

Relational Database Technology

As computer systems built on top of database software grew, so did the problems, most of which came to be centered around reorganization and navigation.

Reorganization became a major problem, because with the early hierarchical database systems, the physical data structures were closely tied to the logical data model. Hierarchical data relationships were typically implemented through the use of pointers consisting of actual disk addresses that were stored in physical database records. The database software typically caused a data field for these pointers to be inserted into each application program that accessed the database. Unfortunately, this implied that any change to the record layout by the database software required, at a minimum, recompilation of all programs. Worse yet, if the physical address of record changed, all references to it in the database had to be found so that the correct new location could be inserted. As the size and complexity of databases grew, so grew the problems associated with physical reorganization. With large databases it would be a big problem.

Navigation, the second significant source of problems, refers to the way that applications are bound to the database software. Application programs use commands supported by the database software to find a particular logical record and then use other commands to access related records. For real database software implementation, a program might make an access request for a particular Order record. Subsequent requests might access all the Line Item records that are related to the Order record. In this way, application programs make explicit requests to navigate through the records stored in the database.

A requirement for explicit navigation means the application program has to have incorporated within it knowledge of the data relationships that are documented in the logical data model. However, since the logical description of the relationships and their physical implementation were close, the application also contained implicit knowledge of the implementation. This way potentially confusing. Major changes or extensions to the logical data model could make it necessary to navigate through the database differently, thus invalidating many application programs. In such a situation, recompilation alone would no longer cure the problem. Application programs often had to be rewritten to accommodate a new logical data model. In this environment, maintenance activities began to take precedence over new development, as systems became more and more expensive and time consuming to maintain.

Relational databases systems had several important characteristics that solved many of the problems associated with the use of hierarchical or another databases. The fundamental technology behind modern database is generally based on the **relational data model**, first described in an article published in 1970 by Dr. E.F. Codd. The relational model has a formal mathematical foundation but at the same time defines a table-oriented view of data that is intuitive even to nontechnical end users. The relational data model also formally defines a set of operators that can be applied to the data in relational database. An important aspect of the relational operations is that each operation uses tabular data as input and produces output that also consists of tables.

The relational data model defines three fundamental constructs: *the table*, *the column*, and *the row*.

Tables

With the relational data model, data is stored in a *relational database*, which we will refer to simply as a *database*. A database is represented by, and perceived by its users as, a set of *tables* that contain *row* and

columns. A table represents an *entity type* – the type of person, object, or concept about which we are storing information. A column represents an *attribute* of the entity type – one of information that we are storing about the entity. A row represents a particular *entity occurrence*, or *entity instance*. A particular row contains a set of *data element values*, one for each column in the table.

Table properties

The tables that make up a relational database have certain properties. The properties of the tables stored in a relational database are as follows:

- Each row-column entry in a table consists of a single, or **atomic**, data element.
- All the data elements in a given column are of the same type. The relational data model specifies that for each column, a **domain** must be defined that describes the set of values that are allowed for the data elements in that column. All relational database systems allow the user to specify a particular data type for each column.
- Each column in a table has a unique name within that table.
- The relational data model specifies that all the rows in a table must be unique. Although it is possible with most relational database systems to impose this condition on a table, uniqueness is not always required. Most relational database systems allow tables to contain duplicate rows.
- In the relational data model, the sequence of the rows and columns in the table is not meaningful; the rows and the columns can be viewed in any sequence without affecting either the information content or the semantics of any function that uses the table.

Keys

As discussed earlier, one of the properties of a table, as defined by the relational data model, is that each row must be unique. This means there must be a column, or a set of columns, that uniquely identifies each row. This column, or set of columns, is called the table's **primary key**. For some tables a single column can be found that uniquely identifies each row. Two or more columns that are combined to serve as a primary key are called a **concatenated key**.

The primary key of a table has two important properties:

- **Unique identification.** The value of primary key in each row must be different from the value of the primary key in any other row.
- **Nonredundancy.** If a primary key consists of more than one column, then none of the columns that make up the primary key can be discarded without destroying the property of unique identification.

Integrity Constraints

The relational data model defines two types of integrity constraints that concern the data element values that can be placed in the tables. The term **integrity** refers to the *accuracy* or *correctness* of data in the database. There are two integrity constraints in the relational data model, which are called **entity integrity** and **referential integrity**.

Entity integrity

The *entity integrity* rule states that no column that is part of primary key can have a null value. This rule is necessary if the primary key is to fulfill its role of uniquely identifying the rows in the table. If we allowed a primary key value to be completely *null*, we would be saying that there is some particular entity occurrence that could not be distinguished from other entity occurrences. This is a contradiction,

since two entity occurrences that cannot be distinguished from one another must be the same occurrence. Similar arguments can be made for disallowing partial key values.

Most relational database systems implement the entity integrity rule. Many systems implement primary keys by objects called *indexes*. Such a database package may ensure entity integrity for a given table only if a unique index is defined for that table. If a table is defined with no unique index associated with it, the entity integrity rule may not be enforced for that table.

Referential integrity.

It is possible for one table to contain a column, or set of columns, that contain data element value drawn from the same domain as the column or columns that form the primary key in some other table. This column or set of columns is called a *foreign key*.

In the example shown in Figure 1, the *DeptManager* column in the department table is a foreign key because each *DeptManager* value is drawn from the same domain as the *Emp_#* values in the *Employee* table. In this case, the domain is the set of valid employee numbers.

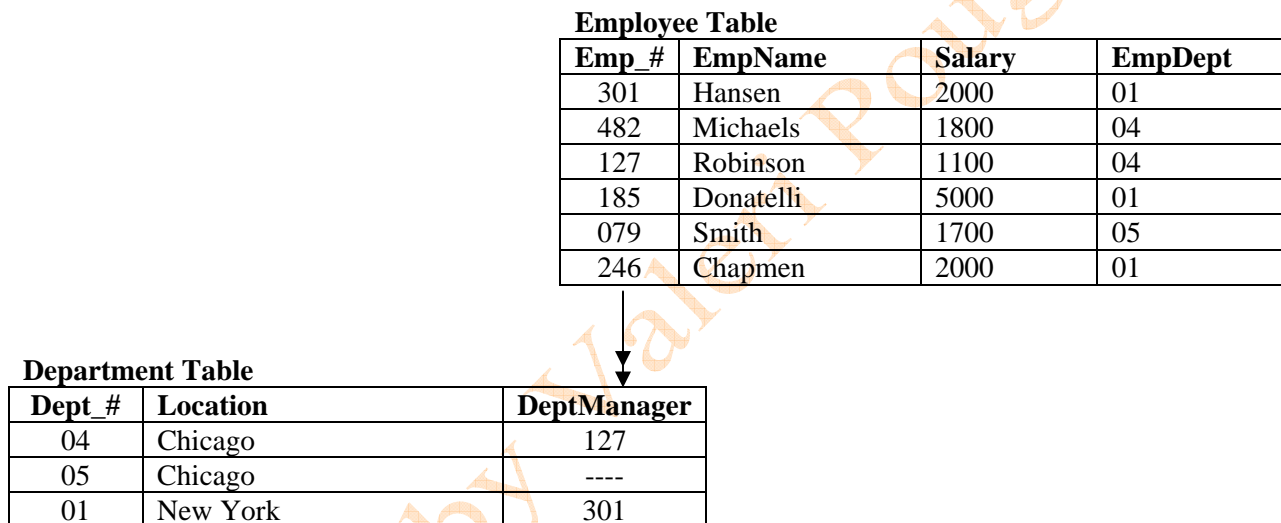


Figure 1 The *DeptManager* column is a foreign key

The referential integrity rule states that every foreign key value must either match a primary key value in its associated table, or it must be wholly null. In other words, any value of *DeptManager* in the department table must either be null or it must match an *Emp_#* value in the *Employee* table. The referential integrity rule guarantees that the *Department* table will not reference a manager who is not also an employee. Allowing a null value in the foreign key, however, does allow the *Department* table to contain a row for a department that currently has no manager.

1.3.1 Normalized Data Structures.

As described above, each row-column entry in a table consists of an atomic data element; a single table entry cannot store a repeating data element or a repeating group of data elements. A data structure from which repeating data elements or repeating groups have been removed in order to place the data into form is called a *normalized data structure*.

The overall goals of the normalization process are to:

- Arrange data so that it can be represented in tables, where each row-column position contains a single data element (no repeating data elements or groups).
- Ensure that data elements are associated with the correct keys, thereby minimizing data redundancy and increasing stability.

Normalization involves a series of steps that change the column structure of the various tables that make up a relational database by placing data into a series of different forms called *first normal form*, *second normal form*, and so on.

First Normal Form

The first step in the normalization process is to place the data into *first normal form*. This process involves the removal of repeating groups. ***A table that contains no repeating groups and can be represented, as a tabular data structure is a normalized data structure and is it at first normal form.*** Any relational database table is automatically in first normal form, since each item in a table must be a single data element.

We can remove repeating groups from a data structure by simply creating a separate row for each of the elements in the repeating group. Suppose we begin with the employee data shown in Figure 2

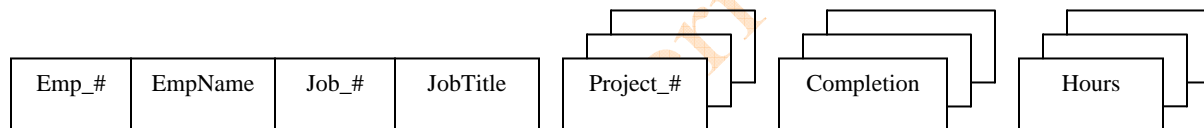


Figure 2 Unnormalized data structure with a repeating group.

Here, a given record stores information about a number of different projects that a particular employee has worked on. In order to represent this data in tabular form, we generate a row for each project on which an employee has worked by repeating the employee information in each row (see Figure 3).

Employee Table

Emp_#	EmpName	Job_#	JobTitle	Project_#	Completion	Hours
120	Jones	1	Programmer	01	7/17	37
120	Jones	1	Programmer	08	1/12	12
121	Harpo	1	Programmer	01	7/17	45
121	Harpo	1	Programmer	08	1/12	21
121	Harpo	1	Programmer	12	3/21	107
270	Garfunkel	2	Analyst	08	1/12	10
270	Garfunkel	2	Analyst	12	3/21	78
273	Selsi	3	Designer	01	7/17	22
274	Abrahms	2	Analyst	12	3/21	41
279	Higgins	1	Programmer	01	7/17	27
279	Higgins	1	Programmer	08	1/12	20
279	Higgins	1	Programmer	12	3/21	51
301	Flannel	1	Programmer	01	7/17	16
301	Flannel	1	Programmer	12	3/21	85

306	McGraw	3	Designer	12	3/21	67
-----	--------	---	----------	----	------	----

Figure 3 Data structure in first normal form.

Earlier we noted that the relational data model requires that we define a primary key for each table, and we will assume here that we are following this principle in defining relational database tables. Both the *Emp_#* and *Project_#* columns are needed to uniquely identify each row, so *Emp_#* and *Project_#* comprise a concatenated key for the table. This is shown in Figure 4.

Employee Table

Emp_#	Project_#	EmpName	Job_#	JobTitle	Completion	Hours
120	01	Jones	1	Programmer	7/17	37
120	08	Jones	1	Programmer	1/12	12
121	01	Harpo	1	Programmer	7/17	45
121	08	Harpo	1	Programmer	1/12	21
121	12	Harpo	1	Programmer	3/21	107
270	08	Garfunkel	2	Analyst	1/12	10
270	12	Garfunkel	2	Analyst	3/21	78
273	01	Selsi	3	Designer	7/17	22
274	12	Abrahms	2	Analyst	3/21	41
279	01	Higgins	1	Programmer	7/17	27
279	08	Higgins	1	Programmer	1/12	20
279	12	Higgins	1	Programmer	3/21	51
301	01	Flannel	1	Programmer	7/17	16
301	12	Flannel	1	Programmer	3/21	85
306	12	McGraw	3	Designer	3/21	67

Figure 4 *Emp_#* and *Project_#* make up the concatenated key of the *Employee* table

We have said that one of the goals of the normalization process is to reduce data redundancy. In fact, we have actually data redundancy by placing the data into first normal form. *Job_#* and *JobTitle* data elements are repeated many times, and the same *Completion* values are stored multiple times as well. This increase in data redundancy is an important for why further normalization steps are required in order to produce a stable design.

Second Normal Form

The second step in the normalization process places our data into *second normal form*. Second normal form involves the idea of *functional dependence*.

Definition of functional dependence:

We say that column B, is functionally dependent on some other column, say column A, if for any given value of column A there is a single value of column B associated with it.

Saying that column B is functionally dependent on the column A is equivalent to say that column A identifies column B. Notice in our table, shown in Figure 4, that there are three rows that have an *Emp_#* of 121, but in each of those rows, the *EmpName* data element value is the same – *Harpo*.

A similar relationship exists between the *Emp_#* and *EmpName* columns in the other rows that have same *Emp_#* value. Therefore, as long as we assume that no two employees can have the same employee number, *EmpName* is functionally dependent on *Emp_#*. We can use the same argument to show that *Job_#* and *JobTitle* are also functionally dependent on *Emp_#*.

In order to place a group of columns that are in first normal form into second normal form, we identify all the full functional dependencies that exist and create a separate table for each set of these. We begin by identifying a likely key – in this case *Emp_#* - and determining which other columns are fully functionally dependent on that key. *EmpName*, *Job_#*, and *JobTitle* are functionally dependent on *Emp_#*, so we leave them in the *Employee* table, which has *Emp_#* as its primary key (See Figure 5)

Employee Table			
Emp_#	EmpName	Job_#	JobTitle
120	Jones	1	Programmer
121	Harpo	1	Programmer
270	Garfunkel	2	Analyst
273	Selsi	3	Designer
274	Abrahms	2	Analyst
279	Higgins	1	Programmer
301	Flannel	1	Programmer
306	McGraw	3	Designer

Figure 5 The columns that are functional dependent on the *Emp_#* column form an *Employee* table.

Of the remaining columns, *Completion* is dependent on the *Project_#*, so we move the *Project_#* and *Completion* columns to a separate *Project* table that has *Project_#* as the primary key, as shown in Figure 6.

Project Table	
Project_#	Completion
01	7/17
08	1/12
12	3/21

Figure 6 The columns those are functionally dependent on the *Project_#* key form a *Project* table.

This leaves the *Hours* column. *Hours* is fully functional dependent on the concatenated key that consists of *Emp_#* and *Project_#*. So we create a third table, called *Hours*, that consists of *Emp_#*, *Project_#*, and *Hours*. The key of the *Hours* table consists of *Emp_#* and *Project_#* (see Figure 7).

Hours Table

Emp_#	Project_#	Hours
120	01	37
120	08	12
121	01	45
121	08	21
121	12	107
270	08	10
270	12	78
273	01	22
274	12	41
279	01	27
279	08	20
279	12	51
301	01	16
301	12	85
306	12	67

Figure 7 The columns that are functionally dependent on the *Emp_#* and *Project_#* concatenated key forms an *Hours* table.

We can now say that the new set of three tables represents a data structure that is in second normal form. All three tables are in first normal form, and every non-key column is fully functionally dependent on the primary key of its table.

Employee Table

Emp_#	EmpName	Job_#	JobTitle
120	Jones	1	Programmer
121	Harpo	1	Programmer
270	Garfunkel	2	Analyst
273	Selsi	3	Designer
274	Abrahams	2	Analyst
279	Higgins	1	Programmer
301	Flannel	1	Programmer
306	McGraw	3	Designer

Hours Table

Emp_#	Project_#	Hours
120	01	37
120	08	12
121	01	45
121	08	21
121	12	107
270	08	10
270	12	78
273	01	22
274	12	41
279	01	27
279	08	20
279	12	51
301	01	16
301	12	85
306	12	67

Project Table

Project_#	Completion
01	7/17
08	1/12
12	3/21

Figure 8 Employee data in second normal form.

Notice in Figure 8 that we have included the *Emp_#* column in two of the tables and that we have included *Project_#* column in two of the tables. Duplicating columns is perfectly valid and often occurs when converting a group of columns to second normal form. By duplicating columns in multiple

tables we are able to use relational operations to combine the tables in various ways to extract the data we need from them.

Second Normal Form Advantages

Notice that placing the data into the second normal form has reduced much of the value redundancy that existed in the original table. *Job_#* and *JobTitle* data elements values now appear only per employee, and *Completion* data element values appear only once per project. Now that we have our data in second normal form, we can point out another understandable characteristic of a table that is in first normal form only.

When our data was in the first normal form only, we had a single table whose primary key consisted of *Emp_#* and *Project_#*. Because of the entity integrity constraint, we need to have valid values for both *Emp_#* and *Project_#* in order to create a new row. This means that we would be unable to store a row for an employee that is currently not assigned to a project (null *Project_#* value). Similarly, we would be prevented from storing a row for a project that currently has no employees assigned to it (null *Emp_#* value).

With our columns in second normal form, we can add a new employee by adding a row to the *Employee* table without having to change either of the other two tables. After the new employee has been assigned to a project and has logged some time on that project, we can add a row to the *Hours* table to describe the project assignment and the number of hours worked. In a similar manner, we can add a row to the *Project* table to describe a new project without changing either the *Employee* or *Hours* tables. Both of these functions would be impossible if our tables were not in the second normal form.

Third Normal Form

Definition of Third Normal Form:

A table is in third normal form if all columns of it are functionally dependent on the key, the whole key, and nothing but the key.

If we check each of tables above (See Figure 8) we can note then in the table "*Employee Table*" column "*JobTitle*" and "*Job_#*" are not functionally depends from *Emp_#* key, because there are four rows with *Emp_#* equal 120, 121, 279, and 301 have a value "*Programmer*" in column "*JobTitle*".

In order to place the table "*Employee Table*" into third normal form we must remove this information in separate table, which we names "*Job Table*". Now we have *Employee* data in the third normal form (see Figure 9)

Employee Table

Emp_#	EmpName	Job_#
120	Jones	1
121	Harpo	1
270	Garfunkel	2
273	Selsi	3
274	Abrahms	2
279	Higgins	1
301	Flannel	1
306	McGraw	3

Hours Table

Emp_#	Project_#	Hours
120	01	37
120	08	12
121	01	45
121	08	21
121	12	107
270	08	10
270	12	78
273	01	22
274	12	41
279	01	27
279	08	20
279	12	51
301	01	16
301	12	85
306	12	67

Project Table	
Project_#	Completion
01	7/17
08	1/12
12	3/21

Job Table	
Job_#	JobTitle
1	Programmer
2	Designer
3	Analyst

Figure 9 Employee data in the third normal form.

In most cases, ensuring that a relational data structure is in third normal form is enough for database design.

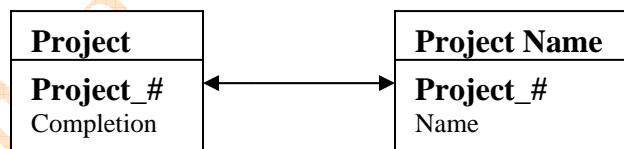
We will consider result of database design as diagrams (See Figure 11), which show the relationships between normalized tables (for the key's columns we use a bold font.).

In this Figure arrow “ \longleftrightarrow ” means relationships (one – to many). For example, this type of relationships exists between tables “*Employee*” and “*Hours*”, which means that each employee may have any projects.

It is very convenient to use graphical design for describing relationships between tables.

There are another relationships arrows for describing structure of the tables:

- “ \longleftrightarrow ” – arrows which define type of relationships “one – to one”. As an example of that type of relations between tables we can consider additional table “*Project Name*”, and it relationships with table “*Project*” (see below)



It is convenient to have this additional table “*Project Name*” for describing name of project, because in all of tables it is enough to keep just code of project and keep the name in on only one table.

- “ \longleftrightarrow ” arrows, which define type of relationships “many to many”. This type of relationships would be in our model, if we wouldn't define the table “*Hours*”. In this case relationships between tables “*Employee*” and “*Project*” have type “*many to many*”. It means each employee can participate in many projects, and each project can develop by many employees. These relationships we can see in the Figure 10.

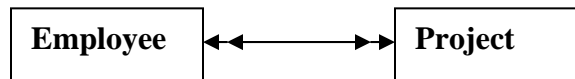


Figure 10 Relationships “many to many”

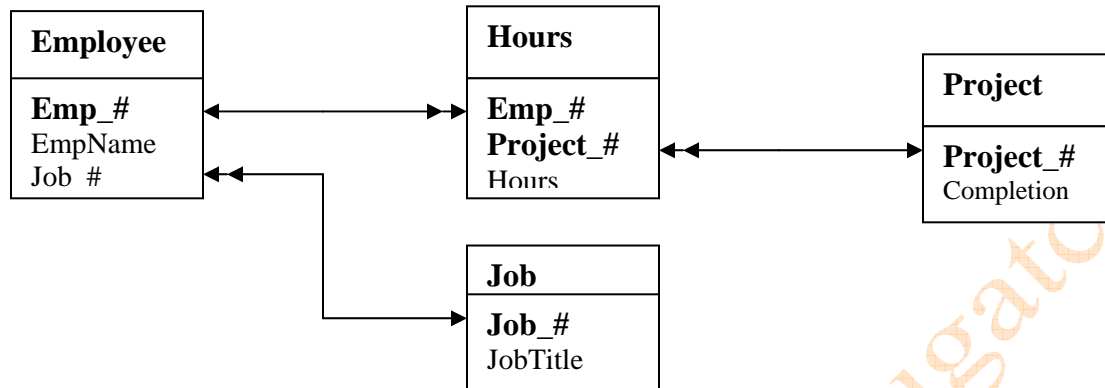


Figure 11 Relationships between tables.

SQL Stored Procedures

A stored procedure is a precompiled collection of Transact-SQL statements stored under a name and processed as a unit. It means that stored procedure is a group of Transact-SQL statements compiled into a single execution plan.

When you create an application with Microsoft® SQL Server™ 2005, the Transact-SQL programming language is the primary programming interface between your applications and the SQL Server database. When you use Transact-SQL programs, two methods are available for storing and executing the programs. You can store the programs locally and create applications that send the commands to SQL Server and process the results, or you can store the programs as stored procedures in SQL Server and create applications that execute the stored procedures and process the results.

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

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

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

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

- ***They allow modular programming.***
You can create the procedure once, store it in the database, and call it any number of times in your program. Stored procedures can be created by a person who specializes in database programming, and they can be modified independently of the program source code.
- ***They allow faster execution.***
If the operation requires a large amount of Transact-SQL code or is performed repetitively, stored procedures can be faster than batches of Transact-SQL code. They are parsed and optimized when they are created, and an in-memory version of the procedure can be used after the procedure is executed the first time. Transact-SQL statements repeatedly sent *FROM* the client each time they run are compiled and optimized every time they are executed by SQL Server.
- ***They can reduce network traffic.***
An operation requiring hundreds of lines of Transact-SQL code can be performed through a single statement that executes the code in a procedure, rather than by sending hundreds of lines of code over the network.
- ***They can be used as a security mechanism.***
Users can be granted permission to execute a stored procedure even if they do not have permission to execute the procedure's statements directly.

A SQL Server stored procedure is created with the Transact-SQL *CREATE PROCEDURE*. The stored procedure definition contains two primary components: the specification of the procedure name and its parameters, and the body of the procedure, which contains Transact-SQL statements that perform the procedure's operations.

Microsoft® SQL Server™ 2005 stored procedures *RETURN* data in four ways:

- Output parameters, which can *RETURN* either data (such as an integer or character value) or a cursor variable (cursors are result sets that can be retrieved one row at a time).
- *RETURN* codes, which are always an integer value.
- A result set for each *SELECT* statement contained in the stored procedure or any other stored procedures called by the stored procedure.
- A global cursor that can be referenced outside the stored procedure.

Stored procedures assist in achieving a consistent implementation of logic across applications. The SQL statements and logic needed to perform a commonly performed task can be designed, coded, and tested once in a stored procedure. Each application needing to perform that task can then simply execute the stored procedure. Coding business logic into a single stored procedure also offers a single point of control for ensuring that business rules are correctly enforced.

Stored procedures can also improve performance. Many tasks are implemented as a series of SQL statements. Conditional logic applied to the results of the first SQL statements determines which subsequent SQL statements are executed. If these SQL statements and conditional logic are written into a stored procedure, they become part of a single execution plan on the server. The results do not have to be returned to the client to have the conditional logic applied; all of the work is done on the server. The

IF statement in this example shows embedding conditional logic in a procedure to keep *FROM* sending a result set to the application:

```
IF (@QuantityOrdered < (SELECT QuantityOnHand
                        FROM Inventory
                        WHERE PartID = @PartOrdered))

    BEGIN
        -- SQL statements to update tables and process order.
    END
ELSE
    BEGIN
        -- SELECT statement to retrieve the IDs of alternate items to suggest as replacements to the customer.
    END
```

Applications do not need to transmit all of the SQL statements in the procedure: they have to transmit only an *EXECUTE* or *CALL* statement containing the name of the procedure and the values of the parameters.

Stored procedures can also shield users from needing to know the details of the tables in the database. If a set of stored procedures supports all of the business functions users need to perform, users never need to access the tables directly; they can just execute the stored procedures that model the business processes with which they are familiar.

SQL Server includes a set of system stored procedures whose names usually start with **sp_**. These system stored procedures support all of the administrative tasks required to run a SQL Server system.

An example of a basic stored procedure

USE Northwind

```
GO
CREATE PROC spShippers
AS
    SELECT * FROM Shippers
```

Other examples of stored procedure

USE Northwind

```
GO
DROP PROCEDURE OrderSummary
GO
CREATE PROCEDURE OrderSummary
    @MaxQuantity INT OUTPUT
AS
    -- SELECT to RETURN a result set summarizing employee sales.
    SELECT Ord.EmployeeID, SummSales = SUM(OrDet.UnitPrice * OrDet.Quantity)
    FROM Orders AS Ord JOIN [Order Details] AS OrDet ON (Ord.OrderID = OrDet.OrderID)
    GROUP BY Ord.EmployeeID
    ORDER BY Ord.EmployeeID
```

```

-- SELECT to fill the output parameter with the maximum quantity FROM Order Details.
SELECT @MaxQuantity = MAX(Quantity) FROM [Order Details]

-- RETURN the number of all items ordered.
RETURN (SELECT SUM(Quantity) FROM [Order Details])

GO

-- Test the stored procedure.

-- DECLARE variables to hold the RETURN code and output parameter.
DECLARE @OrderSum INT
DECLARE @LargestOrder INT

-- Execute the procedure, which RETURNS the result set FROM the first SELECT.
EXEC @OrderSum = OrderSummary @MaxQuantity = @LargestOrder OUTPUT

-- Use the RETURN code and output parameter.
PRINT 'The size of the largest single order was: ' + CONVERT(CHAR(6), @LargestOrder)
PRINT 'The sum of the quantities ordered was: ' + CONVERT(CHAR(6), @OrderSum)

GO

```

The output from running this sample is:

EmployeeID	SummSales
1	202,143.71
2	177,749.26
3	213,051.30
4	250,187.45
5	75,567.75
6	78,198.10
7	141,295.99
8	133,301.03
9	82,964.00

The size of the largest single order was: 130
The sum of the quantities ordered was: 51317

Programming Stored Procedures

Almost any Transact-SQL code that can be written as a batch can be used to create a stored procedure.

Rules for Programming Stored Procedures

Rules for programming stored procedures include:

- The *CREATE PROCEDURE* definition itself can include any number and type of SQL statements except for the following *CREATE* statements, which cannot be used anywhere within a stored procedure:

<i>CREATE DEFAULT</i>	<i>CREATE TRIGGER</i>
<i>CREATE PROCEDURE</i>	<i>CREATE VIEW</i>
<i>CREATE RULE</i>	

- Other database objects can be created within a stored procedure. You can reference an object created in the same stored procedure as long as it is created before it is referenced.
- You can reference temporary tables within a stored procedure.
- If you create a local temporary table inside a stored procedure, the temporary table exists only for the purposes of the stored procedure; it disappears when you exit the stored procedure.
- If you execute a stored procedure that calls another stored procedure, the called stored procedure can access all objects created by the first stored procedure, including temporary tables.
- If you execute a remote stored procedure that makes changes on a remote instance of Microsoft® SQL Server™ 2005, those changes cannot be rolled back. Remote stored procedures do not take part in transactions.
- The maximum number of parameters in a stored procedure is 2100.
- The maximum number of local variables in a stored procedure is limited only by available memory.
- Depending on available memory, the maximum size of a stored procedure is 128 megabytes (MB).

Encrypting Procedure Definitions

If you are creating a stored procedure and you want to make sure that other users cannot view the procedure definition, you can use the *WITH ENCRYPTION* clause. The procedure definition is then stored in an unreadable form.

After a stored procedure is encrypted, its definition cannot be decrypted and cannot be viewed by anyone, including the owner of the stored procedure or the system administrator.

Examples

A. Create a stored procedure that uses parameters

This example creates a stored procedure that is useful in the **pubs** database. Given the last and first name of an author, the stored procedure displays the title and publisher of each book by that author.

```
CREATE PROC au_info
    @lastname varchar(40),
    @firstname varchar(20)
AS
SELECT au_lname,
       au_fname,
       title,
       pub_name
FROM
    authors INNER JOIN titleauthor ON authors.au_id=titleauthor.au_id
    JOIN titles ON titleauthor.title_id = titles.title_id
    JOIN publishers ON titles.pub_id = publishers.pub_id
```



```
WHERE au_fname = @firstname
      AND au_lname = @lastname
GO
```

Now execute the *au_info* stored procedure:

```
EXECUTE au_info Ringer, Anne
GO
```

Here is the result set:

au_lname	au_fname	title	pub_name
-----	-----	-----	-----
Ringer	Anne	The Gourmet Microwave	Binnet & Hardley
Ringer	Anne	Is Anger the Enemy?	New Moon Books

(2 row(s) affected)

B. Create a stored procedure that uses default values for parameters

This example creates a stored procedure, **pub_info2** that displays the names of all authors who have written a book published by the publisher given as a parameter. If no publisher name is supplied, the stored procedure shows the authors published by Algodata Infosystems.

```
CREATE PROC pub_info2
    @pubname varchar(40) = 'Algodata Infosystems'
AS
SELECT au_lname,
       au_fname,
       pub_name
FROM authors a INNER JOIN titleauthor ta ON a.au_id = ta.au_id
JOIN titles t ON ta.title_id = t.title_id
JOIN publishers p ON t.pub_id = p.pub_id
WHERE @pubname = p.pub_name
```

Execute **pub_info2** with no parameter specified:

```
EXECUTE pub_info2
GO
```

Here is the result set:

au_lname	au_fname	pub_name
-----	-----	-----
Green	Marjorie	Algodata Infosystems
Bennet	Abraham	Algodata Infosystems

O'Leary	Michael	Algodata Infosystems
MacFeather	Stearns	Algodata Infosystems
Straight	Dean	Algodata Infosystems
Carson	Cheryl	Algodata Infosystems
Dull	Ann	Algodata Infosystems
Hunter	Sheryl	Algodata Infosystems
Locksley	Charlene	Algodata Infosystems

(9 row(s) affected)

C. Create a stored procedure using a parameter default including wildcard characters

The default can include wildcard characters (*%*, *_* *[]* and *[^]*) if the stored procedure uses the parameter with the *LIKE* keyword. The following variation of the stored procedure **au_info** has defaults with wildcard characters for both parameters:

```
CREATE PROC au_info2
    @lastname varchar(30) = 'D%',
    @firstname varchar(18) = '%'
AS
SELECT au_lname,
       au_fname,
       title,
       pub_name
FROM authors INNER JOIN titleauthor ON authors.au_id = titleauthor.au_id
JOIN titles ON titleauthor.title_id = titles.title_id
JOIN publishers ON titles.pub_id = publishers.pub_id
WHERE au_fname LIKE @firstname
      AND au_lname LIKE @lastname
```

If **au_info2** is executed with no parameters, all the authors with last names beginning with the letter *D* are displayed:

```
EXECUTE au_info2
GO
```

Here is the result set:

au_lname	au_fname	title	pub_name
-----	-----	-----	-----
Dull	Ann	Secrets of Silicon Val	Algodata Infosystems
del Castillo	Innes	Silicon Val Gastrono	Binnet & Hardley
DeFrance	Michel	The Gourmet Microwave	Binnet & Hardley

(3 row(s) affected)

This example omits the second parameter when defaults for two parameters have been defined, so you can find the books and publishers for all authors with the last name **Ringer**:

```
EXECUTE au_info2 Ringer
GO
```

au_lname	au_fname	title	pub_name
-----	-----	-----	-----
Ringer	Anne	The Gourmet Microwave	Binnet & Hardley
Ringer	Anne	Is Anger the Enemy?	New Moon Books
Ringer	Albert	Is Anger the Enemy?	New Moon Books
Ringer	Albert	Life Without Fear	New Moon Books

(4 row(s) affected)

Creating Output Parameters

Sometimes, you want to pass non-recordset information out to whatever called your stored procedure. Let's say, for example, that we are performing an insert into a table, but we are planning to do additional work using the inserted record. Supposed we are inserting a new record into *Order* table in *Northwind*, but we also need to insert detail records in the *Order Details* table. In order to keep the relationship intact, we have to know the identity of the *Order* record before we can do our insert into the *Order Details* table. We will have an output parameter for the identity value that is generated by our insert:

```
USE Northwind
GO
```

```
CREATE PROC spInsertOrder
    @CustomerID    nvarchar(5),
    @EmployeeID    int,
    @OrderDate     datetime    = NULL,
    @RequiredDate  datetime    = NULL,
    @ShippedDate   datetime    = NULL,
    @ShipVia       int,
    @Freight       money,
    @ShipName      nvarchar(40) = NULL,
    @ShipAddress   nvarchar(60) = NULL,
    @ShipCity      nvarchar(15) = NULL,
    @ShipRegion    nvarchar(15) = NULL,
    @ShipPostalCode nvarchar(10) = NULL,
    @Country       nvarchar(15) = NULL,
    @OrderID       int    OUTPUT
```

```
AS
```

```
    /* Create the record */
    INSERT INTO Orders
    VALUES
```

```

(
    @CustomerID,
    @EmployeeID,
    @OrderDate,
    @RequiredDate,
    @ShippedDate,
    @ShipVia,
    @Freight,
    @ShipName,
    @ShipAddress,
    @ShipCity,
    @ShipRegion,
    @ShipPostalCode,
    @Country
)
/* Move the identity value from the newly inserted record into our output variable */
SELECT @OrderID = @@IDENTITY

```

SQL Views

A view can be thought of as either a virtual table or a stored query. The data accessible through a view is not stored in the database as a distinct object. What is stored in the database is a *SELECT* statement. The result set of the *SELECT* statement forms the virtual table returned by the view. A user can use this virtual table by referencing the view name in Transact-SQL statements the same way a table is referenced. A view is used to do any or all of these functions:

- ***Restrict a user to specific rows in a table.***
For example, allow an employee to see only the rows recording his or her work in a labor-tracking table.
- ***Restrict a user to specific columns.***
For example, allow employees who do not work in payroll to see the name, office, work phone, and department columns in an employee table, but do not allow them to see any columns with salary information or personal information.
- ***Join columns from multiple tables so that they look like a single table.***
- ***Aggregate information instead of supplying details.***
For example, present the sum of a column, or the maximum or minimum value from a column.

Views are created by defining the *SELECT* statement that retrieves the data to be presented by the view. The data tables referenced by the *SELECT* statement are known as the base tables for the view. In this example, ***titleview*** in the ***pubs*** database is a view that selects data from three base tables to present a virtual table of commonly needed data:

```

CREATE VIEW titleview
AS
SELECT title,
    au_ord,
    au_lname,

```

```

        price,
        ytd_sales,
        pub_id
FROM authors AS a JOIN titleauthor AS ta ON (a.au_id = ta.au_id)
JOIN titles AS t ON (t.title_id = ta.title_id)

```

You can then reference **titleview** in statements in the same way you would reference a table:

```
SELECT * FROM titleview
```

A view can reference another view. For example, **titleview** presents information that is useful for managers, but a company typically discloses year-to-date figures only in quarterly or annual financial statements. A view can be built that selects all the **titleview** columns except **au_ord** and **ytd_sales**. This new view can be used by customers to get lists of available books without seeing the financial information:

```

CREATE VIEW Cust_titleview
AS
SELECT title,
        au_lname,
        price,
        pub_id
FROM titleview

```

Views in all versions of SQL Server are updatable (can be the target of **UPDATE**, **DELETE**, or **INSERT** statements), as long as the modification affects only one of the base tables referenced by the view, for example:

```

-- Increase the prices for publisher '0736' by 10%.
UPDATE titleview
SET price = price * 1.10
WHERE pub_id = '0736'
GO

```

Triggers

A trigger is a special type of stored procedure that is not called directly by a user. When the trigger is created, it is defined to execute when a specific type of data modification is made against a specific table or column. Triggers are a special class of stored procedure defined to execute automatically when an **UPDATE**, **INSERT**, or **DELETE** statement is issued against a table or view. Triggers are powerful tools that sites can use to enforce their business rules automatically when data is modified. Triggers can extend the integrity checking logic of SQL Server constraints, defaults, and rules.

Tables can have multiple triggers. The **CREATE TRIGGER** statement can be defined with the **FOR UPDATE**, **FOR INSERT**, or **FOR DELETE** clauses to target a trigger to a specific class of data modification actions. When **FOR UPDATE** is specified, the **IF UPDATE (column_name)** clause can be used to target a trigger to updates affecting a particular column.

Triggers can automate the processing for a company. In an inventory system, update triggers can detect when a stock level reaches a reorder point and generate an order to the supplier automatically. In a database recording the processes in a factory, triggers can e-mail or page operators when a process exceeds defined safety limits.

The following trigger generates an e-mail whenever a new title is added in the **pubs** database:

```
CREATE TRIGGER reminder
ON titles
FOR INSERT
AS
EXEC master..xp_sendmail 'MaryM',
    'New title, mention in the next report to distributors.'
```

Triggers contain Transact-SQL statements, much the same as stored procedures. Triggers, like stored procedures, return the result set generated by any *SELECT* statements in the trigger. Including *SELECT* statements in triggers, except statements that only fill parameters, is not recommended. This is because users do not expect to see any result sets returned by an *UPDATE*, *INSERT*, or *DELETE* statement.

You can specify one of two options to control when a trigger fires:

- **AFTER** triggers fire after the triggering action (*INSERT*, *UPDATE*, or *DELETE*) and after any constraints are processed. You can request *AFTER* triggers by specifying either the *AFTER* or *FOR* keywords. Because the *FOR* keyword has the same effect as *AFTER*, triggers with the *FOR* keyword are also classified as *AFTER* triggers.
- **INSTEAD OF** triggers fire in place of the triggering action and before constraints are processed.

Each table or view can have one *INSTEAD OF* trigger for each triggering action (*UPDATE*, *DELETE*, and *INSERT*). A table can have several *AFTER* triggers for each triggering action.

Examples

Testing for Changes to Specific Columns

The *IF UPDATE (column_name)* clause in the definition of a trigger can be used to determine if an *INSERT* or *UPDATE* statement affected a specific column in the table. The clause evaluates to *TRUE* whenever the column is assigned a value.

Note: Because a specific value in a column cannot be deleted using the *DELETE* statement, the *IF UPDATE* clause does not apply to the *DELETE* statement.

Alternatively, the *IF COLUMNS_UPDATED()* clause can be used to check which columns in a table an *INSERT* updated or *UPDATE* statement. This clause uses an integer bitmask to specify the columns to test.

A. Use the *IF UPDATE* clause to test data modifications

This example creates an *INSERT* trigger **my_trig** on table **my_table** and tests whether column **b** was affected by any *INSERT* statements.

```
CREATE TABLE my_table*  
(a int NULL, b int NULL)  
GO
```

```
CREATE TRIGGER my_trig  
ON my_table  
FOR INSERT  
AS  
IF UPDATE(b)  
    PRINT 'Column b Modified'  
GO
```

Appendix

Structured Query Language (SQL).

Contents

- *SELECT* Statement Example 1
- Specifying columns order Example 2
- Displaying selected rows Example 3
- Displaying selected rows (another example) Example 4
- Selection using arithmetic expressions Example 5
- Selection using a list or range Example 6
- Selection using a list or range (another example) Example 7
- Pattern matching Example 8
- Pattern matching (another example) Example 9
- Eliminating duplicate rows Example 10
- Multiple conditions Example 11
- *NULL* values Example 12
- Negative conditions and using the *NOT* operator Example 13
- Generated columns and naming output columns Example 14
- Sequencing rows Example 15
- Sorting on a generated column Example 16
- Built-in functions Example 17
- Grouping rows Example 18
- *INNER JOIN* operations Example 19
- Multiple *INNER JOIN* operations Example 20
- *LEFT OUTER JOIN* operation Example 21
- *RIGHT OUTER JOIN* operation Example 22
- Inserting Individual Rows

- Updating Table Data
- Updating Individual Rows
- Updating Multiple Rows
- Deleting Table Data

The main language used by most client/server database software products is Structured Query Language (SQL). SQL has been standardized, and its structure is documented in an accepted international standard since 1986.

Although SQL has a term *query* in its name, SQL is more than simply a query language. Different subsets of the statements making up the SQL language allow SQL to be used in three different ways:

- **Data Manipulation Language.** The statements making up a *data manipulation language (DML)* allow SQL to be used to retrieve data from a relational database, to update and delete existing data, to insert new data.
- **Data Description Language.** The statements making up a *data description language (DDL)* allow SQL to be used to create, alter, and delete the various objects – including tables, views, domains, and indices – that are used to implement a relational database.
- **Data Control Language.** The statements making up a *data control language (DCL)* allow SQL to be used to perform administrative procedures relating to SQL objects, such as granting authorization to users for their access and establishing synchronization points to control database recovery processing.

It is important to note that not all implementation of SQL provide the same level of support in all three the areas that the full SQL defines. For example, the Microsoft Access implementation of SQL for personal computer environment concentrates on the use of SQL as a data manipulation language. Data description language and data control language functions are provided in Access through graphical user interface functions rather than through SQL. Other database products, such Microsoft SQL-Server, and IBM's DB2 products make explicit use of the DML, DDL, and features of SQL. Below we can see the list of the most commonly used SQL statements that make up the SQL data manipulation language, data description language, and data control language.

SQL Data Manipulation Language Statements

- **SELECT.** Used to specify database retrievals and to create alternative views of the views of the data contained in one or more base tables
- **UPDATE.** Used to add change the value in existing rows in base tables or views.
- **INSERT.** Used to add new rows to tables or views.
- **DELETE.** Used to delete rows from tables or views.

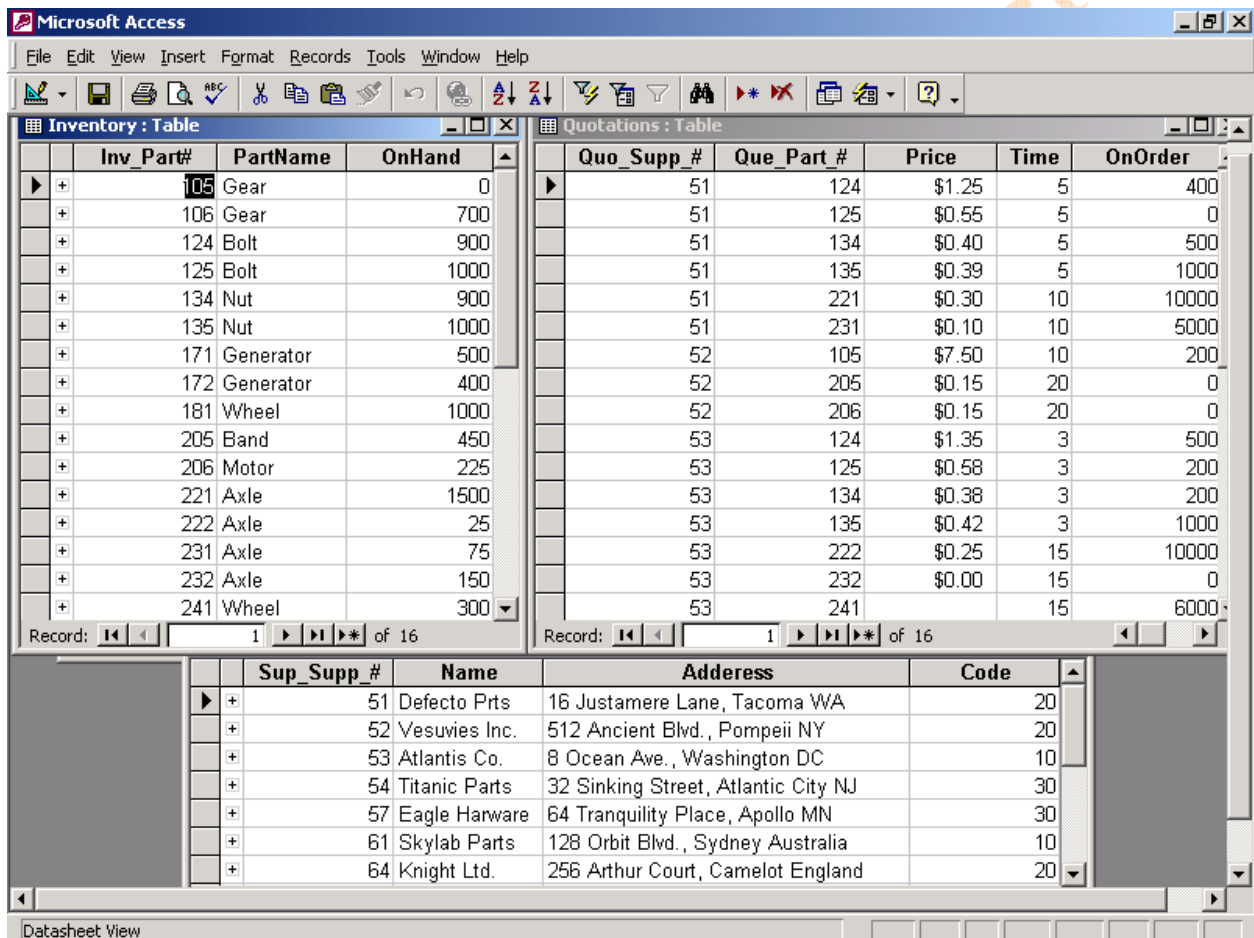
SQL Data Description Language Statements

- **CREATE.** Used to define a new SQL object, such as a table, view, or domain.
- **ALTER.** Used to modify the characteristics of an existing SQL object.
- **DROP.** Used to delete an existing SQL object from the SQL environment.

SQL Data Control Language Statements

- **GRANT.** Used to give users authorization to access an SQL object.
- **REVOKE.** Used to remove a user's authorization to access an SQL object.
- **COMMIT.** Used to establish a synchronization point to indicate that all changes that have been made to SQL objects up to that point are to be made permanent.
- **ROLLBACK.** Used to indicate that changes that have been made to SQL objects up to the most recent synchronization point are to be reversed, thus restoring those objects to the condition in which they existing prior to the establishment of that synchronization point.

We will consider examples of SQL-statements based information of “PartsDepot” database (see Figure 1)



Inv_Part#	PartName	OnHand
105	Gear	0
106	Gear	700
124	Bolt	900
125	Bolt	1000
134	Nut	900
135	Nut	1000
171	Generator	500
172	Generator	400
181	Wheel	1000
205	Band	450
206	Motor	225
221	Axle	1500
222	Axle	25
231	Axle	75
232	Axle	150
241	Wheel	300

Quo_Supp #	Quo_Part #	Price	Time	OnOrder
51	124	\$1.25	5	400
51	125	\$0.55	5	0
51	134	\$0.40	5	500
51	135	\$0.39	5	1000
51	221	\$0.30	10	10000
51	231	\$0.10	10	5000
52	105	\$7.50	10	200
52	205	\$0.15	20	0
52	206	\$0.15	20	0
53	124	\$1.35	3	500
53	125	\$0.58	3	200
53	134	\$0.38	3	200
53	135	\$0.42	3	1000
53	222	\$0.25	15	10000
53	232	\$0.00	15	0
53	241		15	6000

Sup_Supp #	Name	Address	Code
51	Defecto Prts	16 Justamere Lane, Tacoma WA	20
52	Vesuvies Inc.	512 Ancient Blvd., Pompeii NY	20
53	Atlantis Co.	8 Ocean Ave., Washington DC	10
54	Titanic Parts	32 Sinking Street, Atlantic City NJ	30
57	Eagle Harware	64 Tranquility Place, Apollo MN	30
61	Skylab Parts	128 Orbit Blvd., Sydney Australia	10
64	Knight Ltd.	256 Arthur Court, Camelot England	20

Figure 1 Tables of “PartsDepot” database.

Relationships between tables and primary keys we can see in the Figure 2.

The *SELECT* Statement

The basic form of the *SELECT* statements is as follows:

SELECT some data (the name of one or more columns)
FROM some place (the name of table or view)
WHERE conditions (comparisons based on data element values)
ORDER BY desired sequence (the name of one or more columns)

All *SELECT* statements must include a *FROM* clause, but the *WHERE* and *ORDER BY* clauses are optional.

Example 1

SELECT *
FROM Quotations

Result of this query we can see in the Figure 3.

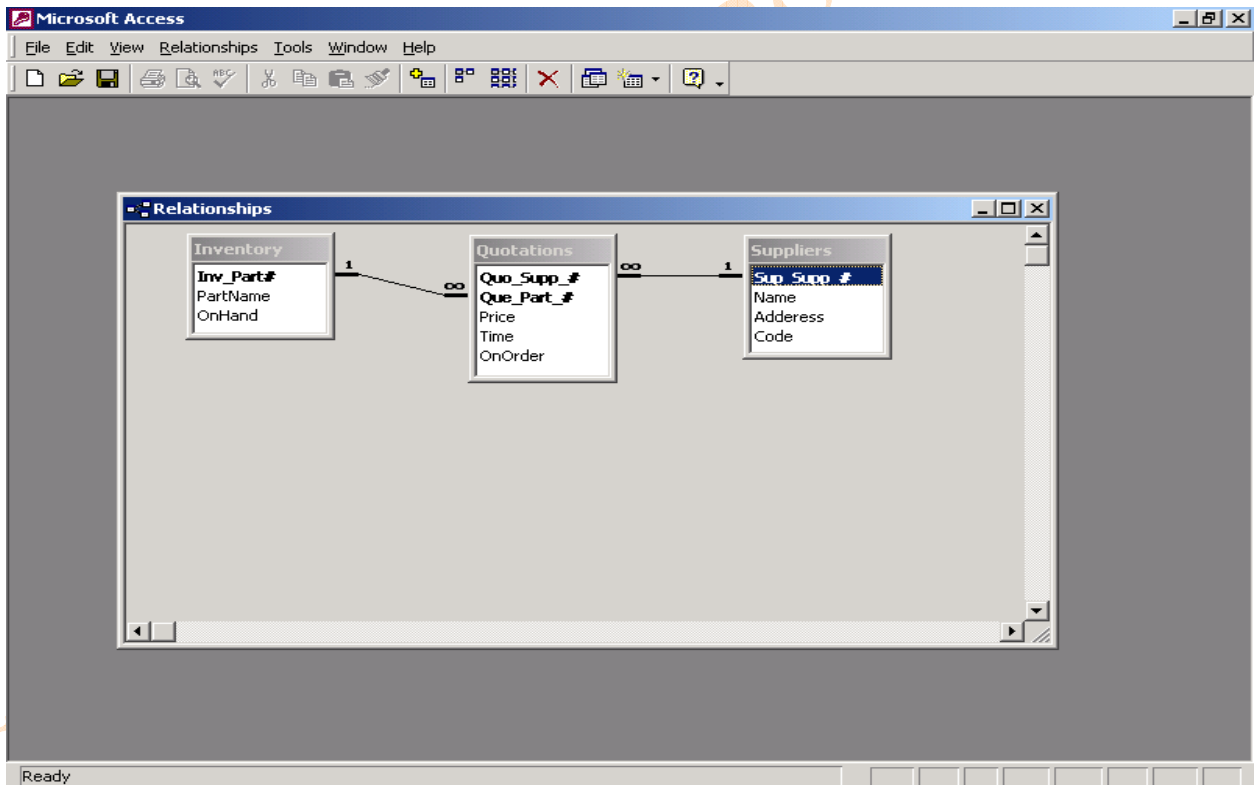


Figure 2 Relationships between tables and Primary Keys of “PartsDepot” database.

Quo_Supp_#	Quo_Part_#	Price	Time	OnOrder
51	124	\$1.25	5	400
51	125	\$0.55	5	0
51	134	\$0.40	5	500

Quo_Supp_#	Quo_Part_#	Price	Time	OnOrder
51	135	\$0.39	5	1000
51	221	\$0.30	10	10000
51	231	\$0.10	10	5000
52	105	\$7.50	10	200
52	205	\$0.15	20	0
52	206	\$0.15	20	0
53	124	\$1.35	3	500
53	125	\$0.58	3	200
53	134	\$0.38	3	200
53	135	\$0.42	3	1000
53	222	\$0.25	15	10000
53	232	\$0.00	15	0
53	241		15	6000

Figure 3 Result of query (Example 1)

Example 2 Specifying columns order

*SELECT Time, [Quo_Part_#], [Quo_Supp_#], OnOrder, Price
FROM Quotations*

For columns *Quo_Part_#* and *Quo_Supp_#* we use square brackets, because the names of these columns (fields) consists special symbol “#”, which is the special symbol for the database.

It means that if we want to use some special symbols for name of field (column) within SQL-statement, we have to use square brackets.

Result of this query:

Time	Quo_Part_#	Quo_Supp_#	OnOrder	Price
5	124	51	400	\$1.25
5	125	51	0	\$0.55
5	134	51	500	\$0.40
5	135	51	1000	\$0.39
10	221	51	10000	\$0.30
10	231	51	5000	\$0.10
10	105	52	200	\$7.50
20	205	52	0	\$0.15
20	206	52	0	\$0.15
3	124	53	500	\$1.35
3	125	53	200	\$0.58
3	134	53	200	\$0.38
3	135	53	1000	\$0.42
15	222	53	10000	\$0.25
15	232	53	0	\$0.00

Time	Quo_Part_#	Quo_Supp_#	OnOrder	Price
15	241	53	6000	

Example 3 Displaying selected rows

```
SELECT *
FROM Quotations
WHERE [Quo_Part_#] = 124
```

Result of this query is below:

Quo_Supp_#	Quo_Part_#	Price	Time	OnOrder
51	124	\$1.25	5	400
53	124	\$1.35	3	500

In the above example, we used an equal comparison ($Quo_Part_# = 124$) as the selection condition. Other types of comparisons can also be specified, including:

- > greater than
- >= greater than or equal to
- < less than
- <= less than or equal
- ≠ not equal to
- <> not equal to
- ¬> not greater than
- ¬< not less than

Example 4 Displaying selected rows (another example)

```
SELECT [Quo_Supp_#], [Quo_Part_#], Price
FROM Quotations
WHERE Price > 1.00
```

Result:

Quo_Supp_#	Quo_Part_#	Price
51	124	\$1.25
52	105	\$7.50
53	124	\$1.35

Example 5 Selection using arithmetic expressions.

We can include arithmetic expressions in *WHERE* selection conditions to perform calculations in condition clauses. Arithmetic operators that we can use in *SELECT* statements are:

- + add

- subtract
 * multiply
 / divide

The following **SELECT** statement displays all supplier and part values where the quotation amount (*Price + OnOrder*) is greater than \$500.00:

```
SELECT [Quo_Supp_#], [Quo_Part_#]
FROM Quotations
WHERE Price*OnOrder > 500.00
```

Result

Quo_Supp_#	Quo_Part_#
51	221
52	105
53	124
53	222

Example 6 Selection using a list or range.

We can use the *IN* and *BETWEEN* keywords to specify a list of value or range of values in a *WHERE* condition. We use the *IN* keyword to specify a list of values in parentheses. Whenever a row contains a data element value that matches one of the values in parentheses the *IN* keyword, the row selected.

```
SELECT [Quo_Supp_#], [Quo_Part_#], Price
FROM Quotations
WHERE [Quo_Part_#] IN (105, 135, 205)
```

Result:

Quo_Supp_#	Quo_Part_#	Price
51	135	\$0.39
52	105	\$7.50
52	205	\$0.15
53	135	\$0.42

Example 7 Selection using a list or range (another example).

```
SELECT [Quo_Supp_#], [Quo_Part_#], Price
FROM Quotations
WHERE [Quo_Part_#] BETWEEN 105 AND 205
```

Result:

Quo_Supp_#	Quo_Part_#	Price
52	105	\$7.50
51	124	\$1.25

Quo_Supp_#	Quo_Part_#	Price
53	124	\$1.35
51	125	\$0.55
53	125	\$0.58
51	134	\$0.40
53	134	\$0.38
51	135	\$0.39
53	135	\$0.42
52	205	\$0.15

Example 8 Pattern matching

We can base a selection on the occurrence of a particular pattern of characters in a data element. This is done using the *LIKE* keyword. Selection using *LIKE* can only be used for columns (fields) that contain character data. We can specify that any number of characters may occur either or after the desired value by using the percent sign (%) as a wildcard character. (Some databases packages use the asterisk (*) character instead of the % wildcard). We might use the following *SELECT* statement to select suppliers located in the state of *Minnesota*:

```
SELECT [Sup_Supp_#], Name, Address
FROM Suppliers
WHERE Address LIKE "*MN*"
```

Result:

Sup_Supp_#	Name	Address
57	Eagle Harware	64 Tranquility Place, Apollo MN

Example 9 Pattern matching (another example)

We can specify the number of characters that should precede or follow the desired value by using an underscore (_) wild-card character in each character position that must precede or follow the pattern character. (Some database software products use the question mark (?) instead of _ wild card.) The following *SELECT* statement could be used to display all parts for which the second character in the part name is "o":

```
SELECT *
FROM Inventory
WHERE PartName LIKE '?o*'
```

Result:

Inv_Part#	PartName	OnHand
124	Bolt	900
125	Bolt	1000

Inv_Part#	PartName	OnHand
206	Motor	225

Example 10 Eliminating duplicate rows.

Depending on the selection criteria we specify and the columns that the database software selects for display, it is possible for the result to contain duplicate rows. For example, the following *SELECT* statement displays suppliers that have quotations with an on-order quantity greater than 200:

```
SELECT [Quo_Supp_#]
FROM Quotations
WHERE OnOrder > 200
```

Result:

Quo_Supp_#
51
51
51
51
51
53
53
53
53

In this example, suppliers 51 and 53 are each listed multiple times in the resulting display, since each has multiple parts for which *OnOrder* value is greater than 200. The *DISTINCT* keyword causes the database software to eliminate duplicate rows from the result:

```
SELECT DISTINCT [Quo_Supp_#]
FROM Quotations
WHERE OnOrder > 200
```

Result:

Quo_Supp_#
51
53

Example 11 Multiple conditions

```
SELECT [Quo_Supp_#], [Quo_Part_#], Price
FROM Quotations
Where [Quo_Part_#] = 124 AND Price > 1.30
```

Result:

Quo_Supp_#	Quo_Part_#	Price
53	124	\$1.35

```
SELECT *  
FROM Quotations  
Where [Quo_Part_#] < 200 AND (Price > 1.00 OR Time < 10)
```

Result:

Quo_Supp_#	Quo_Part_#	Price	Time	OnOrder
51	124	\$1.25	5	400
51	125	\$0.55	5	0
51	134	\$0.40	5	500
51	135	\$0.39	5	1000
52	105	\$7.50	10	200
53	124	\$1.35	3	500
53	125	\$0.58	3	200
53	134	\$0.38	3	200
53	135	\$0.42	3	1000

Example 12 NULL values.

Depending on how a table is defined, a column may be allowed to contain data elements that have a null value. A null value means that no value has been entered for the data element in that row.

```
SELECT *  
FROM Quotations  
WHERE Price IS NULL
```

Result:

Quo_Supp_#	Quo_Part_#	Price	Time	OnOrder
53	241		15	6000

Example 13 Negative conditions and using the *NOT* operator

```
SELECT *  
FROM Quotations  
WHERE [Quo_Part_#] <> 124
```

Result:

Quo_Supp_#	Quo_Part_#	Price	Time	OnOrder
51	125	\$0.55	5	0

Quo_Supp_#	Quo_Part_#	Price	Time	OnOrder
51	134	\$0.40	5	500
51	135	\$0.39	5	1000
51	221	\$0.30	10	10000
51	231	\$0.10	10	5000
52	105	\$7.50	10	200
52	205	\$0.15	20	0
52	206	\$0.15	20	0
53	125	\$0.58	3	200
53	134	\$0.38	3	200
53	135	\$0.42	3	1000
53	222	\$0.25	15	10000
53	232	\$0.00	15	0
53	241		15	6000

```

SELECT *
FROM Quotations
WHERE NOT ([Quo_Part_#] < 200 AND (Price > 1.00 OR Time < 10))

```

Result:

Quo_Supp_#	Quo_Part_#	Price	Time	OnOrder
51	221	\$0.30	10	10000
51	231	\$0.10	10	5000
52	205	\$0.15	20	0
52	206	\$0.15	20	0
53	222	\$0.25	15	10000
53	232	\$0.00	15	0
53	241		15	6000

Example 14 Generated columns and naming output columns.

In addition to displaying columns selected from a table, it is possible to display values that are generated based on calculations performed on values from other columns. We can give a specific name to be used for a column in the output display by including an *AS* clause in the *SELECT* statement.

```

SELECT [Quo_Supp_#],
       [Quo_Part_#],
       Price * OnOrder AS Price_Quote
FROM Quotations
Where Price > 1.00

```

Result:

Quo_Supp_#	Quo_Part_#	Price_Quote
51	124	\$500.00
52	105	\$1,500.00
53	124	\$675.00

Example 15 Sequencing rows.

The rows that are displayed as the result of query are often displayed in the same sequence that they occur within the table from which they are selected. We can use the *ORDER BY* clause in the *SELECT* statement to sequence the rows in the displayed table. For example, the following *SELECT* statement lists the selected rows in ascending part number sequence and in descending price sequence price for each part number:

```
SELECT      [Quo_Part_#], [Quo_Supp_#], Price
FROM Quotations
WHERE       [Quo_Part_#] IN (124, 125, 134, 135)
ORDER BY    [Quo_Part_#], Price DESC
```

Result:

Quo_Part_#	Quo_Supp_#	Price
124	53	\$1.35
124	51	\$1.25
125	53	\$0.58
125	51	\$0.55
134	51	\$0.40
134	53	\$0.38
135	53	\$0.42
135	51	\$0.39

Example 16 Sorting on a generated column.

It is possible to use generated columns as one of the *ORDER BY* columns. However, since a calculated column does not have a name by which it can be generated, we must refer to it by its column number.

Here, since the generated column is the third one specified in the *SELECT* statement, we can refer to it by the number 3 in the *ORDER BY* clause:

```
SELECT [Quo_Supp_#],
       [Quo_Part_#],
       Price*OnOrder AS Price_OnOrder
FROM Quotations
WHERE Price > 1.00
ORDER BY [Quo_Supp_#], 3 DESC
```


Result:

Quo_Supp_#	Quo_Part_#	Price_OnOrder
51	124	\$500.00
52	105	\$1,500.00
53	124	\$675.00

Example 17 Built-in functions

SQL includes a set of built-in functions that can be used with the *SELECT* statement to perform operations on the table or on sets of rows from a table. Some commonly used functions are as follows:

Function	Operation
<i>SUM</i>	Calculate a total
<i>MIN</i>	Calculates the minimum value
<i>MAX</i>	Calculates the maximum value
<i>AVG</i>	Calculates an average value
<i>COUNT(*)</i>	Counts the number of selected rows
<i>COUNT(DISTINCT column name)</i>	Counts unique values within a set of selected rows

```
SELECT SUM(OnOrder) AS SUM_OnOrder,  
       SUM(Price*OnOrder) AS SUM_PriceOnOrder,  
       MAX(Price) AS MAX_Price,  
       AVG(Price) AS AV_Price  
FROM Quotations
```

Result:

SUM_OnOrder	SUM_PriceOnOrder	MAX_Price	AV_Price
35000	\$9,877.00	\$7.50	\$0.92

Example 18 Grouping rows

Built-in functions can be applied to select groups of rows in a table. To do this, we use the *GROUP BY* clause to indicate how rows should be grouped together. Generally, *GROUP BY* produces one row in the resulting table for each different value it finds in the column specified in the *GROUP BY* clause.

The following *SELECT* statement uses a *SUM* function and a *GROUP BY* clause to produce a total of the *OnOrder* data elements values for each different part number:

```
SELECT [Quo_Part_#],  
       SUM(OnOrder) AS SUM_OnOrder  
FROM Quotations  
GROUP BY [Quo_Part_#]  
ORDER BY [Quo_Part_#]
```

Result:

Quo_Part_#	SUM_OnOrder
105	200
124	900
125	200
134	700
135	2000
205	0
206	0
221	10000
222	10000
231	5000
232	0
241	6000

Example 19 INNER JOIN operations

A relational *JOIN* operation involves combining rows from one table with rows from another table. The *JOIN* operation is based on a comparison that is performed between the data element values in a column from the first table and the values from a column in the second table.

With an *INNER JOIN* operation, only the rows that satisfy the comparison used to create the *JOIN* are included in the result. Suppose that we wished to list the part number, name, on hand quantity, supplier, and on order quantity for those parts that currently have suppliers and thus have entries in the *Quotations* table. To do this, we need to combine information from the *Inventory* and *Quotations* tables.

The following *SELECT* statement joins the two tables based on the *Inv_Part_#* column from the *Inventory* table and the *Quo_Part_#* column from the *Quotation* table:

```
SELECT [Inv_Part_#], PartName, OnHand, [Quo_Supp_#], OnOrder
FROM Inventory INNER JOIN Quotations ON [Inv_Part_#]=[Quo_Part_#]
ORDER BY [Inv_Part_#], [Quo_Supp_#]
```

The above *JOIN* operation specifies that each row in the *Quotations* table that has a part number of 124 will be joined with the information for part 124 from the *Inventory* table. Only the rows for the part numbers that appear in the both the *Quotations* and *Inventory* tables are included in the result.

Result:

Inv_Part_#	PartName	OnHand	Quo_Supp_#	OnOrder
105	Gear	0	52	200
124	Bolt	900	51	400
124	Bolt	900	53	500
125	Bolt	1000	51	0
125	Bolt	1000	53	200
134	Nut	900	51	500
134	Nut	900	53	200
135	Nut	1000	51	1000

Inv_Part_#	PartName	OnHand	Quo_Supp_#	OnOrder
135	Nut	1000	53	1000
205	Band	450	52	0
206	Motor	225	52	0
221	Axle	1500	51	10000
222	Axle	25	53	10000
231	Axle	75	51	5000
232	Axle	150	53	0
241	Wheel	300	53	6000

In most cases, *JOIN* operations should be performed on columns that contain the same type of data, such as the part number in the previous example. The database software typically checks that the joining columns contain the appropriate type to be compared.

Different database software systems provide various ways of expressing different *JOIN* operations. For example, a *WHERE* clause can be used to express the same *JOIN* operation as in the previous example and gives us the same result:

```
SELECT [Inv_Part_#],
       PartName,
       OnHand,
       [Quo_Supp_#],
       OnOrder
FROM   Inventory, Quotations
WHERE  [Inv_Part_#] = [Quo_Part_#]
ORDER BY [Inv_Part_#], [Quo_Supp_#]
```

Example 20 Multiple *INNER JOIN* operations.

More than two tables can be joined in the same *SELECT* statement. Suppose that in our previous example we also wished to list the supplier name from the *Supplier* table. We simply include an additional *INNER JOIN* specification in the *FROM* clause:

```
SELECT [Inv_Part_#], PartName, OnHand, [Quo_Supp_#], Name, OnOrder
FROM   Inventory,
       Quotations,
       Suppliers,
       Inventory INNER JOIN Quotations ON [Inv_Part_#] = [Quo_Part_#],
       Suppliers INNER JOIN Quotations ON [Sup_Supp_#] = [Quo_Supp_#]
ORDER BY [Inv_Part_#], [Quo_Supp_#]
```

Result:

Inv_Part_#	PartName	OnHand	Quo_Supp_#	Name	OnOrder
105	Gear	0	52	Vesuvies Inc.	200
124	Bolt	900	51	Defecto Prts	400

Inv_Part_#	PartName	OnHand	Quo_Supp_#	Name	OnOrder
124	Bolt	900	53	Atlantis Co.	500
125	Bolt	1000	51	Defecto Prts	0
125	Bolt	1000	53	Atlantis Co.	200
134	Nut	900	51	Defecto Prts	500
134	Nut	900	53	Atlantis Co.	200
135	Nut	1000	51	Defecto Prts	1000
135	Nut	1000	53	Atlantis Co.	1000
205	Band	450	52	Vesuvies Inc.	0
206	Motor	225	52	Vesuvies Inc.	0
221	Axle	1500	51	Defecto Prts	10000
222	Axle	25	53	Atlantis Co.	10000
231	Axle	75	51	Defecto Prts	5000
232	Axle	150	53	Atlantis Co.	0
241	Wheel	300	53	Atlantis Co.	6000

We can get the same result using an alternative *Join* Syntax:

```
SELECT [Inv_Part_#], PartName, OnHand, [Quo_Supp_#], Name, OnOrder
FROM Inventory,
    Quotations,
    Suppliers
WHERE [Inv_Part_#]=[Quo_Part_#] AND [Sup_Supp_#]=[Quo_Supp_#]
ORDER BY [Inv_Part_#], [Quo_Supp_#]
```

Example 21 LEFT OUTER JOIN operation

Suppose we wish to list all part numbers and the suppliers that supply them. If a part number currently has no supplier, it should be listed with a null value displayed in the *Suppliers* column. This is left join of *Inventory* and *Quotations* tables based on part number values. This left outer join operator can be performed with the following statements:

```
SELECT [Inv_Part_#],[Quo_Supp_#]
FROM Inventory LEFT OUTER JOIN Quotations ON [Inv_Part_#]=[Quo_Part_#]
ORDER BY [Inv_Part_#], [Quo_Supp_#]
```

Results:

Inv_Part_#	Quo_Supp_#
105	52
106	
124	51
124	53
125	51
125	53
134	51

Inv_Part_#	Quo_Supp_#
134	53
135	51
135	53
171	
172	
181	
205	52
206	52
221	51
222	53
231	51
232	53
241	53

With most relational database packages, the keyword *OUTER* is optional, and *LEFT JOIN* produces the same result as *LEFT OUTER JOIN*.

Example 22 *RIGHT OUTER JOIN* operation

If we have not specified a relational integrity constraint as part of the database description, and it is possible for a part number to appear in the *Quotations* table without also being in the *Inventory* table, we might want to request a right outer join operation:

```
SELECT [Inv_Part_#],[Quo_Supp_#]
FROM Inventory RIGHT OUTER JOIN Quotations ON [Inv_Part_#]=[Quo_Part_#]
ORDER BY [Inv_Part_#],[Quo_Supp_#]
```

Result:

Inv_Part_#	Quo_Supp_#
105	52
124	51
124	53
125	51
125	53
134	51
134	53
135	51
135	53
205	52
206	52
221	51
222	53
231	51

Inv_Part_#	Quo_Supp_#
232	53
241	53

Using Structured Query Language for Updating Tables.

Inserting Table Data

Inserting Individual Rows

We insert new rows into a table with the *INSERT* statement. The following *INSERT* statement could be used to add the first row to empty *Inventory* table:

```
INSERT INTO Inventory
VALUES (126, 'Bolt', 0)
```

The *VALUES* clause is used to specify the data values to be inserted. The values must be listed in the same sequence in which the columns were defined in the table.

We can insert data into only selected columns by listing the columns names in parentheses following the table name:

```
INSERT INTO Inventory (Inv_Part_#, PartName)
VALUES (105, 'Gear')
```

Inserting Rows from Tables

Data can also be copied from one table to another by including a *SELECT* clause in an *INSERT* statement rather than a *VALUES* clause:

```
INSERT INTO Inventory (Inv_Part_#)
SELECT DISTINCT Quo_Part#
FROM Quotations
WHERE Price IS NOT NULL
```

Updating Table Data

We can modify the data stored in a table by using the *UPDATE* statement. We can make changes that apply to all rows or to selected rows, and can specify new values directly or we can specify calculations to be performed in processing the update.

Updating Individual Rows

The following *UPDATE* statement modifies a single row in the *Quotations* table:

```
UPDATE Quotations
SET    Price=1.30, Time=10
WHERE Quo_Supp_# = 51
      AND
      Quo_Part_# = 124
```

Updating Multiple Rows

The following example increases by 10 percent the value of *Price* in all rows:

```
UPDATE Quotations
SET Price = Price*1.10
```

Deleting Table Data

We can delete rows from a table by using issuing a *DELETE* statement. For example, the following example deletes all rows for supplier 52 from the *Quotations* table:

```
DELETE FROM Quotations
WHERE Quo_Supp_# = 53
```