Essential SQL

This bonus chapter is provided with *Mastering Delphi 6*. It is a basic introduction to SQL to accompany Chapter 14, "Client/Server Programming."

RDBMS packages are generally based so closely on SQL (Structured Query Language, commonly pronounced "sequel") that they are often called *SQL servers*. The SQL standard is defined by an ANSI/ISO committee, although many servers use custom extensions to the last official standard (called SQL-92 or SQL2). Recently many servers have started adding object extensions, which should be part of the future SQL3 standard.

Contrary to what its name seems to imply, SQL is used not only to query a database and manipulate its data, but also to define it. SQL actually consists of two areas: a Data Definition Language (DDL), including the commands for creating databases and tables; and a Data Manipulation Language (DML), including the query commands. The next two sections will explore these two distinct areas.

NOTE All of the SQL snippets presented in this chapter have been tested with InterBase 5 and 6.

SQL: The Data Definition Language

The DDL commands are generally used only when designing and maintaining a database; they are not used directly by a client application. The starting point is the `create database` command, which has a very simple syntax:

```
create database "mddb.gdb";
```

This command creates a new database (in practice, a new InterBase GDB file) in the current directory or in the indicated path. In the above statement, notice the final semicolon, used as a command terminator by the InterBase console. The opposite operation is `drop database`, and you can also modify some of the creation parameters with `alter database`.

NOTE In general, client programs should not operate on metadata, an operation that in most organizations would compete with the database administrator's responsibilities. I've added these calls to a simple Delphi program (called DdlSample) only to let you create new tables, indexes, and triggers in a sample database. You can use that example while reading the following sections. As an alternative, you can type the commands in the Windows Interactive SQL application.

Data Types

After creating the database, you can start adding tables to it with the `create table` command. In creating a table, you have to specify the data type of each field. SQL includes several data types, although it is less rich than Paradox and other local databases. Table 11.1 lists SQL standard data types and some other types available on most servers.

TABLE 11.1: The Data Types Used by SQL

Data Type	Standard	Usage
`char`, `character` (*n*)	Yes	Indicates a string of *n* characters. Specific servers or drivers can impose a length limit (32,767 characters for InterBase).
`int`, `integer`	Yes	An integer number, usually four bytes but platform dependent.
`smallint`	Yes	A smaller integer number, generally two bytes.
`float`	Yes	A floating-point number.
`double` (*precision*)	Yes	A high-precision floating-point number.
`numeric` (*precision*, *scale*)	Yes	A floating-point number, with the indicated precision and scale.
`date`	No	A date. Implementation of this data type varies from server to server.
`blob`	No	An object that holds a large amount of binary data (BLOB stands for binary large object).
`varchar`	No	A variable-size string used to avoid the space consumption of a fixed large string.

NOTE The `TStringField` class in Delphi can distinguish between `char` and `varchar` types, indicating the actual type in a property and fixing some the problems of using a `char` that's not padded with trailing spaces in the `where` clause of an `update` statement.

Programmers who are used to Paradox and other local engines will probably notice the absence of a logical or Boolean type, of date and time fields (the `date` type in InterBase holds both date and time), and of an AutoInc type, which offers a common way to set up a unique ID in a table. The absence of a logical type can create a few problems when upsizing an existing application. As an alternative, you can use a `smallint` field with 0 and 1 values for True and False, or you can use a domain, as explained in the next section. An AutoInc type is present in some servers, such as Microsoft SQL Server, but not in InterBase. This type can be replaced by the use of a generator, as discussed in the book.

Domains

Domains can be used to define a sort of custom data type on a server. A domain is based on an existing data type, possibly limited to a subset (as in a Pascal subrange type). A domain is a useful part of a database definition, as you can avoid repeating the same range check on several fields, and you can make the definition more readable at the same time.

As a simple example, if you have multiple tables with an address field, you can define a type for this field and then use this type wherever an address field is used:

```
create domain AddressType as char(30);
```

The syntax of this statement also allows you to specify a default value and some constraints, with the same notation used when creating a table (as we'll see in the next section). This is the complete definition of a Boolean domain:

```
create domain boolean as smallint default 0 check (value between 0 and 1);
```

Using and updating a domain (with the `alter domain` call) makes it particularly easy to update the default and checks of all the fields based on that domain at once. This is much easier than calling `alter table` for each of the tables involved.

Creating Tables

In the `create table` command, after the name of the new table, you indicate the definition of a number of columns (or fields) and some table constraints. Every column has a data type and some further parameters:

- `not null` indicates that a value for the field must always be present (this parameter is mandatory for primary keys or fields with unique values, as described below).
- `default` indicates the default value for the field, which can be any of the following: a given constant value, `null`, or `user` (the name of the user who has inserted the record).
- One or more constraints may be included, optionally with a name indicated by the `constraint` keyword. Possible constraints are `primary key`, `unique` (which indicates that every record must have a different value for this field), `references` (to refer to a field of another table), and `check` (to indicate a specific validity check).

Here is an example of the code you can use to create a table with simple customer information:

```
create table customer (
  cust_no       integer     not null primary key,
  firstname     varchar(30) not null,
  lastname      varchar(30) not null,
  address       varchar(30),
  phone_number  varchar(20)
);
```

In this example, we've used `not null` for the primary key and for the first and last name fields, which cannot be left empty. The table constraints can include a primary key using multiple fields, as in:

```
create table customers (
  cust_no   integer     not null,
  firstname varchar(30) not null,
  ...
  primary key (cust_no, name)
);
```

NOTE The most important constraint is `references`, which allows you to define a *foreign key* for a field. A foreign key indicates that the value of the field refers to a key in another table (a master table). This relation makes the existence of the field in the master table mandatory. In other words, you cannot insert a record referring to a nonexistent master field; nor can you destroy this master field while other tables are referencing it.

Once you've created a table, you can remove it with the `drop table` command, an operation that might fail if the table has some constrained relations with other tables.

Finally, you can use `alter table` to modify the table definition, removing or adding one or more fields and constraints. However, you cannot modify the size of a field (for example, a `varchar` field) and still keep the current contents of the table. You should move the contents of the resized field into temporary storage, drop the field, add a new one with the same name and a different size, and finally move back the data.

Indexes

The most important thing to keep in mind about indexes is that they are not relevant for the definition of the database and do not relate to the mathematical relational model. An index should be considered simply a suggestion to the RDBMS on how to speed up data access.

In fact, you can always run a query indicating the sort order, which will be available independently from the indexes (although the RDBMS can generate a temporary index). Of course, defining and maintaining too many indexes might require a lot of time; if you don't know exactly how the server will be affected, simply let the RDBMS create the indexes it needs.

The creation of an index is based on the `create index` command:

```
create index cust_name on customers (name);
```

You can later remove the index by calling `drop index`. InterBase also allows you to use the `alter index` command to disable an index temporarily (with the `inactive` parameter) and reenable it (with the `active` parameter).

Views

Besides creating tables, the database allows you to define views of a table. A view is defined using a `select` statement and allows you to create persistent *virtual* tables mapped to the physical ones. From Delphi, views look exactly the same as tables.

Views are a handy way to access the result of a join many times, but they also allow you to limit the data that specific users are allowed to see (restricting access to sensitive data). When the `select` statement that defines a view is simple, the view can also be updated, actually updating the physical tables behind it; otherwise, if the `select` statement is complex, the view will be read-only.

Migrating Existing Data

There are two alternatives to defining a database by manually writing the DDL statements. One option is to use a CASE tool to design the database and let it generate the DDL code. The other is to port an existing database from one platform to another, possibly from a local database to a SQL server. The Enterprise version of Delphi includes a tool to automate this process, the Data Pump Wizard.

The aim of this tool is to extract the structure of a database and recreate it for a different platform. Before starting the Data Pump, you should use BDE Administrator to create an alias for the database you want to create. Using Data Pump is quite simple: You select the source alias and the target alias; then you select the tables to move.

When you select a table (for example, the EMPLOYEE table) and click Next, the Data Pump Wizard will verify whether the upsizing operation is possible. After few seconds the wizard will display a list of the tables and let you modify a few options.

If the field conversion is not straightforward, the Data Pump will show the message "Has Problems" or "Modified." After modifying the options, if necessary, you can click the Upsize button to perform the actual conversion. By selecting a field, you can verify how the wizard plans to translate it; then, clicking the Modify Table Name or Field Mapping Information For Selected Item button, you can change the actual definition.

An alternative to the use of the Data Pump Wizard (available only in Delphi Enterprise) is the BatchMove component, which does a default conversion of the tables and cannot be fine-tuned. Finally, you can simply use the Database Desktop, create a new table on the server, and click the Borrow Struct button to extract the table definition from an existing local table.

SQL: The Data Manipulation Language

The SQL commands within the Data Manipulation Language are commonly used by programmers, so I'll describe them in more detail. There are four main commands: `select`, `insert`, `update`, and `delete`. All these commands can be activated using a Query component, but only `select` returns a result set. For the others, you should open the query using the `ExecSQL` method instead of `Open` (or the `Active` property).

Select

The `select` statement is the most common and well-known SQL command; it's used to extract data from one or more tables (or views) of a database. In its simplest form, the command is

```
select <fields> from <table>
```

In the *<fields>* section, you can: indicate one or more fields of the table, separated by commas; use the * symbol to indicate all the table fields at once; or even specify an operation to apply to one or more fields. Here is a more complex example:

```
select upper(name), (lastname || "," || firstname) as fullname
from customers
```

In this code, `upper` is a server function that converts all the characters to uppercase, the double-pipe symbol (`||`) is the string-chaining operator, and the optional `as` keyword gives a new name to the overall expression involving the first and last name.

By adding a `where` clause, you can use the `select` statement to specify which records to retrieve as well as which fields you are interested in:

```
select *
from customers
where cust_no = 100
```

This command selects a single record, the one corresponding to the customer whose ID number is 100. The `where` clause is followed by one or more selection criteria, which can be joined using the `and`, `or`, and `not` operators. Here is an example:

```
select *
from customers
where cust_no=100 or cust_no=200
```

The selection criteria can contain functions available on the server and use standard operators, including +, –, >, <, =, <>, >=, and <=. There are also few other special SQL operators:

`is null`	Tests whether the value of the field is defined.
`in <list>`	Returns True if the value is included in a list following the operator.
`between <min> and <max>`	Indicates whether the value is included in the range.

Here is an example of these operators:

```
select *
from customers
where address is not null and cust_no between 100 and 150
```

Another powerful operator, used to perform pattern matching on strings, is `like`. For example, if you want to look for all names starting with the letter *B*, you can use this statement:

```
select *
from employee
where last_name like "B%"
```

The `%` symbol indicates any combination of characters and can also be used in the middle of a string. For example, this statement looks for all the names starting with *B* and ending with *n*:

```
select *
from employee
where upper(last_name) like "B%N"
```

The use of `upper` makes the search case-insensitive and is required because the `like` operator performs a case-sensitive matching. An alternative to `like` is the use of the `containing` and `starting with` operators. Using `like` on an indexed field with InterBase might produce a very slow search, as the server won't always use the index. If you are looking for a match in the initial portion of a string, it is better to use the `starting with` expression, which enables the index and is much faster.

Another option is to sort the information returned by the `select` statement by specifying an `order by` clause, using one or more of the selected fields:

```
select *
from employee
order by lastname
```

The `asc` and `desc` operators can be used for ascending and descending order; the default is ascending.

An important variation of the `select` command is given by the `distinct` clause, which removes duplicated entries from the result set. For example, you can see all the cities where you have customers with this expression:

```
select distinct city
from customer
```

The `select` command can also be used to extract aggregate values, computed by standard functions:

`avg`	Computes the average value of a column of the result set (works only for a numeric field).

`count`	Computes the number of elements in the result set; that is, the number of elements satisfying the given condition.
`max` and `min`	Compute the highest and lowest values of a column in the result set.
`sum`	Computes the total of the values of a column of the result set. (It works only for numeric fields.)

These functions are applied to the result set, usually to a specific column, excluding the `null` values. This statement computes the average salary:

```
select avg(salary)
from employee
```

Another important clause is `group by`, which lets you aggregate the elements of the result set according to some criterion before computing aggregate values with the functions listed above. For example, you might want to determine the maximum and average salary of the employees of each department:

```
select max (salary), avg (salary), department
from employee
group by department
```

Notice that all the noncalculated fields must appear in the `group by` clause. The following is not legal:

```
select max (salary), lastname, department
from employee
group by department
```

TIP When you extract aggregate values, it is better to use an alias for the result field with the `as` keyword. This makes it easier to refer to the resulting value in your Delphi code.

The aggregate values can also be used to determine the records in the result set. The aggregate functions cannot be used in the `where` clause, but they are placed in a specific `having` section. The following statement returns the highest salary of each department, but only if the value is above 40,000:

```
select max(salary) as maxsal, department
from employee
group by department
having max(salary) > 40000
```

Another interesting possibility is to nest a `select` statement within another one, forming a subquery. Here is an example, used to return the highest-paid employee (or employees):

```
select firstname, lastname
from employee
where salary = (select max(salary) from employee)
```

We could not have written this code with a single statement, since adding the name in the query result would have implied adding it to the `group by` section as well.

Inner and Outer Joins

Up to now, our example `select` statements have worked on single tables—a serious limitation for a relational database. The operation of merging data from multiple source tables is called a *table join*. The SQL standard supports two types of joins, called *inner* and *outer*.

An inner join can be written directly in the `where` clause:

```
select *
from <table1>, <table2>
where <table1.keyfield>=<table2.externalkey>
```

This is a typical example of an inner join used to extract all the fields of each table involved. An inner join is handy for tables with a one-to-one relation (one record of a table corresponding only to one record of the second table). Actually, the standard syntax should be the following, although the two approaches usually generate the same effect:

```
select *
from <table1> left join <table2>
on <table1.keyfield>=<table2.externalkey>
```

An outer join, instead, can be specifically requested with the statement:

```
select *
from <table1> left outer join <table2>
on <table1.keyfield>=<table2.keyfield>
```

The main difference from an inner join is that the selected rows of an outer join will not consider the null fields of the second table. There are other types of joins, including these: the self-join, in which a table merges with itself; the multi-join, which involves more than two tables; and the Cartesian product, a join with no `where` condition, which merges each row of a table with each row of the second one, usually producing a huge result set. The inner join is certainly the most common form.

Insert

The `insert` command is used to add new rows to a table or an updateable view. When you insert a new record in a DBGrid connected with a SQL server table, the BDE generates an `insert` command and sends it to the server. Besides this implicit use, there are several cases in which you'll want to write explicit SQL `insert` calls (including the use of cached updates, which we'll discuss later in this chapter).

Unless you add a value for each field, you should list the names of the fields you are actually providing, as in the following code:

```
insert into employee (empno, lastname, firstname, salary)
values (0,"brown", "john", 10000)
```

You can also insert in a table the result set of a `select` statement (if the fields of the target table have the same structure of the selected fields), with this syntax:

```
insert into <table> <select statement>
```

Update

The `update` command modifies one or more records of a table or view. Delphi generates an `update` call every time you edit data with visual controls connected to a table or a live query on a SQL server. Again, there are also cases where you'll want to use the `update` statement directly.

In an `update` statement, you can indicate which record to modify, by using a `where` condition similar to that of a `select` statement. For example, you can change the salary of a specific employee with this call:

```
update employee
set salary = 30000
where emp_id = 100
```

WARNING A single `update` instruction can update all the records that satisfy a given condition. An incorrect `where` clause could unintentionally update many records, and no error message would be displayed.

The `set` clause can indicate multiple fields, separated by commas, and it can use the current values of the fields to compute the new values. For example, the following statement gives a nice raise to the employees hired before January 1, 1990:

```
update employee
set salary = salary * 1.20
where hiredate < "01-01-1990"
```

Delete

The `delete` command is equally simple (although its misuse can be quite dangerous). Again, you generally remove records using a visual component, but you can also issue a SQL command like the following:

```
delete from employee
where empid = 120
```

You simply indicate a condition identifying the records to delete. If you issue this SQL command with a Query component (calling `ExecSQL`), you can then use the `RowsAffected` property to see how many records were deleted. The same applies to the `update` commands.

Building Queries Visually in Delphi (with SQL Builder)

As we've seen, SQL has a great many commands, particularly in relation to `select` statements. And we haven't actually seen them all! While the DDL commands are generally used by a database administrator, or only for the initial definition of the database, DML commands are commonly used in everyday Delphi programming work.

To help with the development of correct SQL statements, Delphi Enterprise includes a tool called SQL Builder, which unfortunately can be used only through a BDE connection (even if you can later copy the query and use it, for example, with a dbExpress dataset). You easily activate it by right-clicking a Query component.

Using SQL Builder is very simple. You choose the database you want to work on, and then you select one or more tables, placing them in the work area. After selecting the proper parameters, as explained below, you can use the command Query ➢ Run Query (or F9) to see the result of the query or the command Query ➢ Show SQL (F7) to see the source code of the `select` statement you've generated.

In the selected tables, you can simply mark the fields you want to see in the result set. The check box near the name of the table selects all of its fields. But the real power of SQL Builder lies in two features. First, you can drag a field from one table onto another table to join them.

The other powerful feature is the Query notebook, the multipage control at the bottom of the SQL Builder window. Here is a short description of each of the pages:

The Criteria page indicates the selection criteria of the `where` clause. By selecting one of the fields of the result table, you can indicate a comparison against a fixed value or another field, and you can use `like`, `is null`, `between`, and other operators. Using the local menu of the grid present in this page, you can also activate the `exist` operator or an entire SQL expression. This page allows you to combine multiple conditions with the `and`, `or`, and `not` operators, but it doesn't allow you to specify a precedence among these operators by adding parentheses.

The Selection page lists all the fields of the result set and allows you to give them an alias. With the local menu, you can also introduce aggregate functions (`sum`, `count`, and so on). Finally, the upper-left check box indicates the `distinct` condition.

The Grouping page corresponds to the `group by` clause. SQL Builder automatically groups all the fields that are used in the aggregate functions, as required by the SQL standard.

The Group Criteria page corresponds to a `having` clause, which is available in conjunction with aggregate functions. The operations are similar to those of the Selection page and are activated by using the local menu.

The Sorting page corresponds to the `order by` clause. You simply select the field you want to sort and then select the ascending or descending sort.

The Joins page is the last but probably the most powerful, as it allows you to define join conditions, beyond the simple dragging of a field from one table to another in the work area. This page allows you to fine-tune the join request by indicating its type (`INNER` or `OUTER`) and selecting conditions other than the equality test.

To better understand how to use the SQL Builder, we can build an actual example based on the sample InterBase database installed by Delphi (and corresponding to the LocalIB alias). The example is in the `SqlBuilder` directory and its form has a Query, a DataSource, and a DBGrid component, connected as usual. The `DatabaseName` property of the Query component is set to IBLocal, and right-clicking the component activates SQL Builder.

We want to create a query including the first and last name, department, title, and salary of each employee. This operation requires two joins. Choose the Employee, Department, and Job tables. Click the Dep_No field of the Department table and drag the cursor over the Dep_No field of the Employee table. Similarly, connect the Job table with the Employee table using the three fields Job_Code, Job_Grade, and Job_Country.

After creating the joins, select the fields you want to see in the result set: First_ Name, Last_Name, and Salary from the Employee table; Department from the Department table; and Job_Title from the Job table. Finally, move to the Sorting page of the Query notebook and select Department.Department from the Output Fields list to sort the result set by department.

The following should be the generated SQL:

```
select employee.first_name, employee.last_name, department.department,
  job.job_title, employee.salary
from employee employee
  inner join department department
  on (department.dept_no = employee.dept_no)
  inner join job job
  on (job.job_code = employee.job_code)
  and (job.job_grade = employee.job_grade)
  and (job.job_country = employee.job_country)
order by department.department
```

We might add an extra `where` clause to choose only the employee with a high salary. Simply move to the Selection page, select the Employee.Salary field, go to the column Operator >=, and type the value **100,000**. Executing the query, you'll see a limited number of records, and looking at the SQL source you'll see the extra statement:

```
where employee.salary >= 100000
```

Finally, note that it is possible to export and import the SQL code from a plain text file. Simply by closing SQL Builder, you will also save the text of the query in the `SQL` property of the related Query component.