECT in the same statement. That is, if a SELECT statement is used to assign values to variables, it can't also return values to the client as a result set. In the following simple example, two variables are declared, assigned values to them from the *roysched* table, and then we would select their values as a result set:

```
DECLARE @min_range int, @hi_range int           -- Variables declared
SELECT @min_range=MIN(lorange),
       @hi_range=MAX(hirange) FROM roysched    -- Variables assigned
SELECT @min_range, @hi_range                  -- Values of variables
       returned as result
```

A single DECLARE statement can declare multiple variables. When you use a SELECT statement for assigning values, you can assign more than one value at a time. When you use SET to assign values to variables, you must use a separate SET statement for each variable.

The following SET statement returns an error:

```
SET @min_range = 0, @hi_range = 100
```

Be careful when you assign variables by selecting a value from the database—you want to ensure that the SELECT statement will return only one row. It's perfectly legal to assign a value to a variable in a SELECT statement that will return multiple rows, but the variable's value might not be what you expect: no error message is returned, and the variable has the value of the last row returned.

Suppose that the following stor_name values exist in the stores table of the pubs database:

```
stor_name
-----
Eric the Read Books
Barnum's
News & Brews
Doc-U-Mat: Quality Laundry and Books
Fricative Bookshop
Bookbeat
```

The following assignment runs without error but is probably a mistake:

```
DECLARE @stor_name varchar(30)
SELECT @stor_name=stor_name FROM stores
```

The resulting value of @stor_name is the last row, Bookbeat. But consider the order returned, without an ORDER BY clause, as a chance occurrence. Assigning a variable to a SELECT statement that returns more than one row usually isn't intended, and it's probably a bug introduced by the developer. To avoid this situation, you should qualify the SELECT statement with an appropriate WHERE clause to limit the result set to the one row that meets your criteria. Alternatively, you can use an aggregate function, such as MAX, MIN, or SUM, to limit the number of rows returned. Then you can select only the value you want. If you want the assignment to be to only one row of a SELECT statement that might return many rows (and it's not important which row that is), you should at least use SELECT TOP 1 to avoid the effort of gathering many rows, with all but one row thrown out. (In this case, the first row would be returned, not the last. You could use ORDER BY to explicitly indicate which row should be first.) You might also consider checking the value of @@ROWCOUNT immediately after the assignment and, if it's greater than 1, branch off to an error routine.

Also be aware that if the SELECT statement assigning a value to a variable doesn't return any rows, nothing will be assigned. The variable will keep whatever value it had before the assignment SELECT statement was run. For example, suppose we use the same variable twice to find the first names of authors with a particular last name:

```
DECLARE @firstname varchar(20)
SELECT @firstname = au_fname
FROM authors
WHERE au_lname = 'Greene'
SELECT @firstname -- Return the first name as a result
SELECT @firstname = au_fname
FROM authors
WHERE au_lname = 'Ben-Gan'
SELECT @firstname -- Return the first name as a result
```

If you run the previous code, you'll see the same first name returned both times because no authors with the last name of Ben-Gan exist, at least not in the pubs database!

You can also assign a value to a variable in an UPDATE statement. This approach can be useful and more efficient than using separate statements to store the old value of a column and assign it a new value. The following UPDATE will reduce the price of one book by 20 percent, and save the original price in a variable:

```
DECLARE @old_price money

UPDATE title
SET @old_price = price = price * 0.8
WHERE title_id = 'PC2091'
```

A variable can be assigned a value from a subquery, and in some cases it might look like there's no difference between using a subquery and a straight SELECT. For example, the following two batches return exactly the same result set:

```
-- First batch using a straight SELECT:
DECLARE @firstname varchar(20)
SELECT @firstname = au_fname
FROM authors
WHERE au_lname = 'Greene'
SELECT @firstname
```

-- Second batch using a subquery:

```
DECLARE @firstname varchar(20)
SELECT @firstname = ( SELECT au_fname
FROM authors
WHERE au_lname = 'Greene')
SELECT @firstname
```

However, if we leave off the WHERE clause, these two batches will behave very differently. As we saw earlier in the example selecting from the *stores* table, the first batch will leave the variable with the value that the last row supplied:

```
DECLARE @firstname varchar(20)
SELECT @firstname = au_fname
FROM authors
SELECT @firstname
```

RESULT:

Akiko

This one row is returned because the assignment to the variable is happening for every row that the query would return. With no WHERE clause, every row in the table is returned, and for every row, the first name value is assigned to the variable, overriding whatever value was assigned by the previous row returned. We are left with the value from the last row because there are no more values to override that one.

However, when a subquery is used and the WHERE clause is ignored, the result is very different:

```
DECLARE @firstname varchar(20)
SELECT @firstname = (SELECT au_fname
FROM authors)
```

```
SELECT @firstname
```

RESULT:

Server: Msg 512, Level 16, State 1, Line 0
Subquery returned more than 1 value. This is not permitted when the subquery follows =, !=, <, <=, >, >= or when the subquery is used as an expression.

NULL

In this case, the query returns an error. When you use a subquery, the entire subquery is processed before any assignment to the variable is made. The subquery's result set contains 23 rows, and then all 23 rows are used in the assignment to the variable. Since we can assign only a single value to a variable, the error message is generated.

Session Variables

SQL Server 2000 allows you to store and retrieve session-level context information by directly querying the *sysprocesses* table in the *master* database. Normally it is recommended that you never access the system tables directly, but for this release, direct access of *sysprocesses* is the only way to retrieve this information. A construct is supplied for setting this information, but you have nothing other than direct system table access for retrieving it.

The *sysprocesses* table in the *master* database in SQL Server 2000 has a 128-byte field called *context_info* of type *binary*, which means you have to manipulate the data using hexadecimal values. You can use the entire column to store a single value or, if you're comfortable with manipulating binary values, you can use different groups of bytes for different purposes. The following statements store the price of one particular book in the *context_info* column:

```
DECLARE @cost money  
SELECT @cost = price FROM titles  
WHERE title_id = 'bu1032'  
SET context_info @cost
```

If we stored this value in a local variable, it would have the scope of only a single query batch and we wouldn't be able to come back after executing several more batches and check the value. But storing it in the *sysprocesses* table means that the value will be available as long as my connection is active. The *sysprocesses* table has one row for each active process in SQL Server, and each row is assigned a unique Server Process ID (spid) value. A system function @@spid returns the process ID for the current connection. Using @@spid, we can find the row in *sysprocesses* that contains the *context_info* value.

Using *SET context_info*, we can assign a money value to the binary column. If you check the documentation for the CONVERT function, you'll see that money is implicitly convertible to binary. For convenience, in Figure 26 we have reproduced the chart that shows which datatypes are implicitly and explicitly convertible.

If, instead, we had tried to assign a character value to CONTEXT_INFO, we would have received an error message because the chart in Figure 10-3 tells us that conversion from character to binary must be done explicitly. The SET CONTEXT_INFO statement stores the assigned value in the fewest number of bytes, starting with the first byte. So we can use the SUBSTRING function to extract a value from the *context_info* column, and since the *money* datatype is stored in 8 bytes, we need to get the first 8 bytes of the column. we can then convert those first 8 bytes to *money*:

```
SELECT convert(money, substring(context_info, 1, 8))
FROM master..sysprocesses
WHERE spid = @@spid
```

If we hadn't taken only the first 8 bytes of the *context_info* field, as shown below, and tried to convert that directly to *money*, SQL Server would have assumed that the *money* value was in the last 8 bytes of the 128-byte field, and return a 0.

```
SELECT convert(money, context_info)
FROM master..sysprocesses
WHERE spid = @@spid
```

figure 26 Conversion table from SQL server Documentation

To:\nFrom:	binary	varbinary	char	varchar	nchar	nvarchar	datetime	smalldatetime	decimal	numeric	float	real	int(INT4)	smallint(INT2)	tinyint(INT1)	money	smallmoney	bit	timestamp	uniqueidentifier	image	ntext	text
binary	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
varbinary	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
char	●	●	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
varchar	●	●	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
nchar	●	●	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
nvarchar	●	●	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
datetime	●	●	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
smalldatetime	●	●	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
decimal	○	○	○	○	○	○	○	○	★	★	○	○	○	○	○	○	○	○	○	○	○	○	○
numeric	○	○	○	○	○	○	○	○	★	★	○	○	○	○	○	○	○	○	○	○	○	○	○
float	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
real	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
int(INT4)	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
smallint(INT2)	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
tinyint(INT1)	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
money	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
smallmoney	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
bit	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
timestamp	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
uniqueidentifier	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
image	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
ntext	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
text	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○

- Explicit conversion
- Implicit conversion
- Conversion not allowed
- ★ Requires CONVERT when loss of precision or scale will occur

Since the *context_info* column is 128 bytes long, there's lot of room to work with. We can store another price in the second set of 8 bytes if we are careful. We need to take the first 8 bytes and concatenate them to the second price value after converting it to hexadecimal. Concatenation works for hexadecimal values the same way as it does for character strings, and the operator is the plus (+):

```
DECLARE @cost money, @context binary(128)
SELECT @cost = price FROM titles
WHERE title_id = 'ps2091'
SELECT @context =
    convert(binary(8),context_info) + convert(binary(8), @cost)
```

```
FROM master..sysprocesses
WHERE spid = @@spid
SET context_info @context
```

Now the second 8 bytes contains the second price, and if we use the SUBSTRING function appropriately, we can extract either price value from the context_info column:

```
SELECT First_price = convert(money, substring(context_info, 1, 8))
FROM master..sysprocesses
WHERE spid = @@spid
```

```
SELECT Second_price = convert(money, substring(context_info, 9, 8))
FROM master..sysprocesses
WHERE spid = @@spid
```

Control-of-Flow Tools

Like any programming language worth its salt, Transact-SQL provides a decent set of control-of-flow tools. (True, Transact-SQL has only a handful of tools—perhaps not as many as you'd like—but they're enough to make it dramatically more powerful than standard SQL.) The control-of-flow tools include conditional logic (IF...ELSE and CASE), loops (only WHILE, but it comes with CONTINUE and BREAK options), unconditional branching (GOTO), and the ability to return a status value to a calling routine (RETURN). Table 10-1 presents a quick summary of these control-of-flow constructs.

Table 5 Control-of-flow constructs in Transact-SQL

<i>Construct</i>	<i>Description</i>
BEGIN...END	Defines a statement block. Using BEGIN...END allows a group of statements to be executed. Typically, BEGIN immediately follows IF, ELSE, or WHILE. (Otherwise, only the next statement will be executed.) For C programmers, BEGIN...END is similar to using a {..} block.
GOTO label	Continues processing at the statement following the specified label.
IF...ELSE	Defines conditional and, optionally, alternate execution when a condition is false.
RETURN [n]	Exits unconditionally. Typically used in a stored procedure or trigger (although it can also be used in a batch). Optionally, a whole number <i>n</i> (positive or negative) can be set as the return status, which can be assigned to a variable when you execute the stored procedure.
WAITFOR	Sets a time for statement execution. The time can be a delay interval (up to 24 hours) or a specific time of day. The time can be supplied as a literal or with a variable.
WHILE	The basic looping construct for SQL Server. Repeats a statement (or block) while a specific condition is true.
...BREAK	Exits the innermost WHILE loop.
...CONTINUE	Restarts a WHILE loop.

CASE

The CASE expression is enormously powerful. Although CASE is part of the ANSI SQL-92 specification, it's not required for ANSI compliance certification, and few database products other than Microsoft SQL Server have implemented it. If you have experience with an SQL database system other than Microsoft SQL Server, chances are you haven't used CASE. If that's the case (pun intended), you should get familiar with it now. It will be time well spent. CASE was added in version 6, and once you get used to using it, you'll wonder how SQL programmers ever managed without it.

CASE is a conceptually simpler way to do IF-ELSE IF-ELSE IF-ELSE IF-ELSE_type operations. It's roughly equivalent to a *switch* statement in C. However, CASE is far more than shorthand for IF—in fact, it's not really even that. CASE is an expression, not a control-of-flow keyword, which means that it can be used *only* inside other statements. You can use CASE in a SELECT statement in the SELECT list, in a GROUP BY clause, or in an ORDER BY clause. You can use it in the SET clause of an UPDATE statement. You can include CASE in the values list for an INSERT statement. You can also use CASE in a WHERE clause in any SELECT, UPDATE, or DELETE statement. Anywhere that Transact-SQL expects an expression, you can use CASE to determine the value of that expression.

Here's a simple example of CASE. Suppose we want to classify books in the *pubs* database by price. We want to segregate them as low-priced, moderately priced, or expensive. And we'd like to do this with a single SELECT statement and not use UNION. Without CASE, we can't do it. With CASE, it's a snap:

```
SELECT
title,
price,
'classification'=CASE
    WHEN price < 10.00 THEN 'Low Priced'
    WHEN price BETWEEN 10.00 AND 20.00 THEN 'Moderately Priced'
    WHEN price > 20.00 THEN 'Expensive'
    ELSE 'Unknown'
END
FROM titles
```

A NULL price won't fit into any of the category buckets.

Here are the abbreviated results.

title	price	classification
-----	-----	-----
The Busy Executive's Database Guide	19.99	Moderately Priced
Cooking with Computers: Surreptitious Balance Sheets	11.95	Moderately Priced
You Can Combat Computer Stress!	2.99	Low Priced
Straight Talk About Computers	19.99	Moderately Priced
Silicon Valley Gastronomic Treats	19.99	Moderately Priced
The Gourmet Microwave	2.99	Low Priced
The Psychology of Computer Cooking But Is It User Friendly?	NULL 22.95	Unknown Expensive

If all the conditions after your WHEN clauses are equality comparisons against the same value, you can use an even simpler form so that you don't have to keep repeating the expression. Suppose we want to print the type of each book in the titles table in a slightly more user-friendly fashion—for example, Modern Cooking instead of mod_cook. We could use the following SELECT statement:

```
SELECT
title,
price,
'Type' = CASE
    WHEN type = 'mod_cook' THEN 'Modern Cooking'
    WHEN type = 'trad_cook' THEN 'Traditional Cooking'
    WHEN type = 'psychology' THEN 'Psychology'
    WHEN type = 'business' THEN 'Business'
    WHEN type = 'popular_comp' THEN 'Popular Computing'
    ELSE 'Not yet decided'
END
FROM titles
```

In this example, because every condition after WHEN is testing for equality with the value in the *type* column, we can use an even simpler form. You can think of this simplification as factoring out the *type* = from every WHEN clause, and you can place the *type* column at the top of the CASE expression:

```
SELECT
title,
price,
'Type' = CASE type
    WHEN 'mod_cook' THEN 'Modern Cooking'
    WHEN 'trad_cook' THEN 'Traditional Cooking'
    WHEN 'psychology' THEN 'Psychology'
    WHEN 'business' THEN 'Business'
    ELSE 'Not yet decided'
END
FROM titles
```

You can use CASE in some unusual places; because it's an expression, you can use it anywhere that an expression is legal. Using CASE in a view is a nice technique that can make your database more usable to others. Using CASE in an UPDATE statement can make the update easier. More important, CASE can allow you to make changes in a single pass of the data that would otherwise require you to do multiple UPDATE statements, each of which would have to scan the data.

Cousins of CASE

SQL Server provides three nice shorthand derivatives of CASE: COALESCE, NULLIF, and ISNULL. COALESCE and NULLIF are both part of the ANSI SQL-92 specification; they were added to version 6 at the same time that CASE was added. ISNULL is a longtime SQL Server function.

COALESCE This function is equivalent to a CASE expression that returns the first NOT NULL expression in a list of expressions.

COALESCE(*expression1*, *expression2*, ... *expressionN*)

If no non-null values are found, COALESCE returns NULL (which is what *expressionN* was, given that there were no non-null values). Written as a CASE expression, COALESCE looks like this:

```
CASE
  WHEN expression1 IS NOT NULL THEN expression1
  WHEN expression2 IS NOT NULL THEN expression2
  ELSE expressionN
END
```

NULLIF This function is equivalent to a CASE expression in which NULL is returned if *expression1* = *expression2*.

```
NULLIF(expression1, expression2)
```

If the expressions aren't equal, *expression1* is returned. NULLIF can be handy if you use dummy values instead of NULL, but you don't want those dummy values to skew the results produced by aggregate functions. Written using CASE, NULLIF looks like this:

```
CASE
  WHEN expression1=expression2 THEN NULL
  ELSE expression1
END
```

ISNULL This function is almost the mirror image of NULLIF:

```
ISNULL(expression1, expression2)
```

ISNULL allows you to easily substitute an alternative expression or value for an expression that's NULL. The use of ISNULL to substitute the string 'ALL' for a GROUPING NULL when using CUBE and to substitute a string of question marks ('????') for a NULL value. The results were more clear and intuitive to users. The functionality of ISNULL written instead using CASE looks like this:

```
CASE
  WHEN expression1 IS NULL THEN expression2
  ELSE expression1
END
```

The second argument to ISNULL (the value to be returned if the first argument is NULL) can be an expression or even a SELECT statement. Say, for example, that we want to query the *titles* table. If a title has NULL for *price*, we want to instead use the lowest price that exists for any book. Without CASE or ISNULL, this isn't an easy query to write. With ISNULL, it is easy:

```
SELECT title, pub_id, ISNULL(price, (SELECT MIN(price)
  FROM titles)) FROM titles
```

For comparison, here's an equivalent SELECT statement written with the longhand CASE formulation:

```
SELECT title, pub_id,
  CASE WHEN price IS NULL THEN (SELECT MIN(price) FROM titles)
  ELSE price
  END
FROM titles
```


PRINT

Transact-SQL, like other programming languages, provides printing capability through the PRINT and RAISERROR statements. I'll discuss RAISERROR momentarily, but first we'll take a look at the PRINT statement.

PRINT is the most basic way to display any character string of up to 8000 characters. You can display a literal character string or a variable of type *char*, *varchar*, *nchar*, *nvarchar*, *datetime*, or *smalldatetime*. You can concatenate strings and use functions that return string or date values. In addition, PRINT can print any expression or function that is implicitly convertible to a character datatype. (Refer to Figure 10-3 to determine which datatypes are implicitly convertible.)

It seems like every time you learn a new programming language, the first assignment is to write the "Hello World" program. Using Transact-SQL, it couldn't be easier:

```
PRINT 'Hello World'
```

Even before you saw PRINT, you could've produced the same output in your SQL Query Analyzer screen with the following:

```
SELECT 'Hello World'
```

So what's the difference between the two? The PRINT statement returns a message, nothing more.

The SELECT statement returns output like this:

```
-----  
Hello World
```

(1 row(s) affected)

The SELECT statement returns the string as a result set. That's why we get the (1 row(s) affected) message. When you use SQL Query Analyzer to submit your queries and view your results, you might not see much difference between the PRINT and the SELECT statements. But for other client tools (besides ISQL, OSQL, and SQL Query Analyzer), there could be a big difference. PRINT returns a message of severity 0, and you need to check the documentation for the client API that you're using to determine how messages are handled. For example, if you're writing an ODBC program, the results of a PRINT statement are handled by *SQLError*.

On the other hand, a SELECT statement returns a set of data, just like selecting prices from a table does. Your client interface might expect you to bind data to local (client-side) variables before you can use them in your client programs. Again, you'll have to check your documentation to know for sure. The important distinction to remember here is that PRINT returns a message and SELECT returns data.

PRINT is extremely simple, both in its use and in its capabilities. It's not nearly as powerful as, say, *printf* in C. You can't give positional parameters or control formatting. For example, you can't do this:

```
PRINT 'Today is %s', GETDATE()
```

Part of the problem here is that %s is meaningless. In addition, the built-in function GETDATE returns a value of type *datetime*, which can't be concatenated to the string constant. So, you can do this instead:

```
PRINT 'Today is ' + CONVERT(char(30), GETDATE())
```

RAISERROR

RAISERROR is similar to PRINT but can be much more powerful. RAISERROR also sends a message (not a result set) to the client; however, RAISERROR allows you to specify the specific error number and severity of the message. It also allows you to reference an error number, the text of which you can add to the *sysmessages* table. RAISERROR makes it easy for you to develop international versions of your software. Rather than using multiple versions of your SQL code when you want to change the language of the text, you need only one version. You can simply add the text of the message in multiple languages. The user will see the message in the language used by the particular connection.

Unlike PRINT, RAISERROR accepts *printf*-like parameter substitution as well as formatting control. RAISERROR can also allow messages to be logged to the Windows NT or Windows 2000 event service, making the messages visible to the Windows NT or Windows 2000 Event Viewer, and it allows SQL Server Agent alerts to be configured for these events. In many cases, RAISERROR is preferable to PRINT, and you might choose to use PRINT only for quick-and-dirty situations. For production-caliber code, RAISERROR is usually a better choice. Here's the syntax:

```
RAISERROR ({msg_id | msg_str}, severity, state[, argument1  
[, argumentn]])  
[WITH LOG][WITH NOWAIT]
```

After you call RAISERROR, the system function @@ERROR returns the value passed as *msg_id*. If no ID is passed, the error is raised with error number 50000, and @@ERROR is set to that number. Error numbers lower than 50000 are reserved for SQL Server use, so choose a higher number for your own messages.

You can also set the severity level; an informational message should be considered severity 0 (or severity 10; 0 and 10 are used interchangeably for a historical reason that's not important). Only someone with the system administrator (SA) role can raise an error with a severity of 19 or higher. Errors of severity 20 and higher are considered fatal, and the connection to the client is automatically terminated.

By default, if a batch includes a WAITFOR statement, results or messages are not returned to the client until the WAITFOR has completed. If you want the messages generated by your RAISERROR statement to be returned to the client immediately, even if the batch contains a subsequent WAITFOR, you can use the WITH NOWAIT option. You must use the WITH LOG option for all errors of severity 19 or higher. For errors of severity 20 or higher, the message text isn't returned to the client. Instead, it's sent to the SQL Server error log, and the error number, text, severity, and state are written to the operating system event log. A SQL Server system administrator can use the WITH LOG option with any error number or any severity level.

Errors written to the operating system event log from RAISERROR go to the application event log, with *MSSQLServer* as the source if the server is the default server, and *MSSQL\$InstanceName* as the source if your server is a named instance. These errors have event ID 17060. (Errors that SQL Server raises can have different event IDs, depending on which component they come from.) The type of message in the event log depends on the severity used in the RAISERROR statement. Messages with severity 14 and lower are recorded as *informational messages*, severity 15 messages are *warnings*, and severity 16 and higher messages are *errors*. Note that the severity used by the event log is the number that's passed to RAISERROR, regardless of the stated severity in *sysmessages*. If you look at the data section of the event log for a message raised from RAISERROR, or if you're reading from the event log in a program, you can choose to display the data as bytes or words.

You can use any number from 1 through 127 as the *state* parameter. You might pass a line number that tells where the error was raised. But in a user-defined error message, *state* has no real relevance to SQL Server. For internally generated error messages, the value of *state* is an indication of where in the code the error came from. It can sometimes be of use to Microsoft Product Support personnel to know the value of *state* that was returned with an error message.

There's a little-known way to cause `isql.exe` or `OSQL.EXE` (the command-line query tools) to terminate. Raising any error with *state* 127 causes `isql` or `OSQL` to immediately exit, with a return code equal to the error number. (You can determine the error number from a batch command file by inspecting the system `ERRORLEVEL` value.) You could write your application so that it does something similar—it's a simple change to the error-handling logic. This trick can be a useful way to terminate a set of scripts that would be run through `isql` or `OSQL`. You could get a similar result by raising a high-severity error to terminate the connection. But using a *state* of 127 is actually simpler, and no scary message will be written to the error log or event log for what is a planned, routine action.

You can add your own messages and text for a message in multiple languages by using the `sp_addmessage` stored procedure. By default, SQL Server uses U.S. English, but you can add languages. SQL Server distinguishes U.S. English from British English because of locale settings, such as currency. The text of messages is the same. The procedure `sp_helplanguage` shows you what languages are available on your SQL Server instance. If a language you are interested in doesn't appear in the list returned by `sp_helplanguage`, take a look at the script `instlang.sql`, which can be found in the `INSTALL` subdirectory of your SQL Server instance installation files. This script adds locale settings for additional languages but does not add the actual translated error messages. Suppose we want to rewrite "Hello World" to use `RAISERROR`, and we want the text of the message to be able to appear in both U.S. English and German (language ID of 1).

Here's how:

```
EXEC sp_addmessage 50001, 10, 'Hello World', @replace='REPLACE'  
-- New message 50001 for U.S. English  
EXEC sp_addmessage 50001, 10, 'Hallo Welt' , @lang='Deutsch',  
    @replace='REPLACE'  
- New message 50001 for German
```

When `RAISERROR` is executed, the text of the message becomes dependent on the `SET LANGUAGE` setting of the connection. If the connection does a `SET LANGUAGE Deutsch`, the text returned for `RAISERROR (50001, 10, 1)` will be *Hallo Welt*. If the text for a language doesn't exist, the U.S. English text (by default) is returned. For this reason, the U.S. English message must be added before the text for another language is added. If no entry exists, even in U.S. English, error 2758 results:

```
Msg 2758, Level 16, State 1
```

```
RAISERROR could not locate entry for error n in Sysmessages.
```

We could easily enhance the message to say whom the "Hello" is from by providing a parameter marker and passing it when calling `RAISERROR`. (For illustration, we'll use some `printf`-style formatting, `#6x`, to display the process number in hexadecimal format.)

```
EXEC sp_addmessage 50001, 10,  
    'Hello World, from: %s, process id: %#6x', @replace='REPLACE'  
When user kalend executes  
DECLARE @parm1 varchar(30), @parm2 int  
SELECT @parm1=USER_NAME(), @parm2=@@spid  
RAISERROR (50001, 15, -1, @parm1, @parm2)
```

this error is raised:

```
Msg 50001, Level 15, State 1  
Hello, World, from: kalend, process id: 0xc
```

FORMATMESSAGE

If you have created your own error messages with parameter markers, you might need to inspect the entire message with the parameters replaced by actual values. RAISERROR makes the message string available to the client but not to your SQL Server application. To construct a full message string from a parameterized message in the *sysmessages* table, you can use the function FORMATMESSAGE.

If we consider the last example from the preceding RAISERROR section, we can use FORMATMESSAGE to build the entire message string and save it in a local variable for further processing. Here's an example of the use of FORMATMESSAGE:

```
DECLARE @parm1 varchar(30), @parm2 int, @message varchar(100)
SELECT @parm1=USER_NAME(), @parm2=@@spid
SELECT @message = FORMATMESSAGE(50001, @parm1, @parm2)
PRINT 'The message is: ' + @message
```

This returns the following:

The message is: Hello World, from: kalend, process id: 0xc
Note that no error number or state information is returned.

Operators

Transact-SQL provides a large collection of operators for doing arithmetic, comparing values, doing bit operations, and concatenating strings. These operators are similar to those you might be familiar with in other programming languages. You can use Transact-SQL operators in any expression, including in the select list of a query, in the WHERE or HAVING clauses of queries, in UPDATE and IF statements, and in CASE expressions.

Arithmetic Operators

We won't go into much detail here because arithmetic operators are pretty standard stuff. If you need more information, please refer to the online documentation. Table 10-2 shows the arithmetic operators.

Table 6 Arithmetic operators in Transact-SQL

Symbol	Operation	Used with These Datatypes
+	Addition	<i>int, smallint, tinyint, bigint, numeric, decimal, float, real, money, smallmoney, datetime and smalldatetime</i>
-	Subtraction	<i>int, smallint, tinyint, bigint, numeric, decimal, float, real, money, smallmoney, datetime and smalldatetime</i>
*	Multiplication	<i>int, smallint, tinyint, bigint, numeric, decimal, float, real, money, and smallmoney</i>
/	Division	<i>int, smallint, tinyint, bigint, numeric, decimal, float, real, money, and smallmoney</i>
%	Modulo	<i>int, smallint, tinyint, and bigint</i>

As with other programming languages, in Transact-SQL you need to consider the datatypes you're using when you perform arithmetic operations. Otherwise, you might not get the results you expect.

For example, which of the following is correct?

1 / 2 = 0

1 / 2 = 0.5

If the underlying datatypes are both integers (including *bigint*, *tinyint* and *smallint*), the correct answer is 0. If one or both of the datatypes are *float* (including *real*) or *numeric/decimal*, the correct answer is 0.5 (or 0.50000, depending on the precision and scale of the datatype). When an operator combines two expressions of different datatypes, the datatype precedence rules specify which datatype is converted to the other. The datatype with the lower precedence is converted to the datatype with the higher precedence.

Here's the precedence order for SQL Server datatypes:

sql_variant (highest)
datetime
smalldatetime
float
real
decimal
money
smallmoney
bigint
int
smallint
tinyint
bit
ntext
text
image
timestamp (rowversion)
uniqueidentifier
nvarchar
nchar
varchar
char
varbinary
binary (lowest)

As you can see from the precedence list, an *int* multiplied by a *float* produces a *float*. If you need to use data of type *sql_variant* in an arithmetic operation, you must first convert the *sql_variant* data to its base datatype. You can explicitly convert to a given datatype by using the *CAST* function or its predecessor, *CONVERT*.

The addition (+) operator is also used to concatenate character strings. In this example, we want to concatenate last names, followed by a comma and a space, followed by first names from the *authors* table (in *pubs*), and then we want to return the result as a single column:

```
SELECT 'author'=au_lname + ', ' + au_fname FROM authors
```

Here's the abbreviated result:

```
author
-----
White, Johnson
Green, Marjorie
```

Carson, Cheryl
O'Leary, Michael

Bit Operators

Table 10-3 shows the SQL Server bit operators.

Table 7. SQL Server bit operators.

Symbol	Meaning
&	Bitwise AND (two operands)
	Bitwise OR (two operands)
^	Bitwise exclusive OR (two operands)
~	Bitwise NOT (one operand)

The operands for the two-operand bitwise operators can be any of the datatypes of the integer or binary string datatype categories (except for the *image* datatype), with the exception that both operands cannot be a type of binary string. So, one of the operands can be *binary* or *varbinary*, but they can't both be *binary* or *varbinary*. In addition, the right operand can be of the *bit* datatype only if the left operand is of type *bit*. Table 10-4 shows the supported operand datatypes.

Table 8. Datatypes supported for two-operand bitwise operators.

Left Operand	Right Operand
<i>Binary</i>	<i>int, smallint, tinyint</i> or <i>bigint</i>
<i>Bit</i>	<i>int, smallint, tinyint, bigint</i> or <i>bit</i>
<i>Int</i>	<i>int, smallint, tinyint, bigint, binary, or varbinary</i>
<i>Smallint</i>	<i>int, smallint, tinyint, bigint, binary, or varbinary</i>
<i>Tinyint</i>	<i>int, smallint, tinyint, bigint, binary, or varbinary</i>
<i>Bigint</i>	<i>int, smallint, tinyint, bigint, binary, or varbinary</i>
<i>Varbinary</i>	<i>int, smallint, tinyint, or bigint</i>

The single operand for the bitwise NOT operator must be one of the integer datatypes. Often, you must keep a lot of indicator-type values in a database, and bit operators make it easy to set up bit masks with a single column to do this. The SQL Server system tables use bit masks. For example, the *status* field in the *sysdatabases* table is a bit mask. If we wanted to see all databases marked as read-only, which is the 11th bit or decimal 1024 (2^{10}), we could use this query:

```
SELECT 'read only databases'=name FROM master..sysdatabases
WHERE status & 1024 > 0
```

This example is for illustrative purposes only. You shouldn't query the system catalogs directly. Sometimes the catalogs need to change between product releases, and if you query directly, your applications can break because of these required changes. Instead, you should use the provided system catalog stored procedures and object property functions, which return catalog information in a standard way. If the underlying catalogs change in subsequent releases, the property functions are also updated, insulating your application from unexpected changes. So, for this example, to see whether a particular database was marked read-only, we could use the DATABASEPROPERTY function:

```
SELECT DATABASEPROPERTY('mydb', 'IsReadOnly')
```

A return value of 1 would mean that *mydb* was set to read-only status.

Keeping such indicators as *bit* datatype columns can be a better approach than using an *int* as a bit mask. It's more straightforward—you don't always need to look up your bit-mask values for

each indicator. To write the "read only databases" query we just looked at, you'd need to check the documentation for the bit-mask value for "read only." And many developers, not to mention end users, aren't that comfortable using bit operators and are likely to use them incorrectly. From a storage perspective, *bit* columns don't require more space than equivalent bit-mask columns require. (Eight *bit* fields can share the same byte of storage.)

But you wouldn't typically want to do that with a "yes/no" indicator field anyway. If you have a huge number of indicator fields, using bit-mask columns of integers instead of *bit* columns might be a better alternative than dealing with hundreds of individual *bit* columns. However, there's nothing to stop you from having hundreds of *bit* columns. You'd have to have an awful lot of *bit* fields to actually exceed the column limit because a table in SQL Server can have up to 1024 columns. If you frequently add new indicators, using a bit mask on an integer might be better. To add a new indicator, you can make use of an unused bit of an integer status field rather than do an ALTER TABLE and add a new column (which you must do if you use a *bit* column). The case for using a *bit* column boils down to clarity.

Comparison Operators

Table 9 shows the SQL Server comparison operators.

Table 9. SQL Server comparison operators.

Symbol	Meaning
=	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<>	Not equal to (ANSI standard)
!=	Not equal to (not ANSI standard)
!>	Not greater than (not ANSI standard)
!<	Not less than (not ANSI standard)

Comparison operators are straightforward. Only a few related issues might be confusing to a novice SQL Server developer:

- When you deal with NULL, remember the issues with three-value logic and the truth tables. Also understand that NULL is an unknown.
- Comparisons of *char* and *varchar* data (and their Unicode counterparts) depend on the collation value associated with the column. Whether comparisons evaluate to TRUE depends on the sort order installed.
- Comparison of *sql_variant* values involves special rules that break down the SQL Server datatype hierarchy shown previously into datatype families, as shown in Table 10-6. The following script creates a table that I can use to illustrate some of the issues involved in comparing values of type *sql_variant*:

```
CREATE TABLE variant2
(a sql_variant, b sql_variant )
GO
INSERT INTO variant2 VALUES (CAST (111 as int), CAST(222 as money ))
GO
INSERT INTO variant2 VALUES (CAST (333 as int), CAST(444 as char(3) ))
GO
```

Here are the special rules governing comparison of *sql_variant* values:

- When *sql_variant* values of different base datatypes are compared and the base datatypes are in different datatype families, the value whose datatype family is higher in the hierarchy chart is considered the higher of the two values.

You can see this behavior by executing the following query:

```
SELECT *
FROM variant2
WHERE a > b
```

Result:

```
a      b
-----
333    444
```

The second row inserted had a value of base type *int* and a value of base type *char*. Even though you would normally think that 333 is not greater than 444 because 333 is of datatype *int* and the family *exact numeric*, it is considered higher than 444, of datatype *char*, and of the family *Unicode*.

- When *sql_variant* values of different base datatypes are compared and the base datatypes are in the same datatype family, the value whose base datatype is lower in the hierarchy chart is implicitly converted to the other datatype and the comparison is then made.

You can see this behavior by executing the following query:

```
SELECT *
FROM variant2
WHERE a < b
```

Result:

```
a      b
-----
222.0000
```

In this case, the two base types are *int* and *money*, which are in the same family, so 111 is considered less than 222.0000.

- When *sql_variant* values of the *char*, *varchar*, *nchar*, or *nvarchar* datatypes are compared, they are evaluated based on additional criteria, including the locale code ID (LCID), the LCID version, the comparison flags use for the column's collation, and the sort ID.

Table 10 Datatype families for use when comparing *sql_variant* values.

Datatype Hierarchy	Datatype Family
<i>sql_variant</i>	<i>sql_variant</i>
<i>datetime</i>	<i>datetime</i>
<i>smalldatetime</i>	<i>datetime</i>
<i>float</i>	<i>approximate numeric</i>
<i>real</i>	<i>approximate numeric</i>
<i>decimal</i>	<i>exact numeric</i>
<i>money</i>	<i>exact numeric</i>
<i>smallmoney</i>	<i>exact numeric</i>
<i>bigint</i>	<i>exact numeric</i>
<i>int</i>	<i>exact numeric</i>
<i>smallint</i>	<i>exact numeric</i>
<i>tinyint</i>	<i>exact numeric</i>
<i>bit</i>	<i>exact numeric</i>
<i>nvarchar</i>	<i>Unicode</i>
<i>nchar</i>	<i>Unicode</i>
<i>varchar</i>	Unicode

<i>char</i>	<i>Unicode</i>
<i>varbinary</i>	<i>binary</i>
<i>binary</i>	<i>binary</i>
<i>uniqueidentifier</i>	<i>uniqueidentifier</i>

Logical and Grouping Operators (Parentheses)

The three logical operators (AND, OR, and NOT) are vitally important in expressions, especially in the WHERE clause. You can use parentheses to group expressions and then apply logical operations to the group. In many cases, it's impossible to correctly formulate a query without using parentheses. In other cases, you might be able to avoid them, but your expression won't be very intuitive and the absence of parentheses could result in bugs. You could construct a convoluted expression using some combination of string concatenations or functions, bit operations, or arithmetic operations instead of using parentheses, but there's no good reason to do this.

Using parentheses can make your code clearer, less prone to bugs, and easier to maintain. Because there's no performance penalty beyond parsing, you should use parentheses liberally. Below is a query that you couldn't readily write without using parentheses. This query is simple, and in English the request is well understood. But you're also likely to get this query wrong in Transact-SQL if you're not careful, and it clearly illustrates just how easy it is to make a mistake. Find authors who do not live in either Salt Lake City, UT, or Oakland, CA.

Following are four examples that attempt to formulate this query. Three of them are wrong. Test yourself: which one is correctly stated? (As usual, multiple correct formulations are possible—not just the one presented here.)

EXAMPLE 1

```
SELECT au_lname, au_fname, city, state FROM authors
WHERE
city <> 'OAKLAND' AND state <> 'CA'
OR
city <> 'Salt Lake City' AND state <> 'UT'
```

EXAMPLE 2

```
SELECT au_lname, au_fname, city, state FROM authors
WHERE
(city <> 'OAKLAND' AND state <> 'CA')
OR
(city <> 'Salt Lake City' AND state <> 'UT')
```

EXAMPLE 3

```
SELECT au_lname, au_fname, city, state FROM authors
WHERE
(city <> 'OAKLAND' AND state <> 'CA')
AND
(city <> 'Salt Lake City' AND state <> 'UT')
```

EXAMPLE 4

```
SELECT au_lname, au_fname, city, state FROM authors
WHERE
NOT
(
(city='OAKLAND' AND state='CA')
OR
```

```
(city='Salt Lake City' AND state='UT')
)
```

you can see that only Example 4 operates correctly. This query would be much more difficult to write without some combination of parentheses, NOT, OR (including IN, a shorthand for OR), and AND. You can also use parentheses to change the order of precedence in mathematical and logical operations. These two statements return different results:

```
SELECT 1.0 + 3.0 / 4.0 -- Returns 1.75
SELECT (1.0 + 3.0) / 4.0 -- Returns 1.00
```

The order of precedence is similar to what you learned back in algebra class (except for the bit operators). Operators of the same level are evaluated left to right. You use parentheses to change precedence levels to suit your needs, when you're unsure, or when you simply want to make your code more readable. Groupings with parentheses are evaluated from the innermost grouping outward. Table 11 shows the order of precedence.

Table 11 Order of precedence in mathematical and logical operations, from highest to lowest.

Operation	Operators
Bitwise	NOT ~
Multiplication/division/modulo	* / %
Addition/subtraction	+ / -
Bitwise exclusive	OR ^
Bitwise AND	&
Bitwise OR	
Logical NOT	NOT
Logical AND	AND
Logical OR	OR

If you didn't pick the correct example, you can easily see the flaw in Examples 1, 2, and 3 by examining the output. Examples 1 and 2 both return every row. Every row is either not in CA or not in UT because a row can be in only one or the other. A row in CA isn't in UT, so the expression returns TRUE. Example 3 is too restrictive—for example, what about the rows in San Francisco, CA? The condition (*city* <> 'OAKLAND' AND *state* <> 'CA') would evaluate to (*TRUE and FALSE*) for San Francisco, CA, which, of course, is FALSE, so the row would be rejected when it should be selected, according to the desired semantics.

Scalar Functions

Aggregate functions, such as MAX, SUM, and COUNT operate on a set of values to produce a single aggregated value. In addition to aggregate functions, SQL Server provides *scalar functions*, which operate on a single value. *Scalar* is just a fancy term for *single value*. You can also think of the value in a single column of a single row as *scalar*. Scalar functions are enormously useful—the Swiss army knife of SQL Server. You've probably noticed that scalar functions have been used in several examples already—we'd all be lost without them.

You can use scalar functions in any legal expression, such as:

- In the select list
- In a WHERE clause, including one that defines a view
- Inside a CASE expression
- In a CHECK constraint
- In the VALUES clause of an INSERT statement
- In an UPDATE statement to the right of the equals sign in the SET clause

- With a variable assignment
- As a parameter inside another scalar function

SQL Server provides too many scalar functions to remember. The best you can do is to familiarize yourself with those that exist, and then, when you encounter a problem and a light bulb goes on in your head to remind you of a function that will help, you can look at the online documentation for details. Even with so many functions available, you'll probably find yourself wanting some functions that don't exist. You might have a specialized need that would benefit from a specific function. The following sections describe the scalar functions that are supplied with SQL Server 2000. We'll first look at the CAST function because it's especially important.

Conversion Functions

SQL Server provides three functions for converting datatypes: the generalized CAST function, the CONVERT function—which is analogous to CAST but has a slightly different syntax—and the more specialized STR function

The CAST function is a synonym for CONVERT and was added to the product to comply with ANSI-92 specifications. You'll see CONVERT used in older documentation and older code. CAST is possibly the most useful function in all of SQL Server. It allows you to change the datatype when you select a field, which can be essential for concatenating strings, joining columns that weren't initially envisioned as related, performing mathematical operations on columns that were defined as character but which actually contain numbers, and other similar operations. Like C, SQL is a fairly strongly typed language.

Some languages, such as Visual Basic or PL/1 (Programming Language 1), allow you to almost indiscriminately mix datatypes. If you mix datatypes in SQL, however, you'll often get an error. Some conversions are implicit, so, for example, you can add or join a *smallint* and an *int* without any such error; trying to do the same between an *int* and a *char*, however, produces an error. Without a way to convert datatypes in SQL, you would have to return the values back to the client application, convert them there, and then send them back to the server. You might need to create temporary tables just to hold the intermediate converted values. All of this would be cumbersome and inefficient.

Recognizing this inefficiency, the ANSI committee added the CAST operator to SQL-92. SQL Server's CAST—and the older CONVERT—provides a superset of ANSI CAST functionality. The CAST syntax is simple:

CAST (*original_expression* AS *desired_datatype*)

Suppose we want to concatenate the *job_id* (*smallint*) column of the *employee* table with the *lname* (*varchar*) column. Without CAST, this would be impossible (unless it was done in the application). With CAST, it's trivial:

```
SELECT lname + '-' + CAST(job_id AS varchar(2)) FROM employee
```

Here's the abbreviated output:

```
Accorti-13  
Afonso-14  
Ashworth-6  
Bennett-12  
Brown-7  
Chang-4  
Cramer-2  
Cruz-10  
Devon-3
```

Specifying a length shorter than the column or expression in your call to CAST is a useful way to truncate the column. You could use the SUBSTRING function for this with equal results, but you might find it more intuitive to use CAST. Specifying a length when you convert to *char*, *varchar*, *decimal*, or *numeric* isn't required, but it's recommended. Specifying a length shields you better from possible behavioral changes in newer releases of the product.

Although behavioral changes generally don't occur by design, it's difficult to ensure 100 percent consistency of subtle behavioral changes between releases. The development team strives for such consistency, but it's nearly impossible to ensure that no behavioral side effects result when new features and capabilities are added. There's little point in relying on the current default behavior when you can just as easily be explicit about what you expect.

You might want to avoid CAST when you convert *float* or *numeric/decimal* values to character strings if you expect to see a certain number of decimal places. CAST doesn't currently provide formatting capabilities for numbers. The older CONVERT function has some limited formatting capabilities, which we'll see shortly. Formatting floating-point numbers and such when converting to character strings is another area that's ripe for subtle behavior differences. So if you need to transform a floating-point number to a character string in which you expect a specific format, you can use the other conversion function, STR. STR is a specialized conversion function that always converts from a number (*float*, *numeric*, and so on) to a character datatype, but it allows you to explicitly specify the length and number of decimal places that should be formatted for the character string. Here's the syntax:

```
STR(float_expression, character_length, number_of_decimal_places)
```

Remember that *character_length* must include room for a decimal point and a negative sign if they might exist. Also be aware that STR rounds the value to the number of decimal places requested, while CAST simply truncates the value if the character length is smaller than the size required for full display. Here's an example of STR:

```
SELECT discounttype, 'Discount'=STR(discount, 7, 3) FROM discounts
```

And here's the output:

```
discounttype      Discount
-----
Initial Customer   10.500
Volume Discount    6.700
Customer Discount  5.000
```

You can think of the ASCII and CHAR functions as special type-conversion functions, but we'll discuss them later in this chapter along with string functions.

If all you're doing is converting an expression to a specific datatype, you can actually use either CAST or the older CONVERT. However, the syntax of CONVERT is slightly different and allows for a third, optional argument. Here's the syntax for CONVERT:

```
CONVERT (desired_datatype[(length)], original_expression
        [, style])
```

CONVERT has an optional third parameter, *style*. This parameter is most commonly used when you convert an expression of type *datetime* or *smalldatetime* to type *char* or *varchar*. You can also use it when you convert *float*, *real*, *money*, or *smallmoney* to a character datatype.

When you convert a *datetime* expression to a character datatype, if the *style* argument isn't specified, the conversion will format the date with the default SQL Server date format (for example, Oct 3 1999 2:25PM). By specifying a *style* type, you can format the output as you want. You can also specify a shorter length for the character buffer being converted to, in order to perhaps eliminate the time portion or for other reasons. Table 10-8 shows the various values you can use as the *style* argument.

Table 12 Values for the style argument of the CONVERT function when you convert a datetime expression to a character expression.

Style Number without Style Century (yy)	Number with Century (yyyy)	Output Type	Style
–	0 or 100	Default	mon dd yyyy hh:miAM (or PM)
1	101	USA	mm/dd/yyyy
2	102	ANSI	yyyy.mm.dd
3	103	British/ French	dd/mm/yyyy
4	104	German	dd.mm.yyyy
5	105	Italian	dd-mm-yyyy
6	106	–	dd mon yyyy
7	107	–	mon dd, yyyy
–	8 or 108	–	hh:mm:ss
–	9 or 109	Default + milliseconds (or PM)	mon dd yyyy hh:mi:ss: mmmAM
10	110	USA	mm-dd-yy
11	111	JAPAN	yy/mm/dd
12	112	ISO	yymmdd
–	13 or 113	Europe default + milliseconds	dd mon yyyy hh:mi:ss:mmm (24h)
14	114	–	hh:mi:ss:mmm (24h)
	20 or 120	ODBC canonical	yyyy-mm-dd hh:mi:ss(24h)
	21 or 121	ODBC canonical + milliseconds	yyyy-mm-dd hh:mi:ss.mmm(24h)

Although SQL Server uses style 0 as the default when converting a *datetime* value to a character string, SQL Query Analyzer (and OSQL) use style 121 when displaying a *datetime* value.

Since we just passed a change in the century, using two-character formats for the year could be a bug in your application just waiting to happen! Unless you have a compelling reason not to, you should always use the full year (*yyyy*) for both the input and output formatting of dates to prevent any ambiguity.

SQL Server has no problem dealing with the year 2000—in fact, the change in century isn't even a boundary condition in SQL Server's internal representation of dates. But if you represent a year by specifying only the last two digits, the inherent ambiguity might cause your application to make an incorrect assumption. When a date formatted that way is entered, SQL Server's default behavior is to interpret the year as 19yy if the value is greater than or equal to 50 and as 20yy if the value is less than 50. That might be OK now, but by the year 2050 you won't want a two-digit year of 50 to be interpreted as 1950 instead of 2050. (If you simply assume that your application will have been long since retired, think again. A lot of COBOL programmers in the 1960s didn't worry about the year 2000.)

SQL Server 2000 allows you to change the cutoff year that determines how a two-digit year is interpreted. A two-digit year that is less than or equal to the last two digits of the cutoff year is in the same century as the cutoff year. A two-digit year that is greater than the last two digits of the cutoff year is in the century that precedes the cutoff year. You can change the cutoff year by using the Properties dialog box for your SQL server in SQL Server Enterprise Manager and selecting the Server Settings tab. Or you can use the *sp_configure* stored procedure:

```
EXEC sp_configure 'two digit year cutoff', '2000'
```

In this example, since *two digit year cutoff* is 2000, all two-digit years other than 00 are interpreted as occurring in the 20th century. With the default *two digit year cutoff* value of 2049, the two-digit year 49 is interpreted as 2049 and the two-digit year 50 is interpreted as 1950. You should be aware that although SQL Server uses 2049 as the cutoff year for interpreting dates, OLE Automation objects use 2030. You can use the *two digit year cutoff* option to provide consistency in date values between SQL Server and client applications. However, to avoid ambiguity with dates, you should always use four-digit years in your data.

You don't need to worry about how changing the *two digit year cutoff* value will affect your existing data. If data is stored using the *datetime* or *smalldatetime* datatypes, the full four-digit year is part of the internal storage. The *two digit year cutoff* value controls only how SQL Server interprets date constants such as *10/12/99*.

Day First or Month First

The *two digit year cutoff* value determines how SQL Server interprets the *99* in *10/12/99*, but this date constant has another ambiguity. Are we talking about October 12 or December 10? The SET option DATEFORMAT determines whether SQL Server interprets a numeric three-part date as month followed by day, day followed by month, or even year followed by month. The default value of DATEFORMAT is controlled by the language you're using, but you can change it using the SET DATEFORMAT command. There are six possible values for DATEFORMAT, which should be self-explanatory: *mdy*, *dmy*, *ymd*, *ydm*, *dym*, and *myd*. Only the first three are commonly used. Note that DATEFORMAT controls only how SQL Server interprets date constants that are entered by you or your program. It does not control how date values are displayed. As we've mentioned, output format is controlled by the client or by having SQL Server convert the *datetime* value to a character string and using a style option.

Here's an example that shows how changing the DATEFORMAT value can change how input dates are interpreted:

```
SET DATEFORMAT mdy
SELECT 'FORMAT is mdy' = CONVERT(datetime, '7/4/2000')
SET DATEFORMAT dmy
SELECT 'FORMAT is dmy' = CONVERT(datetime, '7/4/2000')
```

RESULTS:

```
FORMAT is mdy
-----
2000-07-04 00:00:00.000
```

```
FORMAT is dmy
-----
2000-04-07 00:00:00.000
```

You might consider using ISO style when you insert dates. This format has the year followed by month and then the day, with no punctuation. This style is always recognizable, no matter what

the SQL Server default language and no matter what the DATEFORMAT setting. Here's the above conversion command with dates supplied in ISO format:

```
SET DATEFORMAT mdy
SELECT 'FORMAT is mdy' = CONVERT(datetime, '20000704')
SET DATEFORMAT dmy
SELECT 'FORMAT is mdy' = CONVERT(datetime, '20000704')
RESULTS:
FORMAT is mdy
-----
2000-07-04 00:00:00.000

FORMAT is mdy
-----
2000-07-04 00:00:00.000
```

Be sure to always put your dates in quotes. If we had left off the quotes, SQL Server would assume we meant the number 20,000,704 and would interpret that as the number of days after January 1, 1900. It would try to figure out the date that was that many days after the base date, which is out of the range of possible *datetime* values that SQL Server can store.

One other date format can override your default language or DATEFORMAT setting, but it behaves a little inconsistently. If you enter a date in all numeric format with the year first but you include punctuation (as in 1999.05.02), SQL Server will assume that the first part is the year even if your DATEFORMAT is *dmy*. SQL Server will still use your DATEFORMAT setting to determine the order of the month and the day values. So if your DATEFORMAT is *mdy*, `SELECT CONVERT(datetime, '1999.5.2')` will return `1999-05-02 00:00:00.000`, and if your DATEFORMAT is *dmy*, `SELECT CONVERT(datetime, '1999.5.2')` will return `1999-02-05 0:00:00.000` and `SELECT CONVERT(datetime, '1999.5.22')` will return an out-of-range error. This seems quite inconsistent. SQL Server is partially ignoring the DATEFORMAT, but not completely. We suggest that you avoid this format for inputting dates, and if you're going to use all numbers, leave out the punctuation. Instead of setting the DATEFORMAT value, you can use the *style* argument when you convert from a *character* string to a *datetime* value. You must be sure to enter the date in the format that the style requires, as indicated in Table 10-8 (shown earlier). These two statements return different values:

```
SELECT CONVERT(datetime, '10.12.99',1)

SELECT CONVERT(datetime, '10.12.99',4)
```

The first statement assumes that the date is style 1, *mm.dd.yy*, so the string can be converted to the corresponding *datetime* value and returned to the client. The client then returns the *datetime* value according to its own display conventions. The second statement assumes that the date is represented as *dd.mm.yy*. If we had tried to use style 102 or 104, both of which require a four-digit year, we would have received a conversion error because we specified only a two-digit year. CONVERT can also be useful if you insert the current date using the GETDATE function but don't consider the time elements of the *datetime* datatype to be meaningful. (Remember that SQL Server doesn't currently have separate date and time datatypes.) You can use CONVERT to format and insert *datetime* data with only a date element. Without the time specification in the GETDATE value, the time will consistently be stored as 12:00AM for any given date. That eliminates problems that can occur when you search among columns or join columns of *datetime*. For example, if the date elements are equal between columns but the time element is not, an equality search will fail. You can eliminate this equality problem by making the time portion consistent. Consider this:

```
CREATE TABLE my_date (Col1 datetime)
INSERT INTO my_date VALUES (CONVERT(char(10), GETDATE(), 112))
```

In this **example**, the CONVERT function converts the current date and time value to a character string that doesn't include the time. However, because the table we're inserting into expects a *datetime* value, the new character string is converted back to a *datetime* datatype using the default time of midnight.

You can also use the optional *style* argument when you convert *float*, *real*, *money*, or *smallmoney* to a character datatype. When you convert from *float* or *real*, the style allows you to force the output to appear in scientific notation and also allows you to specify either 8 or 16 digits. When you convert from a *money* type to a character type, the style allows you to specify whether you want a comma to appear every three digits and whether you want two digits to the right of the decimal point or four.

In Table 13, the column on the left represents the *style* values for *float* or *real* conversion to character data.

Table 13 Values for the style argument of the CONVERT function when you convert a floating-point expression to a character datatype

Style Value	Output
0 (the default)	Six digits maximum. In scientific notation when appropriate.
1	Always eight digits. Always scientific notation.
2	Always sixteen digits. Always scientific notation.

In Table 14, the column on the left represents the *style* value for *money* or *smallmoney* conversion to character data.

Table 14. Values for the style argument of the CONVERT function when you convert a money expression to a character datatype

Style Value	Output
0 (the default)	No commas to the left of the decimal point. Two digits appear to the right of the decimal point. Example: 4235.98.
1	Commas every three digits to the left of the decimal point. Two digits appear to the right of the decimal point. Example: 3,510.92.
2	No commas to the left of the decimal point. Four digits appear to the right of the decimal point. Example: 4235.9819.

Here's an example from the *pubs* database. The *advance* column in the *titles* table lists the advance paid by the publisher for each book, and most of the amounts are greater than \$1000. In the *titles* table, the value is stored as a *money* datatype. The following query returns the value of *advance* as a *money* value, a *varchar* value with the default style, and as *varchar* with each of the two optional styles:

```
SELECT "Money" = advance,
       "Varchar" = CONVERT(varchar(10), advance),
       "Varchar-1" = CONVERT(varchar(10), advance, 1),
       "Varchar-2" = CONVERT(varchar(10), advance, 2)
FROM titles
```


Here's the abbreviated output:

Money	Varchar	Varchar-1	Varchar-2
5000.0000	5000.00	5,000.00	5000.0000
5000.0000	5000.00	5,000.00	5000.0000
10125.0000	10125.00	10,125.00	10125.0000
5000.0000	5000.00	5,000.00	5000.0000
.0000	0.00	0.00	0.0000
15000.0000	15000.00	15,000.00	15000.0000
NULL	NULL	NULL	NULL
7000.0000	7000.00	7,000.00	7000.0000
8000.0000	8000.00	8,000.00	8000.0000

Be aware that when you use CONVERT, the conversion occurs on the server and the value is sent to the client application in the converted datatype. Of course, it's also common for applications to convert between datatypes, but conversion at the server is completely separate from conversion at the client. It's the client that determines how to display the value returned. In the previous example, the results in the first column have no comma and four decimal digits. This is how SQL Query Analyzer displays values of type money that it receives from SQL Server. If we run this same query using the command-line ISQL tool, the (abbreviated) results look like this:

Money	Varchar	Varchar-1	Varchar-2
5,000.00	5000.00	5,000.00	5000.0000
5,000.00	5000.00	5,000.00	5000.0000
10,125.00	10125.00	10,125.00	10125.0000
5,000.00	5000.00	5,000.00	5000.0000
0.00	0.00	0.00	00
15,000.00	15000.00	15,000.00	15000.0000
(null)	(null)	(null)	(null)
7,000.00	7000.00	7,000.00	7000.0000
8,000.00	8000.00	8,000.00	8000.0000

In addition to determining how to display a value of a particular datatype, the client program determines whether the output should be left-justified or right-justified and how NULLs are displayed. It is frequently asked how to change this default output to, for example, print *money* values with four decimal digits in SQL Query Analyzer.

Unfortunately, SQL Query Analyzer has its own set of predefined formatting specifications, and, for the most part, you can't change them. Other client programs, such as report writers, might give you a full range of capabilities for specifying the output format, but SQL Query Analyzer isn't a report writer.

Here's another example of how the display can be affected by the client. If we use the GETDATE function to select the current date, it is returned to the client application as a *datetime* datatype. The client application will likely convert that internal date representation to a string for display. On the other hand, if we use GETDATE but also explicitly use CONVERT to change the *datetime* to character data, the column is sent to the calling application already as a character string. So let's see what one particular client will do with that returned data. The command-line program ISQL is

a DB-Library program and uses functions in DBLibrary for binding columns to character strings. DB-Library allows such bindings to automatically pick up locale styles based on the settings of the workstation running the application. (You must select the Use International Settings option in the SQL Server Client Network Utility in order for this to occur by default.) So a column returned internally as a *datetime* datatype is converted by *isql.exe* into the same format as the locale setting of Windows. A column containing a date representation as a character string is not reformatted, of course.

When we issue the following SELECT statement with SQL Server configured for U.S. English as the default language and with Windows NT and Windows 2000 on the client configured as English (United States), the date and string look alike:

```
SELECT
'returned as date'=GETDATE(),
'returned as string'=CONVERT(varchar(20), GETDATE())
```

returned as date	returned as string
-----	-----
Dec 3 2000 3:10PM	Dec 3 2000 3:10PM

But if we change the locale in the Regional Options application of the Windows 2000 Control Panel to French (Standard), the same SELECT statement returns the following:

returned as date	returned as string
-----	-----
3 déc. 2000 15:10	Dec 3 2000 3:10PM

You can see that the value returned to the client in internal date format was converted at the client workstation in the format chosen by the application. The conversion that uses CONVERT was formatted at the server.

This discussion is also relevant to formatting numbers and currency with different regional settings. We cannot use the command-line OSQL program to illustrate the same behavior. OSQL, like SQL Query Analyzer, is an ODBC-based program, and datetime values are converted to ODBC Canonical format, which is independent of any regional settings.

Using OSQL, the previous query would return this result:

returned as date	returned as string
-----	-----
2000-12-03 15:10:24.793	Dec 3 2000 3:10PM

The rest of the book primarily uses CAST instead of CONVERT when converting between datatypes. CONVERT is used only when the *style* argument is needed.

Some conversions are automatic and implicit, so using CAST is unnecessary (but OK). For example, converting between numbers with types *int*, *smallint*, *tinyint*, *float*, *numeric*, and so on happens automatically and implicitly as long as an overflow doesn't occur, in which case you'd get an error for arithmetic overflow. Converting numbers with decimal places to integer datatypes results in the truncation of the values to the right of the decimal point—without warning.

(Converting between decimal or numeric datatypes requires that you explicitly use CAST if a loss of precision is possible.)

Even though Figure 10-3 indicates that conversion from character strings to *datetime* values can happen implicitly, this conversion is possible only when SQL Server can figure out what the *datetime* value needs to be. If you use wildcards in your character strings, SQL Server might not be able to do a proper conversion.

Suppose we use the following query to find all orders placed in August 1996 and stored in the *orders* table in the *Northwind* database:

USE Northwind

```
SELECT * FROM orders WHERE OrderDate BETWEEN '8/1/96' and '8/31/96'
```

Because all the dates have a time of midnight in the *orders* table, SQL Server will correctly interpret this query and return 25 rows. It converts the two string constants to *datetime* values, and then it can do a proper chronological comparison. However, if your string has a wildcard in it, SQL Server cannot convert it to a *datetime*, so instead it converts the *datetime* into a string. To do this, it uses its default date format, which, as you've seen, is *mon dd yyyy hh:miAM (or PM)*. So this is the format you have to match in the character string with wildcards. Obviously, *%1996%* will match but *'8/1/%'* won't. Although SQL Server is usually very flexible in how dates can be entered, once you compare a *datetime* to a string with wildcards, you can assume only the default format.

Note that in the default format, there are two spaces before a single-digit day. So if we want to find all rows in the *orders* table with an *OrderDate* of July 8, 1996, at any time, we have to use the following and be sure to put two spaces between *Jul* and *8*:

```
WHERE OrderDate LIKE 'Jul  8 1996%'
```

Other conversions, such as between character and integer data, can be performed explicitly only by using *CAST* or *CONVERT*. Conversions between certain datatypes are nonsensical—for example, between a *float* and a *datetime*—and attempting such an operation results in error 529, which states that the conversion is impossible. Figure 10-3, reprinted from the SQL Server documentation, is one that you'll want to refer to periodically for conversion issues.

Date and Time Functions

Operations on *datetime* values are common, such as "get current date and time," "do date arithmetic—50 days from today is what date," or "find out what day of the week falls on a specific date." Programming languages such as C or Visual Basic are loaded with functions for operations like these. Transact-SQL also provides a nice assortment to choose from, as shown in Table 10-11. The *datetime* parameter is any expression with a SQL Server *datetime* datatype or one that can be implicitly converted, such as an appropriately formatted character string like 1999.10.31. The *datepart* parameter uses the encodings shown in Table 10-12. Either the full name or the abbreviation can be passed as an argument.

Table 15. Date and time functions in Transact-SQL

Date Function	Return Type	Description
DATEADD (datepart, number, datetime)	datetime	Produces a date by adding an interval to a specified date
DATEDIFF (datepart, datetime1, datetime2)	int	Returns the number of datepart "boundaries" crossed between two specified dates
DATENAME (datepart, datetime)	varchar	Returns a character string representing the specified datepart of the specified date
DATEPART (datepart, datetime)	int	Returns an integer representing the specified datepart of the specified date
DAY (datetime)	int	Returns an integer representing the day datepart of the specified date
MONTH (datetime)	int	Returns an integer representing the month datepart of the specified date
YEAR (datetime)	int	Returns an integer representing the year datepart of the specified date
GETDATE	datetime	Returns the current system date and time in the SQL Server standard internal format for datetime values
GETUTCDATE	datetime	Returns the current Universal Time Coordinate (Greenwich Mean Time), which is derived from the current local time and the time zone setting in the operating system of the machine SQL Server is running on

Table 16. Values for the date part parameter

datepart (Full Name) Abbreviation Values

Year	Yy	1753—9999
Quarter	Qq	1—4
Month	Mm	1—12
dayofyear	Dy	1—366
Day	Dd	1—31
Week	Wk	1—53
weekday	Dw	1—7 (Sunday-Saturday)
Hour	Hh	0—23
Minute	Mi	0—59
Second	Ss	0—59
millisecond	Ms	0—999

Like other functions, the date functions provide more than simple convenience. Suppose we need to find all records entered on the second Tuesday of every month and all records entered within 48 weeks of today. Without SQL Server date functions, the only way we could accomplish such a query would be to select all the rows and return them to the application and then filter them there. With a lot of data and a slow network, this can get ugly. With the date functions, the process is simple and efficient, and only the rows that meet these criteria are selected. For this example, let's assume that the *records* table includes these columns:

```
Record_number  int
Entered_on     datetime
```

This query returns the desired rows:

```
SELECT Record_number, Entered_on
FROM records
```

```
WHERE
DATEPART(WEEKDAY, Entered_on) = 3
-- Tuesday is 3rd day of week (in USA)
AND
DATEPART(DAY, Entered_on) BETWEEN 8 AND 14
-- The 2nd week is from the 8th to 14th
AND
DATEDIFF(WEEK, Entered_on, GETDATE()) <= 48
-- Within 48 weeks of today
```

The day of the week considered "first" is locale-specific and depends on the DATEFIRST setting. SQL Server 2000 also allows you to add or subtract an integer to or from a datetime value. This is actually just a shortcut for the DATEADD function, with a datepart of day. For example, the following two queries are equivalent. They each return the date 14 days from today:

```
SELECT DATEADD(day, 14, GETDATE())
```

```
SELECT GETDATE() + 14
```

The date functions don't do any rounding. The DATEDIFF function just subtracts the components from each date that correspond to the *datepart* specified. For example, to find the number of years between New Year's Day and New Year's Eve of the same year, this query would return a value of 0. Because the *datepart* specifies years, SQL Server subtracts the year part of the two dates, and because they're the same, the result of subtracting them is 0:

```
SELECT DATEDIFF(yy, 'Jan 1, 1998', 'Dec 31, 1998')
```

However, if we want to find the difference in years between New Year's Eve of one year and New Year's Day (the next day), the following query would return a value of 1 because the difference in the year part is 1:

```
SELECT DATEDIFF(yy, 'Dec 31, 1998', 'Jan 1, 1999')
```

There's no built-in mechanism for determining whether two date values are actually the same day unless you've forced all *datetime* values to use the default time of midnight. (We saw a technique for doing this earlier.) If you want to compare two *datetime* values (@date1 and @date2) to determine whether they're on the same day, regardless of the time, one technique is to use a three-part condition like this:

```
IF (DATEPART(mm, @date1) = DATEPART(mm, @date2)) AND
   (DATEPART(dd, @date1) = DATEPART(dd, @date2)) AND
   (DATEPART(yy, @date1) = DATEPART(yy, @date2))
PRINT 'The dates are the same'
```

But there is a much simpler way. Even though these two queries both return the same *dd* value, if we try to find the difference in days between these two dates, SQL Server is smart enough to know that they are different dates:

```
SELECT DATEPART(dd, '7/5/99')
SELECT DATEPART(dd, '7/5/00')
```

So, the following query returns the message that the dates are different:

```
IF DATEDIFF(dd, '7/5/99', '7/5/00') = 0
   PRINT 'The dates are the same'
ELSE PRINT 'The dates are different'
```

And the following general form allows us to indicate whether two dates are really the same date, irrespective of the time:

```
IF DATEDIFF(day, @date1, @date2) = 0
    PRINT 'The dates are the same'
```

Math Functions

Transact-SQL math functions are straightforward and typical. Many are specialized and of the type you learned about in trigonometry class. If you don't have an engineering or mathematical application, you probably won't use those. A handful of math functions are useful in general types of queries in applications that aren't mathematical in nature. ABS, CEILING, FLOOR, and ROUND are the most useful functions for general queries to find values within a certain range. The random number function, RAND, is useful for generating test data or conditions. You'll see examples of RAND later in this chapter.

Table 17 shows the complete list of math functions and a few simple examples. Some of the examples use other functions in addition to the math ones to illustrate how to use functions within functions.

Table 17. Math functions in Transact-SQL

Function	Parameters	Result
ABS	(<i>numeric_expr</i>)	Absolute value of the numeric expression. Results returned are of the same type as <i>numeric_expr</i> .
ACOS	(<i>float_expr</i>)	Angle (in radians) whose cosine is the specified approximate numeric (<i>float</i>) expression.
ASIN	(<i>float_expr</i>)	Angle (in radians) whose sine is the specified approximate numeric (<i>float</i>) expression.
ATAN	(<i>float_expr</i>)	Angle (in radians) whose tangent is the specified approximate numeric (<i>float</i>) expression.
ATN2	(<i>float_expr1</i> , <i>float_expr2</i>)	Angle (in radians) whose tangent is (<i>float_expr1/float_expr2</i>) between two approximate numeric (<i>float</i>) expressions.
CEILING	(<i>numeric_expr</i>)	Smallest integer greater than or equal to the numeric expression. Result returned is the integer portion of the same type as <i>numeric_expr</i> .
COS	(<i>float_expr</i>)	Trigonometric cosine of the specified angle (in radians) in an approximate numeric (<i>float</i>) expression.
COT	(<i>float_expr</i>)	Trigonometric cotangent of the specified angle (in radians) in an approximate numeric (<i>float</i>) expression.
DEGREES	(<i>numeric_expr</i>)	Degrees converted from radians of the numeric expression. Results are of the same type as <i>numeric_expr</i> .
EXP	(<i>float_expr</i>)	Exponential value of the specified approximate numeric (<i>float</i>) expression.
FLOOR	(<i>numeric_expr</i>)	Largest integer less than or equal to the specified numeric expression. Result is the integer portion of the same type as <i>numeric_expr</i> .
LOG	(<i>float_expr</i>)	Natural logarithm of the specified approximate numeric (<i>float</i>) expression.
LOG10	(<i>float_expr</i>)	Base-10 logarithm of the specified approximate numeric (<i>float</i>) expression.
PI	()	Constant value of 3.141592653589793.

POWER (<i>numeric_expr</i> , <i>y</i>)	Value of numeric expression to the power of <i>y</i> , where <i>y</i> is a numeric datatype (<i>bigint</i> , <i>decimal</i> , <i>float</i> , <i>int</i> , <i>money</i> , <i>numeric</i> , <i>real</i> , <i>smallint</i> , <i>smallmoney</i> , or <i>tinyint</i>). Result is of the same type as <i>numeric_expr</i> .
RADIANS (<i>numeric_expr</i>)	Radians converted from degrees of the numeric expression. Result is of the same type as <i>numeric_expr</i> .
RAND ([<i>seed</i>])	Random approximate numeric (<i>float</i>) value between 0 and 1, optionally specifying an integer expression as the seed.
ROUND (<i>numeric_expr</i> , <i>length</i>)	Numeric expression rounded off to the length (or precision) specified as an integer expression (<i>bigint</i> , <i>tinyint</i> , <i>smallint</i> , or <i>int</i>). Result is of the same type as <i>numeric_expr</i> . ROUND always returns a value even if <i>length</i> is illegal. If the specified length is positive and longer than the digits after the decimal point, 0 is added after the fraction digits. If the length is negative and larger than or equal to the digits before the decimal point, ROUND returns 0.00.
SIGN (<i>numeric_expr</i>)	Returns the positive (+1), zero (0), or negative (-1) sign of the numeric expression. Result is of the same type as <i>numeric_expr</i> .
SIN (<i>float_expr</i>)	Trigonometric sine of the specified angle (measured in radians) in an approximate numeric (<i>float</i>) expression.
SQRT (<i>float_expr</i>)	Square root of the specified approximate numeric (<i>float</i>) expression.
SQUARE (<i>float_expr</i>)	Square of the specified approximate numeric (<i>float</i>) expression.
TAN (<i>float_expr</i>)	Trigonometric tangent of the specified angle (measured in radians)

The following examples show some of the math functions at work

EXAMPLE 1

Produce a table with the sine, cosine, and tangent of all angles in multiples of 10, from 0 through 180. Format the return value as a string of eight characters, with the value rounded to five decimal places. (We need one character for the decimal point and one for a negative sign, if needed.)

```
DECLARE @degrees smallint
DECLARE @radians float
SELECT @degrees=0
SELECT @radians=0
WHILE (@degrees <= 180)
BEGIN
    SELECT
        DEGREES=@degrees,
        RADIANS=STR(@radians, 8, 5),
        SINE=STR(SIN(@radians), 8, 5),
        COSINE=STR(COS(@radians), 8, 5),
        TANGENT=STR(TAN(@radians), 8, 5)
    SELECT @degrees=@degrees + 10
    SELECT @radians=RADIANS(CONVERT(float, @degrees))
END
```

This example actually produces 19 different result sets because the SELECT statement is issued once for each iteration of the loop. These separate result sets are concatenated and appear as one result set in this example. That works fine for illustrative purposes, but you should avoid doing an operation like this in the real world, especially on a slow network. Each result set carries with it metadata to describe itself to the client application.

And here are the results concatenated as a single table:

DEGREES	RADIANS	SINE	COSINE	TANGENT
-----	-----	-----	-----	-----
0	0.00000	0.00000	1.00000	0.00000
10	0.17453	0.17365	0.98481	0.17633
20	0.34907	0.34202	0.93969	0.36397
30	0.52360	0.50000	0.86603	0.57735
40	0.69813	0.64279	0.76604	0.83910
50	0.87266	0.76604	0.64279	1.19175
60	1.04720	0.86603	0.50000	1.73205
70	1.22173	0.93969	0.34202	2.74748
80	1.39626	0.98481	0.17365	5.67128
90	1.57080	1.00000	0.00000	*****
100	1.74533	0.98481	-0.1736	-5.6713
110	1.91986	0.93969	-0.3420	-2.7475
120	2.09440	0.86603	-0.5000	-1.7321
130	2.26893	0.76604	-0.6428	-1.1918
140	2.44346	0.64279	-0.7660	-0.8391
150	2.61799	0.50000	-0.8660	-0.5774
160	2.79253	0.34202	-0.9397	-0.3640
170	2.96706	0.17365	-0.9848	-0.1763
180	3.14159	0.00000	-1.0000	-0.0000

EXAMPLE 2

Express in whole-dollar terms the range of prices (non-null) of all books in the *titles* table. This example combines the scalar functions FLOOR and CEILING inside the aggregate functions MIN and MAX:

```
SELECT 'Low End'=MIN(FLOOR(price)),
       'High End'=MAX(CEILING(price))
FROM titles
```

And the result:

Low End	High End
-----	-----

2.00 23.00

EXAMPLE 3

Use the same *records* table that was used in the earlier date functions example. Find all records within 150 days of September 30, 1997. Without the absolute value function ABS, you would have to use BETWEEN or provide two search conditions and AND them to account for both 150 days before and 150 days after that date. ABS lets you easily reduce that to a single search condition.

```
SELECT Record_number, Entered_on
FROM records
WHERE
ABS(DATEDIFF(DAY, Entered_on, '1997.09.30')) <= 150
    Plus or minus 150 days
```

String Functions

String functions make it easier to work with character data. They let you slice and dice character data, search it, format it, and alter it. Like other scalar functions, string functions allow you to perform functions directly in your search conditions and SQL batches that would otherwise need to be returned to the calling application for further processing. The string functions appear in Table 18 For more detailed information, consult the online documentation.

Table 18. String functions in Transact-SQL

Function	Parameters	Result
ASCII	(<i>char_expr</i>)	Indicates the numeric code value of the leftmost character of a character expression.
CHAR	(<i>integer_expr</i>)	Returns the character represented by the ASCII code. The ASCII code should be a value from 0 through 255; otherwise, NULL is returned.
CHARINDEX	('pattern', <i>expression</i>)	Returns the starting position of the specified exact pattern. A <i>pattern</i> is a <i>char_expr</i> . The second parameter is an <i>expression</i> , usually a column name, in which SQL Server searches for <i>pattern</i> .
DIFFERENCE	(<i>char_expr1</i> , <i>char_expr2</i>)	Shows the difference between the values of two character expressions as returned by the SOUNDEX function. DIFFERENCE compares two strings and evaluates the similarity between them, returning a value 0 through 4. The value 4 is the best match.
LEFT	(<i>char_expression</i> , <i>int_expression</i>)	Returns <i>int_expression</i> characters from the left of the <i>char_expression</i> .
LEN	(<i>char_expression</i>)	Returns the number of characters, rather than the number of bytes, of <i>char_expression</i> , excluding trailing blanks.
LOWER	(<i>char_expr</i>)	Converts uppercase character data to lowercase.
LTRIM	(<i>char_expr</i>)	Removes leading blanks.
NCHAR	(<i>int_expression</i>)	Returns the Unicode character with code <i>int_expression</i> , as defined by the Unicode standard.
PATINDEX	('pattern%', <i>expression</i>)	Returns the starting position of the first occurrence of <i>pattern</i> in the specified <i>expression</i> , or 0 if the pattern isn't found.
	QUOTENAME	

QUOTENAME	('char_string' [, 'quote_character'])	Returns a Unicode string with <i>quote_character</i> used as the delimiter to make the input string a valid SQL Server delimited identifier.
REPLACE	('char_expression1', 'char_expression2', 'char_expression3')	Replaces all occurrences of <i>char_expression2</i> in <i>char_expression1</i> with <i>char_expression3</i> .
REPLICATE	(char_expr, integer_expr)	Repeats a character expression a specified number of times. If <i>integer_expr</i> is negative, NULL is returned.
REVERSE	(char_expr)	Returns the <i>char_expr</i> backwards. This function takes a constant, variable, or column as its parameter.
RIGHT	(char_expr, integer_expr)	Returns part of a character string starting <i>integer_expr</i> characters from the right. If <i>integer_expr</i> is negative, NULL is returned.
RTRIM	(char_expr)	Removes trailing blanks.
SOUNDEX	(char_expr)	Returns a four-character (SOUNDEX) of two strings. The SOUNDEX function converts an alpha string to a four-digit code used to find similar-sounding words or names.
SPACE	(integer_expr)	Returns a string of repeated spaces. The number of spaces is equal to <i>integer_expr</i> . If <i>integer_expr</i> is negative, NULL is returned.
STR	(float_expr [, length [, decimal]])	Returns character data converted from numeric data. The <i>length</i> is the total length, including decimal point, sign, digits, and spaces. The <i>decimal</i> value is the number of spaces to the right of the decimal point.
STUFF	(char_expr1, start, length, char_expr2)	Deletes <i>length</i> characters from <i>char_expr1</i> at <i>start</i> and then inserts <i>char_expr2</i> into <i>char_expr1</i> at <i>start</i> .
SUBSTRING	(expression, start, length)	Returns part of a character or binary string. The first parameter can be a character or binary string, a column name, or an expression that includes a column name. The second parameter specifies where the substring begins. The third parameter specifies the number of characters in the substring.
UNICODE	('nchar_expression')	Returns the integer value, as defined by the Unicode standard, for the first character of <i>nchar_expression</i> .
UPPER	(char_expr)	Converts lowercase character data to uppercase.

ASCII and CHAR functions

The function name ASCII is really a bit of a misnomer. ASCII is only a 7-bit character set and hence can deal with only 128 characters. The character parameter to this function doesn't need to be an ASCII character. The ASCII function returns the code point for the character set associated with the database or the column if the argument to the function is a column from a table. The return value can be from 0 through 255. For example, if the current database is using a Latin-1 character set, the following statement returns 196, even though the character `Ä` isn't an ASCII character.

```
SELECT ASCII('Ä')
```

The CHAR function is handy for generating test data, especially when combined with RAND and REPLICATE. CHAR is also commonly used for inserting control characters such as tabs and carriage returns into your character string. Suppose, for example, that we want to return authors' last names and first names concatenated as a single field, but with a carriage return (0x0D, or decimal 13) separating them so that without further manipulation in our application, the names occupy two lines when displayed. The CHAR function makes this simple:

```
SELECT 'NAME'=au_fname + CHAR(13) + au_lname
FROM authors
```

Here's the abbreviated result:

```
NAME
-----
Johnson
White
Marjorie
Green
Cheryl
Carson
Michael
O'Leary
Dean
Straight
Meander
Smith
Abraham
Bennet
Ann
Dull
```

UPPER and LOWER functions

These functions are useful if you must perform case-insensitive searches on data that is case sensitive. The following example copies the *authors* table from the *pubs* database into a new table, using a case-sensitive collation for the authors' name fields.

```
SELECT au_id,
       au_lname = au_lname collate Latin1_General_CS_AS,
       au_fname = au_fname collate Latin1_General_CS_AS,
       phone, address, city, state, zip, contract
INTO authors_CS
FROM authors
```

This query finds no rows in the new table even though author name Cheryl Carson is included in that table:

```
SELECT COUNT(*) FROM authors_CS WHERE au_lname='CARSON'
```

If we change the query to the following, the row will be found:

```
SELECT COUNT(*) FROM authors_CS WHERE UPPER(au_lname)='CARSON'
```

If the value to be searched might be of mixed case, you need to use the function on both sides of the equation. This query will find the row in the case-sensitive table:

```
DECLARE @name_param varchar(30)
SELECT @name_param='cArSoN'
SELECT COUNT(*) FROM authors_CS WHERE UPPER(au_lname)=UPPER(@name_param)
```

In these examples, even though we might have an index on *au_lname*, it won't be useful because the index keys aren't uppercase. However, we could create a computed column on *UPPER(au_lname)* and build an index on the computed column.

You'll also often want to use UPPER or LOWER in stored procedures in which character parameters are used. For example, in a procedure that expects *Y* or *N* as a parameter, you'll likely want to use one of these functions in case *y* is entered instead of *Y*.

TRIM functions

The functions LTRIM and RTRIM are handy for dealing with leading or trailing blanks. Recall that by default (and if you don't enable the ANSI_PADDING setting) a *varchar* datatype is automatically right-trimmed of blanks, but a fixed-length *char* isn't. Suppose that we want to concatenate the *type* column and the *title_id* column from the *titles* table, with a colon separating them but with no blanks. The following query doesn't work because the trailing blanks are retained from the *type* column:

```
SELECT type + ':' + title_id FROM titles
```

Here's the result (in part):

```
business :BU1032
business :BU1111
business :BU2075
business :BU7832
mod_cook :MC2222
mod_cook :MC3021
UNDECIDED :MC3026
popular_comp:PC1035
popular_comp:PC8888
popular_comp:PC9999
```

But RTRIM returns what we want:

```
SELECT RTRIM(type) + ':' + title_id FROM titles
```

And here's the result (in part):

```
business:BU1032
business:BU1111
business:BU2075
business:BU7832
mod_cook:MC2222
mod_cook:MC3021
UNDECIDED:MC3026
popular_comp:PC1035
popular_comp:PC8888
```

popular_comp:PC9999

String manipulation functions

Functions are useful for searching and manipulating partial strings that include SUBSTRING, CHARINDEX, PATINDEX, STUFF, REPLACE, REVERSE, and REPLICATE. CHARINDEX and PATINDEX are similar, but CHARINDEX demands an exact match and PATINDEX works with a regular expression search.

You can also use CHARINDEX or PATINDEX as a replacement for LIKE. For example, instead of saying WHERE name LIKE '%Smith%', you can say WHERE CHARINDEX('Smith', name) > 0.

Suppose we want to change occurrences of the word "computer" within the notes field of the titles table and replace it with "Hi-Tech Computers." Assume that SQL Server is case sensitive and, for simplicity, that we know that "computer" won't appear more than once per column. The regular expression computer[^s] always finds the word "computer" and ignores "computers," so it'll work perfectly with PATINDEX.

Here's the data before the change:

```
SELECT title_id, notes
FROM titles WHERE notes LIKE '%[Cc]omputer%'
```

title_id	notes
BU7832	Annotated analysis of what computers can do for you: a no-hype guide for the critical user.
PC8888	Muckraking reporting on the world's largest computer hardware and software manufacturers.
PC9999	A must-read for computer conferencing.
PS1372	A must for the specialist, this book examines the difference between those who hate and fear computers and those who don't.
PS7777	Protecting yourself and your loved ones from undue emotional stress in the modern world. Use of computer and nutritional aids emphasized.

You might consider using the REPLACE function to make the substitution. However, REPLACE requires that we search for a specific string that can't include wildcards like [^s]. Instead, we can use the older STUFF function, which has a slightly more complex syntax. STUFF requires that we specify the starting location within the string to make a substitution. We can use PATINDEX to find the correct starting location, and PATINDEX allows wildcards:

```
UPDATE titles
SET notes=STUFF(notes, PATINDEX('%computer[^s]%', notes),
    DATALENGTH('computer'), 'Hi-Tech Computers')
WHERE PATINDEX('%computer[^s]%', notes) > 0
```

Here's the data after the change:

```
SELECT title_id, notes
FROM titles
WHERE notes LIKE '%[Cc]omputer%'
```

title_id	notes
BU7832	Annotated analysis of what computers can do for you: a no-hype guide for the critical user.
PC8888	Muckraking reporting on the world's largest Hi-Tech Computers hardware and software manufacturers.
PC9999	A must-read for Hi-Tech Computers conferencing.
PS1372	A must for the specialist, this book examines the difference between those who hate and fear computers and those who don't.
PS7777	Protecting yourself and your loved ones from undue emotional stress in the modern world. Use of Hi-Tech Computers and nutritional aids emphasized.

Of course, we could have simply provided 8 as the *length* parameter of the string "computer" to be replaced. But we used yet another function, LEN, which would be more realistic if we were creating a general-purpose, search-and-replace procedure. Note that DATALENGTH returns NULL if the expression is NULL, so in your applications, you might go one step further and use DATALENGTH inside the ISNULL function to return 0 in the case of NULL. The REPLICATE function is useful for adding filler characters—such as for test data. (In fact, generating test data seems to be about the only practical use for this function.) The SPACE function is a special-purpose version of REPLICATE: it's identical to using REPLICATE with the space character. EVERSE reverses a character string. You can use REVERSE in a few cases to store and, more importantly, to index a character column backward to get more selectivity from the index. But in general, these three functions are rarely used.

SOUNDEX and DIFFERENCE Functions

If you've ever wondered how telephone operators are able to give you telephone numbers so quickly when you call directory assistance, chances are they're using a SOUNDEX algorithm. The SOUNDEX and DIFFERENCE functions let you search on character strings that sound similar when spoken. SOUNDEX converts each string to a four-character code. DIFFERENCE can then be used to evaluate the level of similarity between the SOUNDEX values for two strings as returned by SOUNDEX. For example, you could use these functions to look at all rows that sound like "Erickson," and they would find "Erickson," "Erikson," "Ericson," "Ericksen," "Ericsen," and so on.

SOUNDEX algorithms are commonplace in the database business, but the exact implementation can vary from vendor to vendor. For SQL Server's SOUNDEX function, the first character of the four-character SOUNDEX code is the first letter of the word, and the remaining three characters are single digits that describe the phonetic value of the first three consonants of the word with the underlying assumption that there's one consonant per syllable. Identical consonants right next to each other are treated as a single consonant. Only nine phonetic values are possible in this scheme because only nine possible digits exist. SOUNDEX relies on phonetic similarities between sounds to group consonants together. In fact, SQL Server's SOUNDEX algorithm uses only seven of the possible nine digits. No SOUNDEX code uses the digits 8 or 9.

Vowels are ignored, as are the characters *h* and *y*, so "Zastrak" would have the same code as "Zasituryk." If no second or subsequent consonant exists, the phonetic value for it is 0. In addition, SQL Server's SOUNDEX algorithm stops evaluating the string as soon as the first nonalphabetic character is found. So if you have a hyphenated or two-word name (with a space in

between), SOUNDEX ignores the second part of the name. More seriously, SOUNDEX stops processing names as soon as it finds an apostrophe. So "O'Flaherty," "O'Leary," and "O'Hara" all have the same SOUNDEX code, namely O000.

DIFFERENCE internally compares two SOUNDEX values and returns a score from 0 through 4 to indicate how close the match is. A score of 4 is the best match, and 0 means no matches were found. Executing *DIFFERENCE(a1, a2)* first generates the four-character SOUNDEX values for a1 (call it *sx_a1*) and a2 (call it *sx_a2*). Then, if all four values of the SOUNDEX value *sx_a2* match the values of *sx_a1*, the match is perfect and the level is 4. Note that this compares the SOUNDEX values by character position, not by actual characters.

The names "Smythe" and "Smith" both have a SOUNDEX value of S530, so their difference level is 4, even though the spellings differ. If the first character (a letter, not a number) of the SOUNDEX value *sx_a1* is the same as the first character of *sx_a2*, the starting level is 1. If the first character is different, the starting level is 0. Then each character in *sx_a2* is successively compared to all characters in *sx_a1*. When a match is found, the level is incremented and the next scan on *sx_a1* starts from the location of the match. If no match is found, the next *sx_a2* character is compared to the entire four-character list of *sx_a1*.

The preceding description of the algorithms should make it clear that SOUNDEX at best provides an approximation. Even so, sometimes it works extremely well. Suppose we want to query the *authors* table for names that sound similar to "Larsen." We'll define *similar* to mean "having a SOUNDEX value of 3 or 4":

```
SELECT au_lname,  
Soundex=SOUNDEX(au_lname),  
Diff_Larsen=DIFFERENCE(au_lname, 'Larsen')  
FROM authors  
WHERE  
DIFFERENCE(au_lname, 'Larsen') >= 3
```

Here's the result:

au_lname	Soundex	Diff_Larsen
Carson	C625	3
Karsen	K625	3

In this case, we found two names that rhyme with "Larsen" and didn't get any bad hits of names that don't seem close. For example, do you think "Bennet" and "Smith" sound similar? Well, SOUNDEX does. When you investigate, you can see why the two names have a similar SOUNDEX value. The SOUNDEX values have a different first letter, but the m and n sounds are converted to the same digit, and both names include a t, which is converted to a digit for the third position. "Bennet" has nothing to put in the fourth position, so it gets a 0. For Smith, the h becomes a 0. So, except for the initial letter, the rest of the SOUNDEX strings are the same. Hence, the match.

This type of situation happens often with SOUNDEX—you get hits on values that don't seem close, although usually you won't miss the ones that are similar. So if you query the *authors* table for similar values (with a DIFFERENCE value of 3 or 4) as "Smythe," you get a perfect hit on "Smith" as you'd expect, but you also get a close match for "Bennet," which is far from obvious:

```
SELECT au_lname,  
Soundex=SOUNDEX(au_lname),
```

```
Diff_Smythe=DIFFERENCE(au_Iname, 'Smythe')
FROM authors
WHERE
DIFFERENC3E(au_Iname, 'Smythe') >= 3
```

```
au_Iname   Soundex   Diff_Smythe
-----   -
Smith      S530
Bennet     B530

4
3
```

Sometimes SOUNDEX misses close matches altogether. This happens when consonant blends are present. For example, you might expect to get a close match between "Knight" and "Nite." But you get a low value of 1 in the DIFFERENCE between the two:

```
SELECT
"SX_KNIGHT"=SOUNDEX('Knight'),
"SX_NITE"=SOUNDEX('Nite'),
"DIFFERENCE"=DIFFERENCE('Nite', 'Knight')
```

```
SX_KNIGHT  SX_NITE  DIFFERENCE
-----  -
K523      N300      1
```

System Functions

System functions are most useful for returning certain metadata, security, or configuration settings. Table 19 lists of some of the more commonly used system functions, and it's followed by a brief discussion of the interesting features of a few of them. For complete details, see the SQL Server online documentation.

Table 19 System functions in Transact-SQL

Function	Parameters	Description
APP_NAME	NONE	Returns the program name for the current connection if one has been set by the program before you log on.
COALESCE	(<i>expression1, expression2, ... expressionN</i>)	A specialized form of the CASE statement. Returns the first non-null expression.
COL_LENGTH	(<i>'table_name', 'column_name'</i>)	The defined (maximum) storage length of a column.
COL_NAME	(<i>table_id, column_id</i>)	The name of the column.
DATALENGTH	(<i>'expression'</i>)	The length of an expression of any datatype.
DB_ID	(<i>'database_name'</i>)	The database identification number.
DB_NAME	(<i>database_id</i>)	The database name.
GETANSINULL	(<i>'database_name'</i>)	The default nullability for the database. Returns 1 when the nullability is the ANSI NULL default.

HOST_ID	NONE	The process ID of the application calling SQL Server on the workstation. (If you look at the PID column on the Processes tab in Windows Task Manager, you will see this number associated with the client application.)
HOST_NAME	NONE	The workstation name.
IDENT_INCR	(<i>'table_or_view'</i>)	The increment value specified during creation of an identity column of a table or view that includes an identity column.
IDENT_SEED	(<i>'table_or_view'</i>)	The seed value specified during creation of an identity column of a table or view that includes an identity column.
INDEX_COL	(<i>'table_name', index_id, key_id</i>)	The indexed column name(s).
ISDATE	(<i>expression_of_possible_date</i>)	Checks whether an expression is a <i>datetime</i> datatype or a string in a recognizable <i>datetime</i> format. Returns 1 when the expression is compatible with the <i>datetime</i> type; otherwise, returns 0.
ISNULL	(<i>expression, value</i>)	Replaces NULL entries with the specified value.
ISNUMERIC	(<i>expression_of_possible_number</i>)	Checks whether an expression is a numeric datatype or a string in a recognizable number format. Returns 1 when the expression is compatible with arithmetic operations; otherwise, returns 0.
NULLIF	(<i>expression1, expression2</i>)	A specialized form of CASE. The resulting expression is NULL when <i>expression1</i> is equivalent to <i>expression2</i> .
OBJECT_ID	(<i>'object_name'</i>)	The database object identification number.
OBJECT_NAME	(<i>object_id</i>)	The database object name.
STATS_DATE	(<i>table_id, index_id</i>)	The date when the statistics for the specified index (<i>index_id</i>) were last updated.
SUSER_SID	(<i>'login_name'</i>)	The security identification number (SID) for the user's login name.
SUSER_SNAME	(<i>[server_user_sid]</i>)	The login identification name from a user's security identification number (SID).
USER_ID	(<i>'user_name'</i>)	The user's database ID number.
USER_NAME	(<i>[user_id]</i>)	The user's database username.

The DATALENGTH function is most often used with variable-length data, especially character strings, and it tells you the actual storage length of the expression (typically a column name). A fixed-length datatype always returns the storage size of its defined type (which is also its actual storage size), which makes it identical to COL_LENGTH for such a column. The DATALENGTH of any NULL expression returns NULL. The OBJECT_ID, OBJECT_NAME, SUSER_SNAME, SUSER_SID, USER_ID, USER_NAME, COL_NAME, DB_ID, and DB_NAME functions are commonly used to more easily eliminate the need for joins between system catalogs and to get run-time information dynamically.

Recognizing that an object name is unique within a database for a specific user, we can use the following statement to determine whether an object by the name "foo" exists for our current user ID; if so, we can drop the object (assuming that we know it's a table).

```
IF (SELECT id FROM sysobjects WHERE id=OBJECT_ID('foo')
    AND uid=USER_ID() AND type='U') > 0
    DROP TABLE foo
```

System functions can be handy with constraints and views. Recall that a view can benefit from the use of system functions. For example, if the *accounts* table includes a column that is the system login ID of the user to whom the account belongs, we can create a view of the table that allows the user to work only with his or her own accounts. We do this simply by making the WHERE clause of the view something like this:

```
WHERE system_login_id=SUSER_SID()
```

Or if we want to ensure that updates on a table occur only from an application named CS_UPDATE.EXE, we can use a CHECK constraint in the following way:

```
CONSTRAINT APP_CHECK(APP_NAME()='CS_UPDATE.EXE')
```

For this particular example, the check is by no means foolproof because the application must set the *app_name* in its login record. A rogue application could simply lie. To prevent casual use by someone running an application like *isql.exe*, the preceding constraint might be just fine for your needs. But if you're worried about hackers, this formulation isn't appropriate. Other functions, such as *SUSER_SID*, have no analogous way to be explicitly set, so they're better for security operations like this.

The *ISDATE* and *ISNUMERIC* functions can be useful for determining whether data is appropriate for an operation. For example, suppose your predecessor didn't know much about using a relational database and defined a column as type *char* and then stored values in it that were naturally of type *money*. Then she compounded the problem by sometimes encoding letter codes in the same field as notes. You're trying to work with the data as is (eventually you'll clean it up, but you have a deadline to meet), and you need to get a sum and average of all the values whenever a value that makes sense to use is present. (If you think this example is contrived, talk to a programmer in an MIS shop of a Fortune 1000 company with legacy applications.) Suppose the table has a *varchar(20)* column named *acct_bal* with these values:

```
acct_bal
-----
205.45
E
(NULL)
B
605.78
32.45
8
98.45
64.23
8456.3
```

If you try to simply use the aggregate functions directly, you'll get error 409:

```
SELECT SUM(acct_bal). AVG(acct_bal)
FROM bad_column_for_money
```

MSG 235, Level 16, State 0

Cannot covert CHAR value to MONEY. The CHAR value has incorrect syntax.

But if you use both CONVERT() in the select list and ISNUMERIC() in the WHERE clause to choose only those values for which a conversion would be possible, everything works great:

```
SELECT
"SUM"=SUM(CONVERT(money, acct_bal)),
"AVG"=AVG(CONVERT(money, acct_bal))
FROM bad_column_for_money
WHERE ISNUMERIC(acct_bal)=1
SUM    AVG
-----
9,470  1.352.95
```

Metadata Functions

In earlier versions of SQL Server, the only way to determine which options or properties an object had was to actually query a system table and possibly even decode a bitmap *status* column. SQL Server 2000 provides a set of functions that return information about databases, files, filegroups, indexes, objects, and datatypes.

For example, to determine what recovery model a database is using, you can use the DATABASEPROPERTYEX function:

```
IF DATABASEPROPERTYEX('pubs', 'Recovery') <> 'SIMPLE'
/*run a command to backup the transaction log */
```

Many of the properties that you can check for using either the DATABASEPROPERTY and DATABASEPROPERTYEX functions correspond to database options that you can set using the *sp_dboption* stored procedure. One example is the AUTO_CLOSE option, which determines the value of the IsAutoClose property. Other properties, such as IsInStandBy, are set internally by SQL Server. The documentation also provides the complete list of properties and possible values for use with the following functions:

- COLUMNPROPERTY
- FILEGROUPPROPERTY
- FILEPROPERTY
- INDEXPROPERTY
- OBJECTPROPERTY
- TYPEPROPERTY

Niladic Functions

ANSI SQL-92 has a handful of what it calls *niladic* functions—a fancy name for functions that don't accept any parameters. Niladic functions were first implemented in version 6, and each one maps directly to one of SQL Server's system functions. They were actually added to SQL Server 6 for conformance with the ANSI SQL-92 standard; all of their functionality was already provided, however. Table 20 lists the niladic functions and their equivalent SQL Server system functions.

Table 20. Niladic functions and their equivalent SQL Server functions.

<i>Niladic Function</i>	<i>Equivalent SQL Server System Function</i>
CURRENT_TIMESTAMP	GETDATE
SYSTEM_USER	SUSER_SNAME
CURRENT_USER	USER_NAME

```
SESSION_USER      USER_NAME
USER              USER_NAME
```

If you execute the following two SELECT statements, you'll see that they return identical results:

```
SELECT CURRENT_TIMESTAMP, USER, SYSTEM_USER, CURRENT_USER,
SESSION_USER
```

```
SELECT GETDATE(), USER_NAME(), SUSER_SNAME(),
USER_NAME(), USER_NAME()
```

Other Parameterless Functions

However, these values aren't variables because you can't declare them and you can't assign them values. These functions are global only in the sense that any connection can access their values. However, in many cases the value returned by these functions is specific to the connection. For example, @@ERROR represents the last error number generated for a specific connection, not the last error number in the entire system. @@ROWCOUNT represents the number of rows selected or affected by the last statement for the current connection.

Many of these parameterless system functions keep track of performance-monitoring information for your SQL Server. These include @@CPU_BUSY, @@IO_BUSY, @@PACK_SENT, and @@PACK_RECEIVED.

Some of these functions are extremely static, and others are extremely volatile. For example, @@version represents the build number and code freeze date of the SQL Server executable (Sqlservr.exe) that you're running. It will change only when you upgrade or apply a service pack. Functions like @@ROWCOUNT are extremely volatile. Take a look at the following code and its results.

```
pub_id pub_name          city          state country
-----
0736 New Moon Books     Boston       MA USA
0877 Binnet & Hardley   Washington   DC USA
1389 Algodata Infosystems Berkeley     CA USA
1622 Five Lakes Publishing Chicago      IL USA
1756 Ramona Publishers  Dallas      TX USA
9901 GGG&G             München     NULL Germany
9952 Scootney Books    New York    NY USA
9999 Lucerne Publishing Paris        NULL France
```

(8 row(s) affected)

8

(1 row(s) affected)

1

(1 row(s) affected)

Note that the first time @@ROWCOUNT was selected, it returned the number of rows in the *publishers* table (8). The second time @@ROWCOUNT was selected, it returned the number of rows returned by the previous SELECT @@ROWCOUNT (1). Any query you execute that affects rows will change the value of @@ROWCOUNT.

Table-Valued Functions

SQL Server 2000 provides a set of system functions that return tables; because of this, the functions can appear in the FROM clause of a query. To invoke these functions, you must use the special signifier `::` as a prefix to the function name and then omit any database or owner name. Several of these table-valued functions return information about traces you have defined. Another one of the functions can also be useful during monitoring and tuning, as shown in this example:

```
SELECT *
FROM ::fn_virtualfilestats(5,1)
```

The function `fn_virtualfilestats` takes two parameters, a database ID and a file ID, and it returns statistical information about the I/O on the file. Here is some sample output from running this query on the `pubs` database:

	Number	Number	Bytes	Bytes			
DblId	FileId	TimeStamps	Reads	Writes	Read	Written	IoStallMS
5	1	38140863	44	0	360448	0	3164

Because the function returns a table, you can control the result set by limiting it to only certain columns or to rows that meet certain conditions. As another example, the function `fn_helpcollations` lists all the collations supported by SQL Server 2000. An unqualified SELECT from this function would return 753 rows in two columns.

```
SELECT name
FROM ::fn_helpcollations()
WHERE name LIKE 'SQL%latin%CI%'
      AND name NOT LIKE '%pref%'
```

RESULT:

```
name
-----
SQL_Latin1_General_CP1_CI_AI
SQL_Latin1_General_CP1_CI_AS
SQL_Latin1_General_CP1250_CI_AS
SQL_Latin1_General_CP1251_CI_AS
SQL_Latin1_General_CP1253_CI_AI
SQL_Latin1_General_CP1253_CI_AS
SQL_Latin1_General_CP1254_CI_AS
SQL_Latin1_General_CP1255_CI_AS
SQL_Latin1_General_CP1256_CI_AS
SQL_Latin1_General_CP1257_CI_AS
SQL_Latin1_General_CP437_CI_AI
SQL_Latin1_General_CP437_CI_AS
SQL_Latin1_General_CP850_CI_AI
SQL_Latin1_General_CP850_CI_AS
```