Product Documentation

InterBase XE3 Quick Start

March, 2014

Embarcadero Technologies, Inc. is a leading provider of award-winning tools for application developers and database professionals so they can design systems right, build them faster and run them better, regardless of their platform or programming language. Ninety of the Fortune 100 and an active community of more than three million users worldwide rely on Embarcadero products to increase productivity, reduce costs, simplify change management and compliance and accelerate innovation. The company's flagship tools include: Embarcadero® Change Manager™, CodeGear™ RAD Studio, DBArtisan®, Delphi®, ER/Studio®, JBuilder® and Rapid SQL®. Founded in 1993, Embarcadero is headquartered in San Francisco, with offices located around the world. Embarcadero is online at www.embarcadero.com.

Contents

INTRODUCTION

This Quick Start takes you step-by-step through the process of creating and using a database using the InterBase IBConsole dialog. You learn to create data structures that enforce referential integrity constraints and maintain security. You populate your tables, create triggers and stored procedures, and learn a number of techniques for retrieving the data with precision.

There are five parts in this tutorial:

- Part I: Getting Started
  - How to Install and Start the InterBase Server
  - How to Create a New User and a Database
- Part II: Data Definition
  - What are the Fundamentals of Database Design
  - How to Work in the IBConsole Environment
  - How to create the Data Structures for your Database
  - How to Recover from Errors
- Part III: Populating the Database
  - How to Populate Your Database
- Part IV: Retrieving Data
  - How to Access or Retrieve your Data
- Part V: Advanced Topics
  - How to Work with Database Security
  - How to Create Some Triggers and Stored Procedures to Automate Some of Your Database Tasks

USING THIS QUICK START

Throughout this Quick Start, you are instructed to enter SQL statements manually at the beginning of each new topic in order to give you a hands-on experience. Then you are instructed to read in one of the SQL scripts that accompany this document. These steps allows you to create a database that is complex enough to be interesting without excessive keyboarding. The database that you create is, in fact, the EMPLOYEE database that is used as the example database for InterBase and is referenced throughout the InterBase document set.

Finding the Files you Need

As you reach the places in this tutorial there are scripts that need to be run. Use the script files (*.sql) that are located in the InterBase Quick Start Scripts.zip file in http://docs.embarcadero.com/products/interbase/

Printing Conventions

The following typographic conventions are used through this document:

Convention	Purpose	Example
UPPERCASE	SQL keywords, SQL functions, and names of all database objects such as tables, columns, indexes, and stored procedures	the SELECT statement retrieves data from the CITY column in the CITIES table can be used in CHAR, VARCHAR, and BLOB text columns the CAST() function
italic	New terms, emphasized words, all elements from host languages, and all user-supplied items	isc_decode_date() the host variable, *segment_length* contains six variables, or *data members*
bold	File names, menu commands, and all commands that are entered at a system prompt, including their switches, arguments, and parameters	a script, **ib_udf.sql**, in the examples subdirectory the **employee.gdb** database the **Session > Advanced Settings** command

SYNTAX CONVENTIONS

The following table lists the conventions used in syntax statements and sample code, and provides examples of their use:

Convention	**Purpose**	**Example**
UPPERCASE	Keywords that must be typed exactly as they appear when used	SET TERM !!; ADD [CONSTRAINT] CHECK
italic	User-supplied parameters that cannot be broken into smaller units	CREATE TRIGGER *name* FOR *table*; ALTER EXCEPTION *name* '*message*'
<*italic*>	Parameters in angle brackets can be broken into smaller syntactic units; the expansion syntax for these parameters follows the current syntax statement	WHILE (<condition>) DO <compound_statement>33
[ ]	Optional syntax: you do not need to include anything that is enclosed in square brackets; when elements within these brackets are separated by the pipe symbol (|), you can choose only one	CREATE [UNIQUE][ASCENDING | DESCENDING] [FILTER [FROM *subtype*] TO *subtype*]
{ }	You must include one and only one of the enclosed options, which are separated by the pipe symbol (|)	{INTO | USING}
|	You can choose only one of a group whose elements are separated by this pipe symbol	SELECT [DISTINCT | ALL]
...	You can repeat the clause enclosed in brackets with the "…" symbol as many times as necessary	(<col> [,<col>…])

Reading and typing capitals

Type of entry	Case sensitivity
SQL statements	When you're entering SQL statements into IBConsole, you can ignore the capitalization. The conventions listed above are to make it easy to read and understand the examples. You can enter the exercises in all lower case if you prefer.
Strings	Strings (anything inside of quotation marks) *are* case sensitive. There are many strings in single quotes in this Quick Start, and you must enter the case exactly as it's shown.
External references	When you refer to something outside of InterBase, such as a filename, the reference is case sensitive.

Line Breaks

- Line breaks are added within example statements to make them easy to read and understand. They are not required.
- When you're entering statements you don't have to follow the line breaks in the examples. Enter ones that make it easy for you to keep track of what you're doing. InterBase ignores line breaks within input SQL statements.

Understanding Which Parts to Do

This tutorial contains some code examples that you are not supposed to enter into the TUTORIAL database. In other places, it gives the text of code that you *are* supposed to enter - these are your action items. So the code that you actually enter are all preceded by topic headings with an ▶ symbol. For example all code in the "Disconnecting from a Database" topic needs to be entered because it is preceded by the green symbol.

Advance to Part I

Click here to advance to Part I: Getting Started which covers installation, starting the local InterBase server, logging onto a server from the InterBase IBConsole, creating a new user on a server, and creating a new database on a server

Part I: Getting Started

In Part I, you perform the following actions:

- Install InterBase XE
  - Installing InterBase XE3 Update 4 for Windows
  - Installing InterBase XE for Mac OS X
  - Installing InterBase XE for Linux
- Start the Local InterBase server
- Log onto a server from the InterBase IBConsole
- Create a new user on a server
- Create a new database on a server

Installing InterBase XE3 Update 4 for Windows

Before you get started you need to do the following:

- Go to https://downloads.embarcadero.com/free/interbase and download InterBase XE Developer Edition for Windows.The file is InterBase_XE3.zip.

  NOTE: The InterBase XE3 download includes both 32-bit and 64-bit in the same Windows install.

- A Developer's Network account is required. Click http://edn.embarcadero.com to either log on to an existing account or create a new one.

  NOTE: For installing on the Mac click: Installing InterBase XE for Mac OS X. For installing on Linux click: Installing InterBase XE for Linux

Installation

1 Double-click the InterBase_XE3.zip file, and then double-click **ib_install.exe**.

NOTE: It is recommended that you exit all Windows programs before running the setup program.

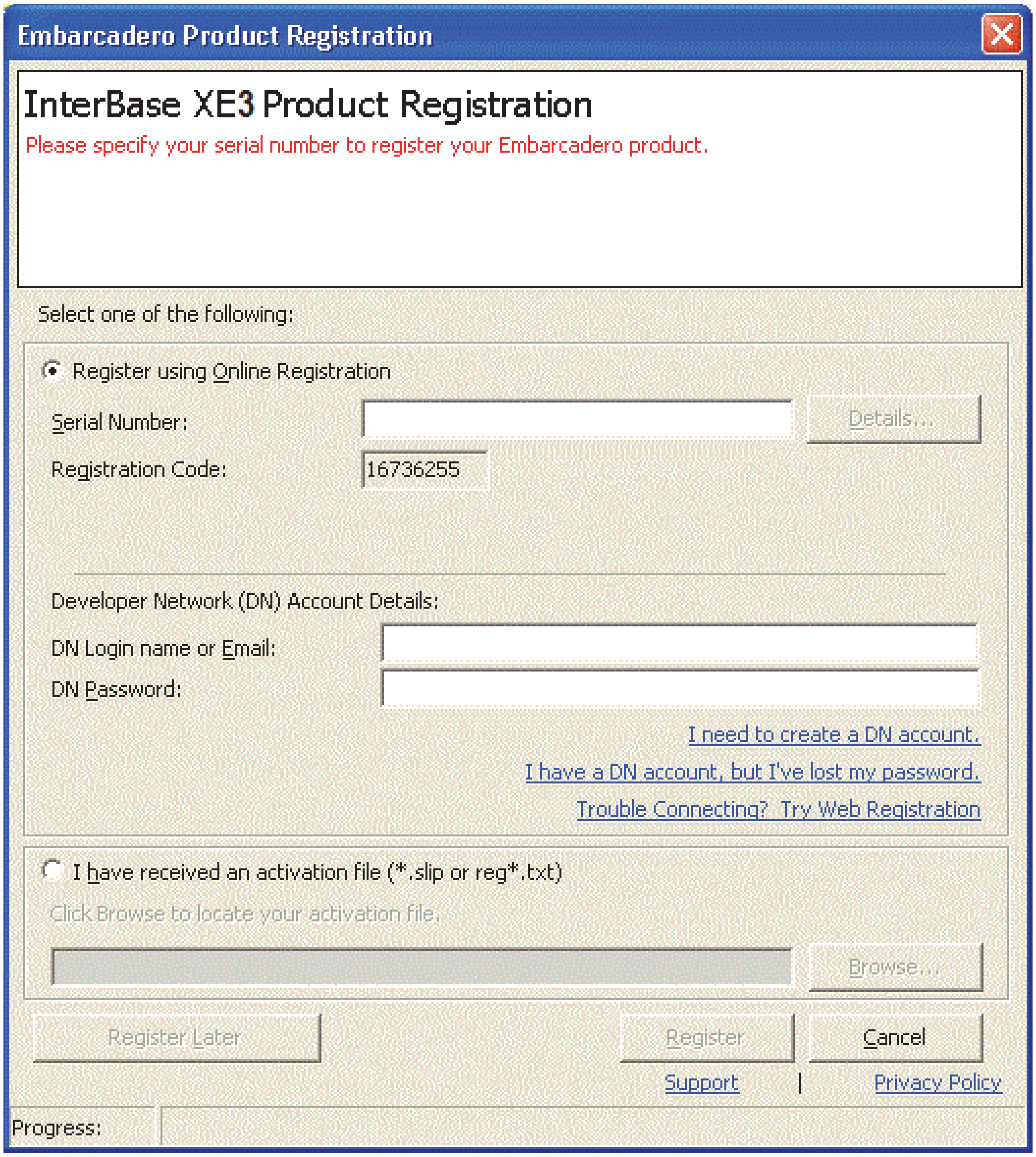
2 Following the instructions as you are guided through the install wizard.

3 In the Multi Instance Feature pane, make sure you leave the default setting of "No".

4 In the Select Components pane, select **Server and Client**, **Documentation**, and **Register** and click **Next**.

5 Leave the default destination folder as is and click **Next** again to start the installation. The Embarcadero InterBase XE installation dialog opens.

6 Once InterBase is installed the InterBase XE Product Registration dialog appears.



Embarcadero Product Registration

InterBase XE3 Product Registration

Please specify your serial number to register your Embarcadero product.

Select one of the following:

● Register using Online Registration

Serial Number:

Registration Code: 16736255

Developer Network (DN) Account Details:

DN Login name or Email:

DN Password:

I need to create a DN account.

I have a DN account, but I've lost my password.

Trouble Connecting? Try Web Registration

○ I have received an activation file (*.slip or reg*.txt)

Click Browse to locate your activation file.

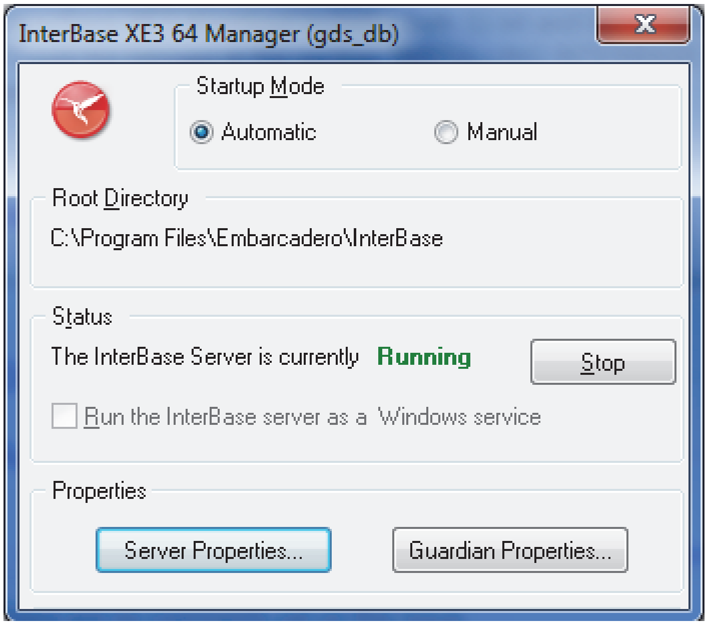
Progress:

7 When you selected to download your application, you were emailed a serial number. Enter it and your DN account information and click **Register**

8 Click **Finish** to complete the installation.

To Start the Local InterBase Server

1 To start the Local InterBase server running as an application on Windows, choose **Start > Programs > Embarcadero InterBase XE [instance = gds_db] > InterBase Server Manager** and the InterBase XE 32 Manager dialog opens.



InterBase XE3 64 Manager (gds_db)

Startup Mode

● Automatic ○ Manual

Root Directory

C:\Program Files\Embarcadero\InterBase

Status

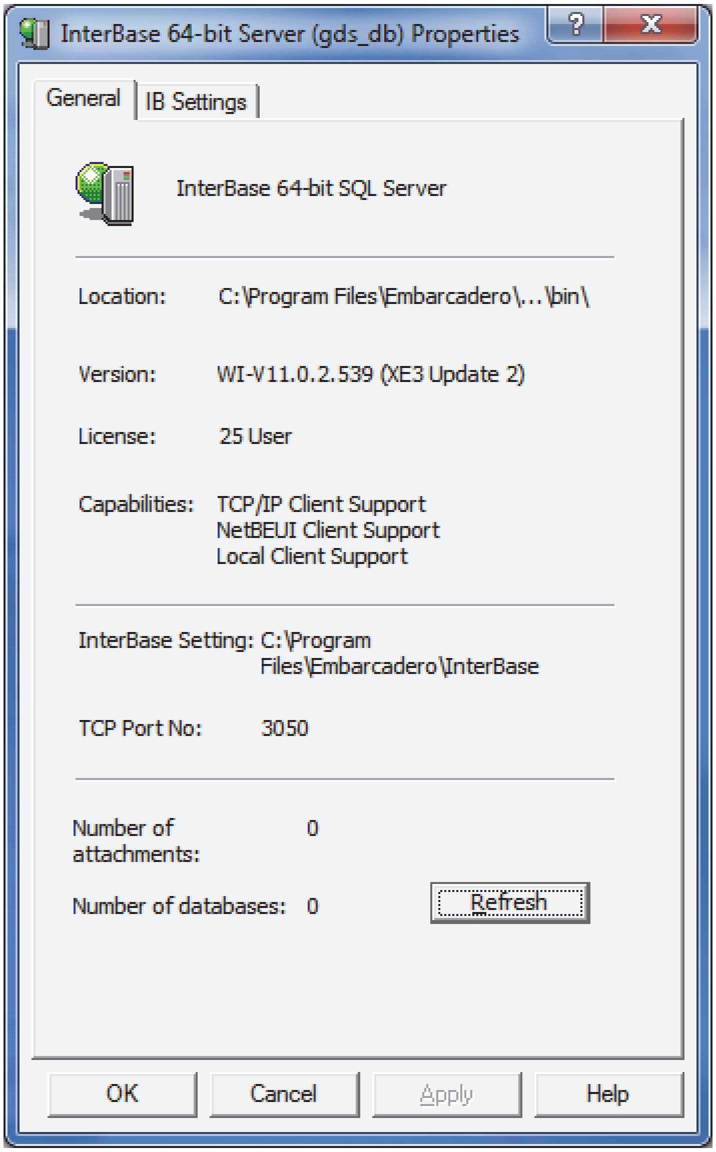
The InterBase Server is currently Running

□ Run the InterBase server as a Windows service

Properties

2 This dialog shows the default Startup Mode, the root directory, and the status of the server. Click **Start** and the status changes to **Running**.

3 Deselect the “Run the InterBase server as a Windows Service” and click **Server Properties** and additional information appears in the dialog showing the number of databases and attachments.



InterBase 64-bit SQL Server

Location: C:\Program Files\Embarcadero\...\bin\

Version: WI-V11.0.2.539 (XE3 Update 2)

License: 25 User

Capabilities: TCP/IP Client Support
NetBEUI Client Support
Local Client Support

InterBase Setting: C:\Program Files\Embarcadero\InterBase

TCP Port No: 3050

Number of attachments: 0

Number of databases: 0

4 Click **Guardian Properties** and a dialog opens showing the location, version, and action associated with the Guardian server. Close **OK** to close this dialog.

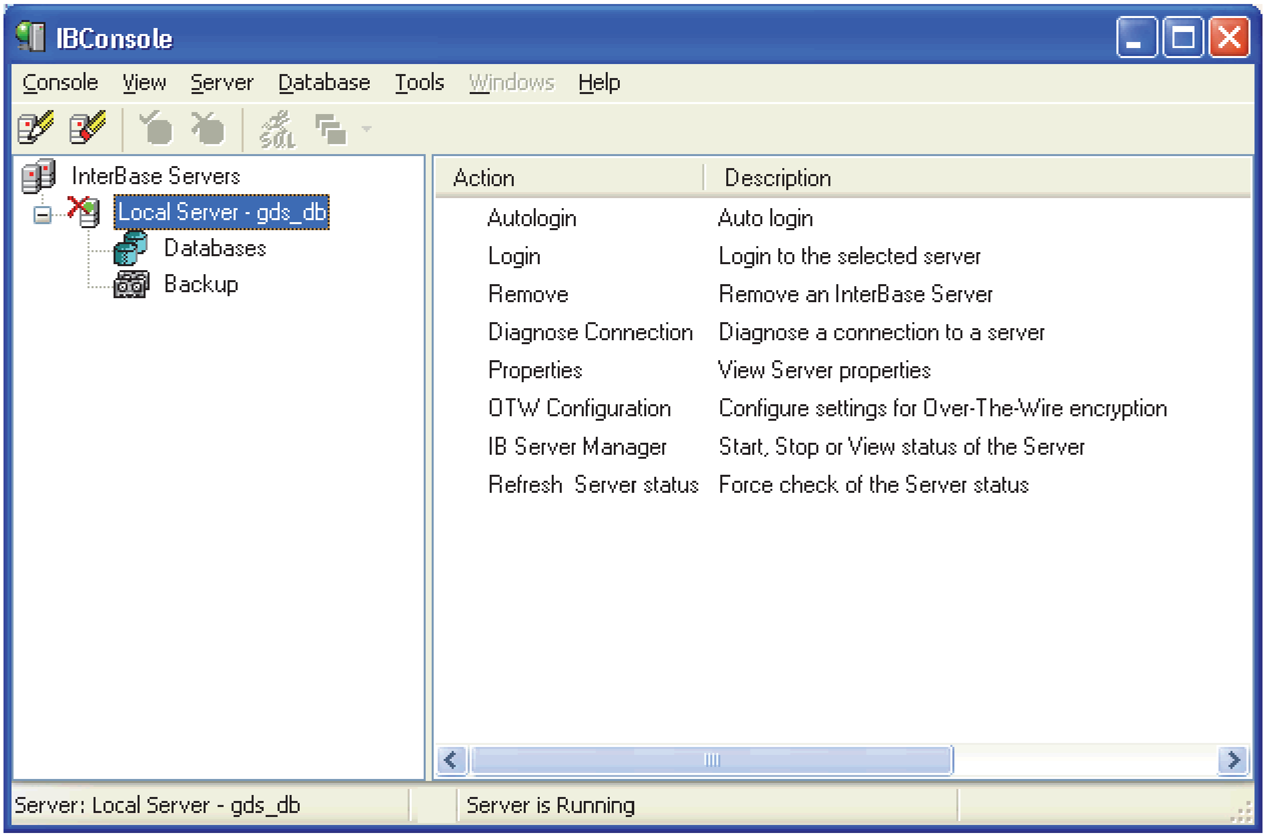
5 To open IBConsole choose **`Start > Programs > Embarcadero InterBase XE3 [instance = gds_db] > IBConsole.`**

NOTE: To start Local InterBase as a service on a Windows NT platforms, click **Administrative Tools > Services** in the Control Panel, highlight the **InterBase XE Guardian** entry, and click **Start**.

Connecting to a Server from IBConsole

The text of this tutorial assumes that you are working on the Local InterBase server. If you want to work on a remote server, you must have the password for a valid InterBase user on that server.

In this exercise, you connect as the SYSDBA user, since that is the only user who can create new user accounts. Later you will create a user called TUTOR, which will be the account you will use for the rest of these exercises.

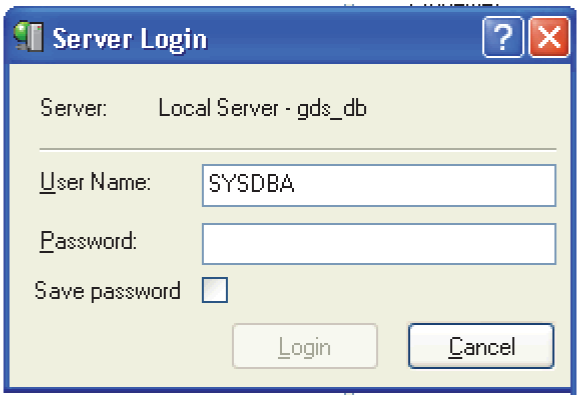


Action	Description
Autologin	Auto login
Login	Login to the selected server
Remove	Remove an InterBase Server
Diagnose Connection	Diagnose a connection to a server
Properties	View Server properties
OTW Configuration	Configure settings for Over-The-Wire encryption
IB Server Manager	Start, Stop or View status of the Server
Refresh Server status	Force check of the Server status

Server: Local Server - gds_db

Server is Running

- To log on to the Local InterBase server click **Server > Login** the **Server Login** dialog opens.



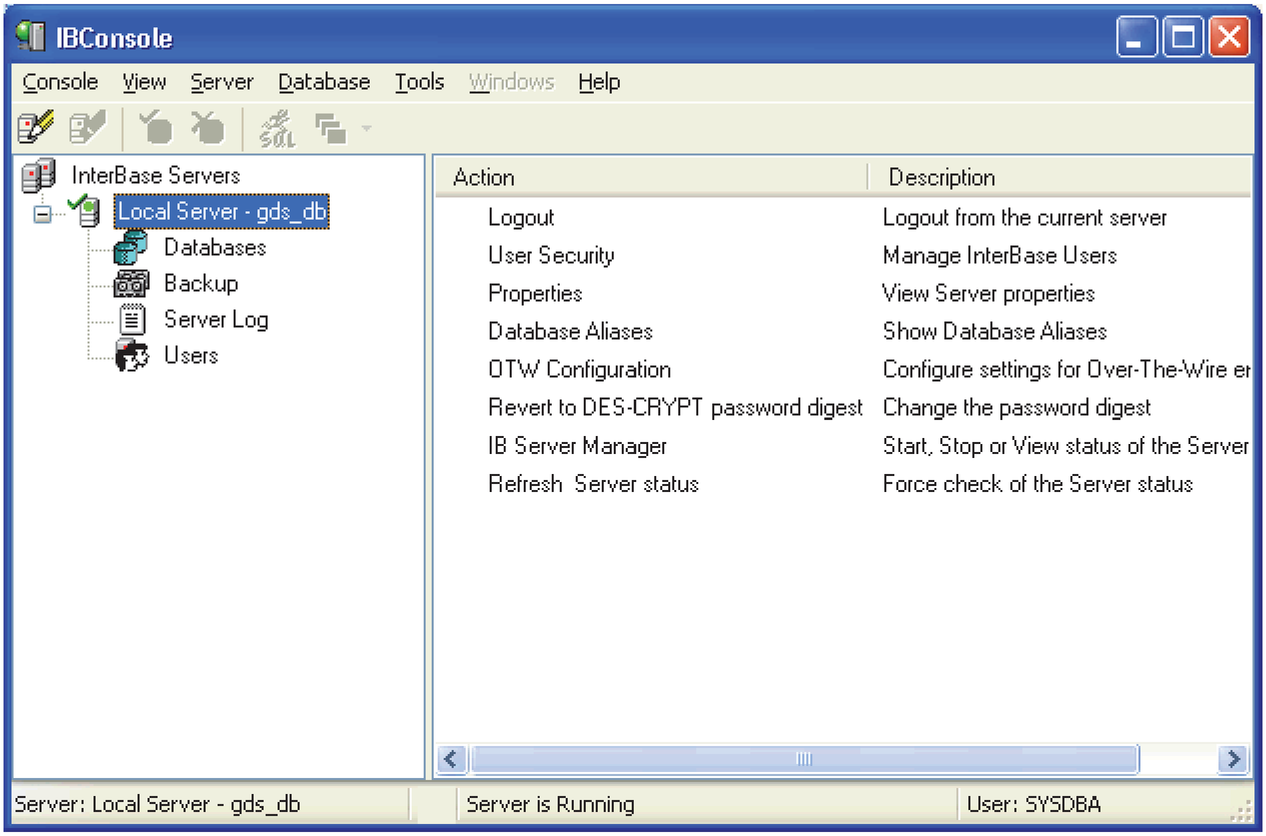
Server: Local Server - gds_db

User Name: SYSDBA

Password:

Save password □

6 Use the password **masterkey** for the SYSDBA user and click **Login**. You are now logged into the Local Server gds_db database.



Action	Description
Logout	Logout from the current server
User Security	Manage InterBase Users
Properties	View Server properties
Database Aliases	Show Database Aliases
OTW Configuration	Configure settings for Over-The-Wire er
Revert to DES-CRYPT password digest	Change the password digest
IB Server Manager	Start, Stop or View status of the Server
Refresh Server status	Force check of the Server status

Server: Local Server - gds_db

Server is Running

User: SYSDBA

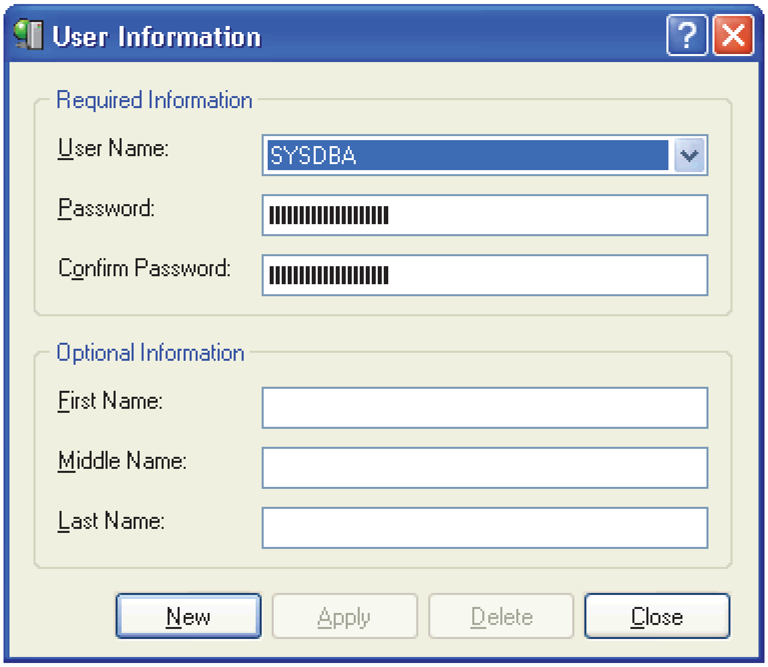
Creating a New User

Once you create the new user named TUTOR, the rest of this tutorial assumes that you are the user **TUTOR** and that your password is **tutor4ib**.

Creating a New User

NOTE: InterBase ships with one user, SYSDBA as the default.

1 To display the User Information dialog in IBConsole, click **Server > User Security**.



User Name: SYSDBA

Password: ||||||||||||||||||

Confirm Password: ||||||||||||||||||

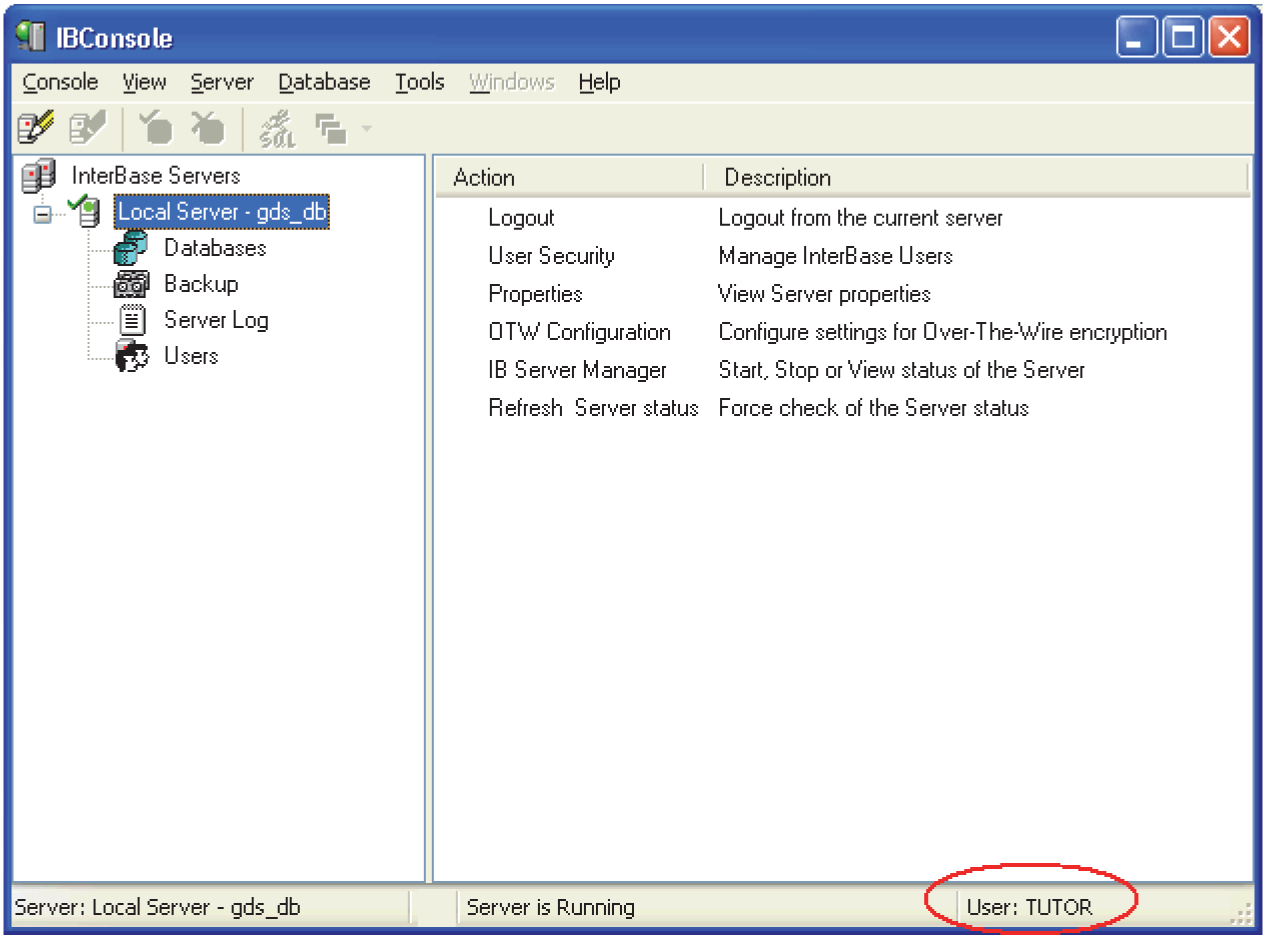
2 Click **New** to clear the Required Information and Optional Information fields.

3 Type **TUTOR** in the User Name field. This appears in all caps even if the Caps Lock is off. Type **tutor4ib** in the Password and Confirm Password field. The password is case-sensitive. You do not need to complete the Optional Information.

4 Click **Apply** to create the user, and then click **Close.**

5 Click **Server > Logout** and log out of the server. You will be asked to confirm this action.

6 Now click **Server > Login** and enter your new user name TUTOR and password. Once the login is complete the TUTOR user appears in the Summary pane.



Action	Description
Logout	Logout from the current server
User Security	Manage InterBase Users
Properties	View Server properties
OTW Configuration	Configure settings for Over-The-Wire encryption
IB Server Manager	Start, Stop or View status of the Server
Refresh Server status	Force check of the Server status

Server: Local Server - gds_db

Server is Running

User: TUTOR

You have now created a user on the server you're logged into. Users are defined server-wide and can connect to any database that resides on that server. Tables within these databases have additional security, however. Being able to connect to a database won't do you much good if you don't have privileges on any of its tables. This will be covered later.

Create a Database

Now that you have created a valid user name, you can create the TUTORIAL database that you will use for the exercises in this tutorial.

InterBase databases are stored in files that have a .gdb extension.

Creating the TUTORIAL Database

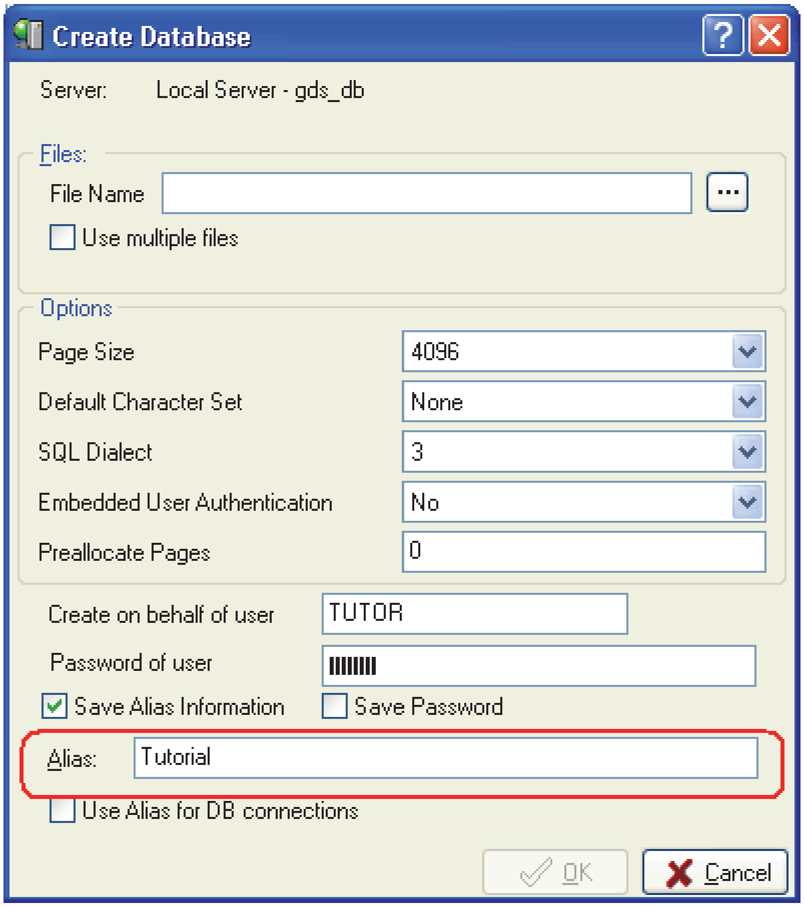
1 If it isn't already open, open IBConsole by choosing it from the InterBase folder on the **Start > Programs** menu.

2 Log on to the server as the **TUTOR** user.

NOTE: Be sure you log on as TUTOR for the remaining exercises in this tutorial.

3 For this tutorial you will use `C:\Program Files\Embarcadero\Interbase\database\`. If you choose a different location, you must edit the CONNECT statement in the SQL script files to reflect the new location. (You're instructed how to do this a little later.)

4 To open the Create Database dialog, highlight the server in the left pane and select **Database > Create Database**.



Server: Local Server - gds_db

File Name

□ Use multiple files

Page Size 4096

Default Character Set None

SQL Dialect 3

Embedded User Authentication No

Preallocate Pages 0

Create on behalf of user TUTOR

Password of user ||||||||

☑ Save Alias Information □ Save Password

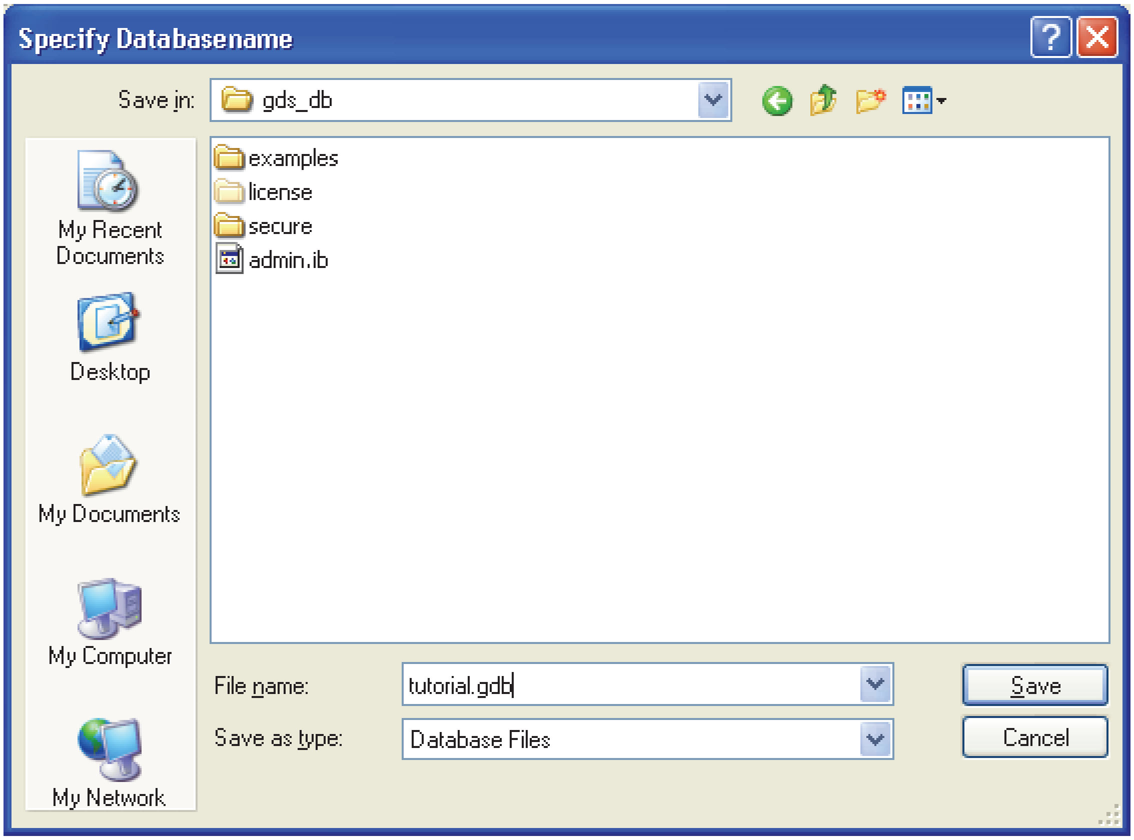
Alias: Tutorial

□ Use Alias for DB connections

5 Enter **Tutorial** in the Alias field

6 In the Filename field, browse to the location of your new database at

```
 C:\Documents and Settings\All Users\Application
Data\Embarcadero\Interbase\gds_db\
```


Specify Databasename

Save in: gds_db

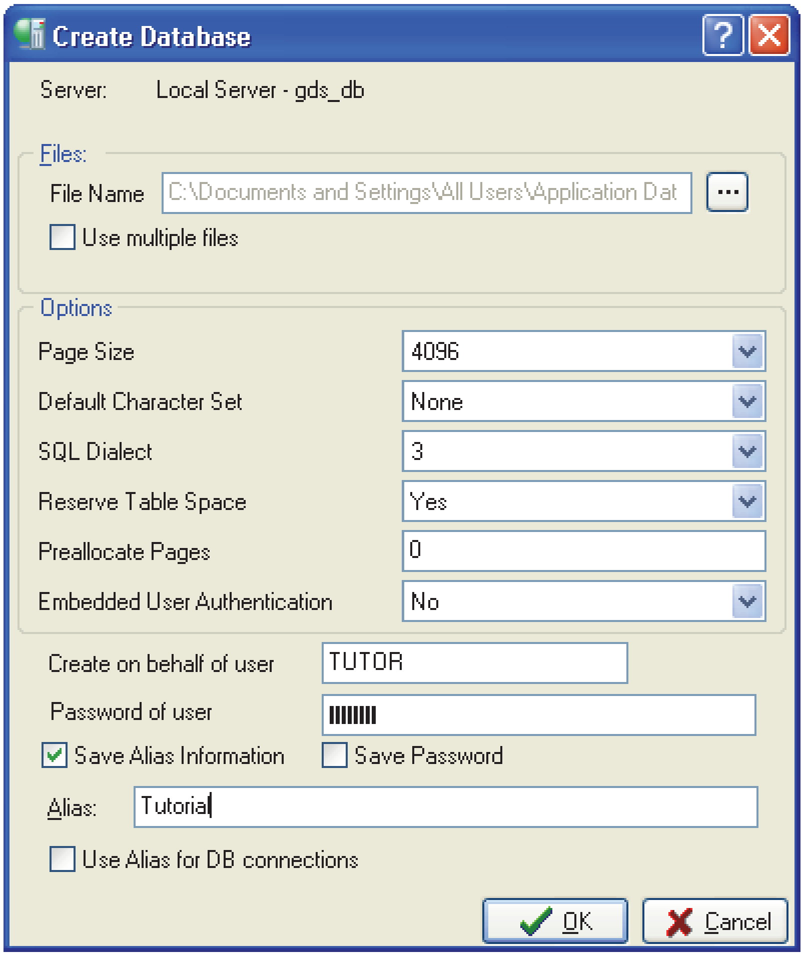
examples
license
secure
admin.ib

File name: tutorial.gdb

Save as type: Database Files

7 Enter **tutorial.gdb** in the File name field and click **Save**

8 Set the **SQL Dialect** to 3. Leave the rest of the settings as is.



Server: Local Server - gds_db

Files:

File Name C:\Documents and Settings\All Users\Application Dat ...

□ Use multiple files

Options

Page Size	4096
Default Character Set	None
SQL Dialect	3
Reserve Table Space	Yes
Preallocate Pages	0
Embedded User Authentication	No

Create on behalf of user TUTOR

Password of user ||||||||

☑ Save Alias Information □ Save Password

Alias: Tutorial

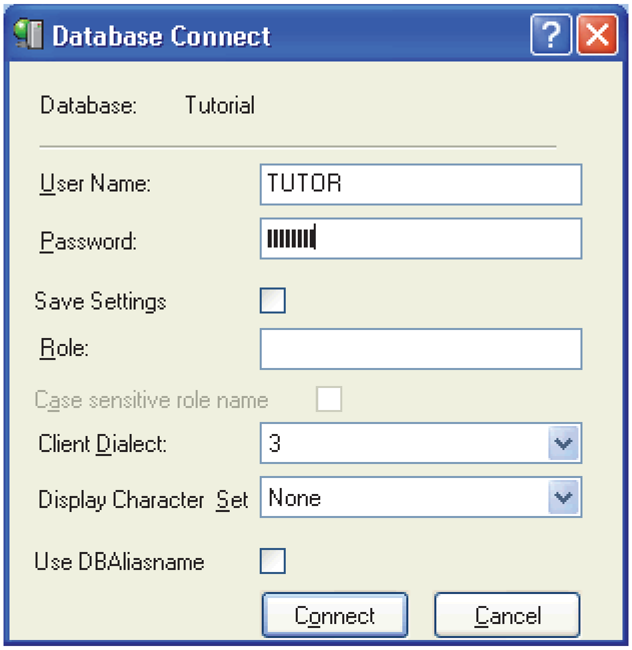
□ Use Alias for DB connections

9 Click **OK** to create the TUTORIAL database. This new database named TUTORIAL belongs to the user TUTOR.

Connecting to a Database

1 To connect to the database, click and the DataBase Connect dialog opens.

NOTE: The Database Connect dialog opens automatically when you create the TUTORIAL database.



Database: Tutorial

User Name: TUTOR

Password: ••••••••

Save Settings □

Role:

Case sensitive role name □

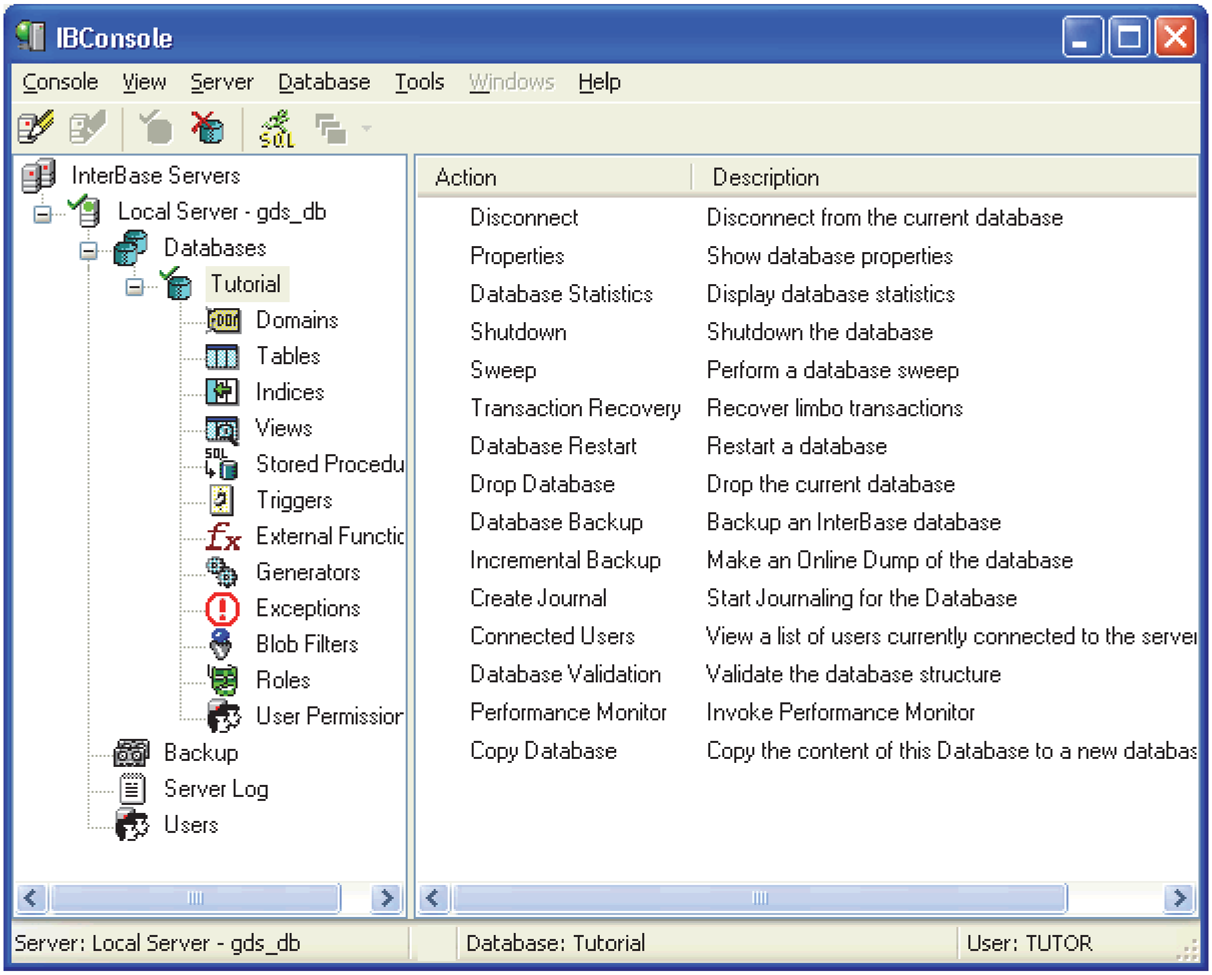
Client Dialect: 3

Display Character Set None

Use DBAliasname □

2 Enter your Password and then click **Connect**.

Once you are connected, the Status Bar at the bottom of the IBConsole window shows the server name, the name of the database you just created, and the name of the user connected to the server.



Action	Description
Disconnect	Disconnect from the current database
Properties	Show database properties
Database Statistics	Display database statistics
Shutdown	Shutdown the database
Sweep	Perform a database sweep
Transaction Recovery	Recover limbo transactions
Database Restart	Restart a database
Drop Database	Drop the current database
Database Backup	Backup an InterBase database
Incremental Backup	Make an Online Dump of the database
Create Journal	Start Journaling for the Database
Connected Users	View a list of users currently connected to the serve
Database Validation	Validate the database structure
Performance Monitor	Invoke Performance Monitor
Copy Database	Copy the content of this Database to a new databas

Server: Local Server - gds_db | Database: Tutorial | User: TUTOR

Disconnecting From a Database

To disconnect from a database:

- Click Disconnect ( )

OR

- Select **Database > Disconnect**.

When InterBase asks whether you want to disconnect from the database, click **Yes.**

Installing InterBase XE for Mac OS X

Before you get started you need to do the following:

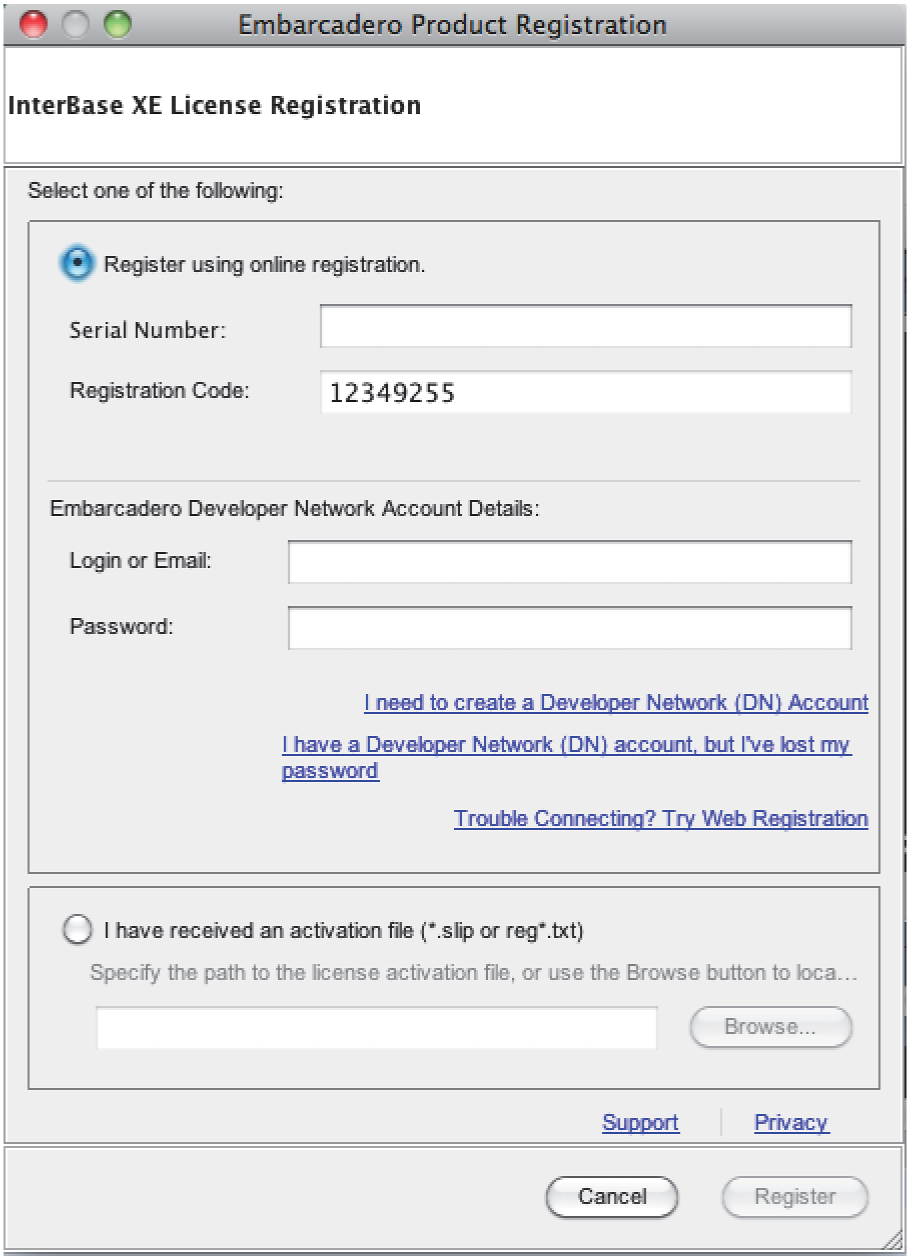
- A Developer's Network account is required. Click http://edn.embarcadero.com to either log on to an existing account or create a new one.
- Go to https://downloads.embarcadero.com/free/interbase and download InterBase XE (10.0.4.590) 32-bit Developer Edition for Mac 0S X.The file is **InterBase_XE_Mac-2.zip**.

INSTALLATION

1. Double-click the InterBase_XE_Mac-2.zip file and the InterBase_XE_Mac-2 dialog opens.
2. Double-click **ib_install.app** and a warning dialog appears asking you to confirm the installation. Click **Open** and dialog opens requesting your password.

   NOTE: The installation requires an Administrator password to install the software.

3. Click the **Lock** icon and enter your password. Once the password is authenticated the install wizard appears.
4. Follow the wizard and in the **Welcome** pane, after you read the instructions, click **Next**.
5. Accept the License Agreement and click **Next**.
6. In the **Choose Install Set** pane, select the **Server and Client** option. This is the default setting.
7. Leave the default setting of **"No"** in the Multi-Instance InterBase pane and click **Next**.
8. On the **Choose Install Folder** pane, select **Restore Default Folder** which is *Macintosh HD > Applications > InterBase*.
9. Click **Next** and review the **Pre-Installation Summary** and then click **Install**.
10. During the install the Embarcadero Product Registration dialog appears.



Embarcadero Product Registration

InterBase XE License Registration

Select one of the following:

● Register using online registration.

Serial Number:	
Registration Code:	12349255

Embarcadero Developer Network Account Details:

Login or Email:	
Password:	

I need to create a Developer Network (DN) Account

I have a Developer Network (DN) account, but I've lost my password

Trouble Connecting? Try Web Registration

○ I have received an activation file (*.slip or reg*.txt)

Specify the path to the license activation file, or use the Browse button to loca...

11 Enter the Serial Number that was sent to your EDN account. You also need to enter your Login and Password.

12 Click **Register**. Once the registration is complete you are notified that you now have the InterBase XE Server Edition 15 Day Trial.

13 When the install is complete, click **Done**.

14 InterBase for Mac is a command-line application. To start the server go to **Macintosh HD > Applications > InterBase > bin > ibmgr**.

INSTALLING INTERBASE XE FOR LINUX

Before you get started you need to do the following:

- A Developer's Network account is required. Click http://edn.embarcadero.com to either log on to an existing account or create a new one.
- Go to https://downloads.embarcadero.com/free/interbase and download InterBase XE 32-bit Developer Edition for Linux.The file is InterBase_XE_Linux.tar.gz.
- After extracting the Linux Tar, run **install_linux.sh**. This ensure that the sanctuary library for registration is set in the environment.

INSTALLATION

1 Run the script install_linux.sh as a root user.

NOTE: The installation requires an Administrator password to install the software.

2 In the InterBase XE License Registration enter the Serial Number that was sent to your EDN account. You also need to enter your Login and Password.

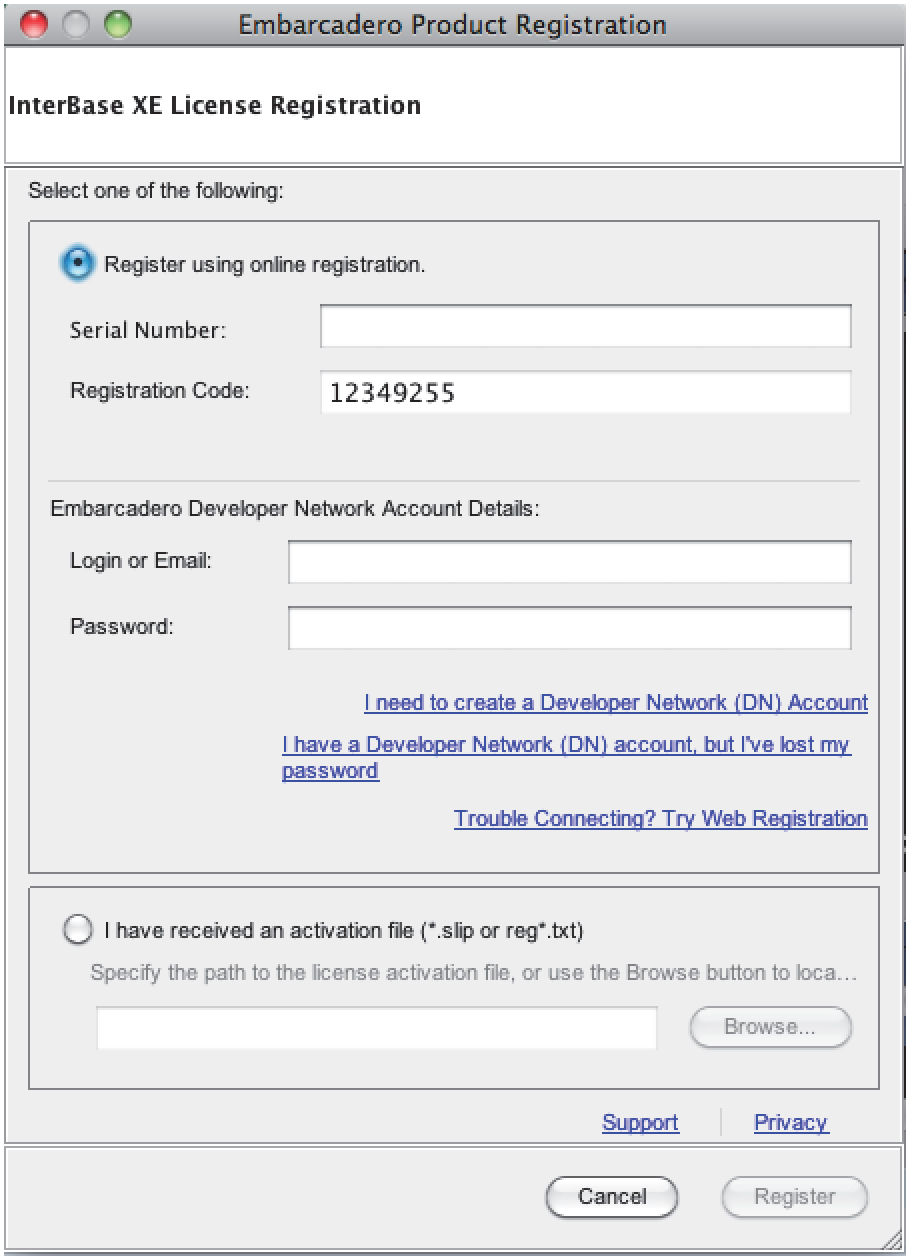
3 Follow the wizard and in the **Choose Install** pane, select the **Server and Client** option. This is the default setting.

4 Leave the default setting of "No" in the Multi-Instance InterBase pane.

5 On the **Choose Install Folder** pane, select **Restore Default Folder**.

6 Review the Pre-Installation Summary and then click **Install**.

7 In the InterBase XE License Registration enter the Serial Number that was sent to your EDN account. You also need to enter your Login and Password.



Embarcadero Product Registration

InterBase XE License Registration

Select one of the following:

● Register using online registration.

Serial Number:

Registration Code: 12349255

Embarcadero Developer Network Account Details:

Login or Email:

Password:

I need to create a Developer Network (DN) Account

I have a Developer Network (DN) account, but I've lost my password

Trouble Connecting? Try Web Registration

○ I have received an activation file (*.slip or reg*.txt)

Specify the path to the license activation file, or use the Browse button to loca...

8 Click **Register**. Once the registration is complete you are notified that you now have the InterBase XE Server Edition 15 Day Trial.

9 Click **Finish** and you are returned to the install wizard. Click **Done**

10 InterBase for Linux is command-line application. To start the server go to: **opt > InterBase > bin > ibmgr**.

Advance to Part II

This completes Part I of the Quick Start. In Part III, you will cover the following:

- Data modeling overview
- Creating domains
- Executing SQL scripts
- Creating tables and views
- Viewing object definitions
- Altering a table
- Creating, modifying and dropping indexes.

PART II: DATA DEFINITION

Part II covers the following topics:

DATABASE DESIGN

The crucial first step in constructing any database is *database design*. You can't create a functional, efficient database without first thinking through all its components and desired functionality in great detail. Chapter 2 of the Data Definition Guide provides a good introduction to the topic.

A Quick Look at Data Modeling

This following list provides a brief and simple overview of the process of designing a database:

1 **Determine data content**: What information needs to be stored? In thinking about this, look at it from the end user's point of view: What groups of end users will access the database? What information will they need to retrieve? What questions will they be asking of the database?

2 **Group types of data together**: Information items tend to group together naturally. Later in this Quick Start, when you create tables in the database, you will create one table for each group of data items. The granularity with which you divide the mass of information into groups depends on factors such as the quantity and complexity of the information your database must handle. The goal is to have each item of information in only one place. The process of identifying such groups is called *normalization*. Identify *entities* and their *attributes*. In this Quick Start, for example, one type of entity is the project. A project's attributes are its ID number, name, description, leader's name, and product. Later you will see that there is a table named *Project* that has columns named *proj_id*, *proj_name*, *proj_desc*, *team_leader*, and *product*.

3 **Design the tables**: Determine what tables you will create, what columns will be in each table, and what type of data each column will contain. If you have identified your entities and their attributes carefully, each entity corresponds to a table and each attribute is a column in that table. At this point you then decide on the datatype for each column, as well.

Is the data numeric or text? If it's numeric, what is the expected range of values? If it's text, how long a string do you need? Identify an appropriate datatype for each column. InterBase's supported datatypes are discussed in Chapter 4 of the *Data Definition Guide*.

4 **Consider the interdependencies of your table columns**: You can't sell an item, for example, unless you have it in an inventory. You can't deliver it unless it's in stock. You create primary keys and foreign keys to maintain these dependencies. This is called *maintaining database integrity*. Other mechanisms for maintaining database integrity and security include CHECK constraints, and using GRANT and ROLE statements to control access to tables.

THE TUTORIAL DATABASE

The TUTORIAL database that you create is, in fact, an exact copy of the EMPLOYEE database that is used for examples throughout the InterBase document set. The TUTORIAL database is a generic business database. Imagine that you are responsible for creating a database for this company. In the data modeling phase, you identify the following entities (information groups):

departments
salary history for each employee
department budgets for each project
jobs
projects
employees
countries
employee projects
customers
sales

As you progress through this Quick Start, you will see that the TUTORIAL database contains ten tables that correspond exactly to the ten items above. To get an overview now, you can look at the EMPLOYEE database, since the EMPLOYEE database and the finished TUTORIAL database are identical. The EMPLOYEE database is located in the `\Documents and Settings\All Users\Application Data\Embarcadero\InterBase\gds_db\examples` directory on your machine if you installed the examples.

AN OVERVIEW OF SQL

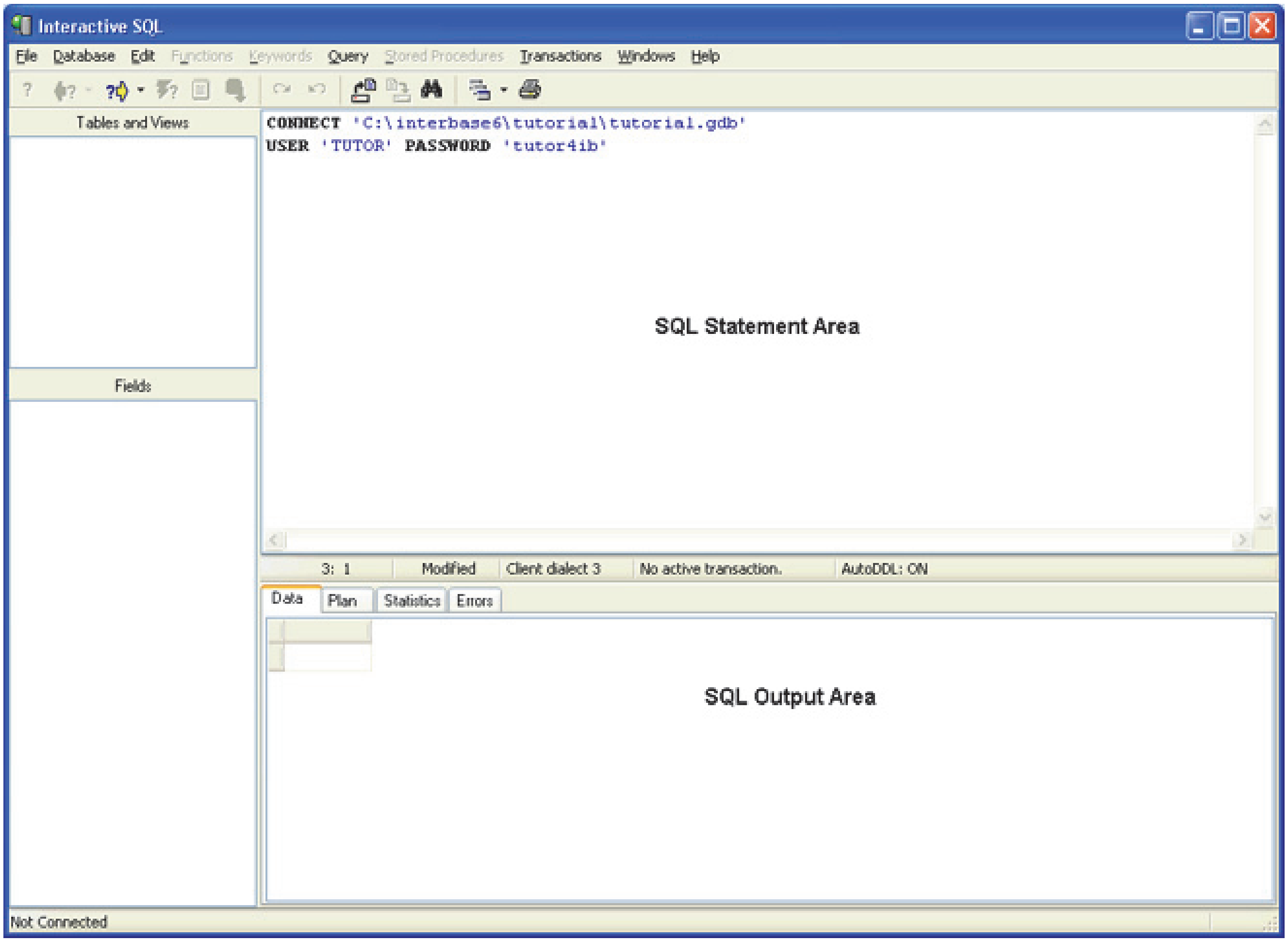
SQL statements fall into two major categories: Data definition language (DDL) statements, and Data manipulation language (DML) statements.

- **DDL statements** define, change, and delete the structures that hold data. These include the database itself, tables, and other elements that are part of the database such as domains, indexes, triggers, stored procedures, roles, and shadows. Collectively, the objects defined with DDL statements are known as *metadata*. DDL statements begin with the keywords CREATE, ALTER, and DROP. For example, CREATE TABLE defines a table, ALTER TABLE modifies an existing table, and DROP TABLE deletes a table.
- **DML statements** manipulate data within these data structures. The four fundamental DML statements are INSERT, UPDATE, DELETE, and SELECT. INSERT adds data to a table, UPDATE modifies existing data, and DELETE removes data. The SELECT statement retrieves or *queries* information from the database. It is the most important, and most complex, of all the SQL statements, because it is the means by which you gain access to all the information that you have so meticulously stored.

In this part of the Quick Start, you will use several DDL statements—CREATE DOMAIN, CREATE TABLE, ALTER TABLE, CREATE VIEW, and CREATE INDEX—to create data structures for your TUTORIAL database.

USING IBCONSOLE TO EXECUTE SQL

This section describes how to use the Interactive SQL to enter, execute, and commit SQL statements.



```
CONNECT 'C:\interbase6\tutorial\tutorial.gdb'
USER 'TUTOR' PASSWORD 'tutor4ib'
```

Entering SQL Statements in the SQL Statement Area

You enter SQL statements by typing them in the SQL statement area in the Interactive SQL dialog.

- You do not need to end statements with a terminator, such as a semicolon. However, the terminator is not prohibited and does not cause a problem. You can, for example, copy and paste statements from scripts (where they must have terminators) and run them without removing the terminators.
- InterBase SQL statements are not case sensitive. You can enter all SQL statements in lowercase if you prefer.
- *Anything inside quotation marks is case sensitive* and must be entered as shown.
- InterBase ignores line breaks within statements. They are for your convenience only.
- You can use spaces to indent lines to make them easier to read. InterBase ignores these spaces. You cannot indent using tabs.

- You must execute each statement before entering the next one.

Executing Statements

To execute a statement, do one of the following:

- Click Execute (⚡?)

  OR

- Select **Query > Execute.**

  You must execute each statement before entering the next one.

  NOTE: Although this method of entering SQL statements by hand one at a time is an option in InterBase, users often use SQL scripts (data definition files) as a more convenient way of entering data. See "Running an SQL Script" on page 9 for more information.

Committing your Work

Until you commit your work, your transaction is said to be *active*. Work associated with an active transaction is not yet visible to other users. You often want to enter an entire group of related items before committing, so that misleading intermediate states are never visible. When you commit, your transaction changes to a committed state and the work you did in that transaction becomes visible to other users. When you are creating metadata, however, you usually want to commit each data structure as you complete the DDL statement. (See page 3 for a discussion of DDL and DML statements.)

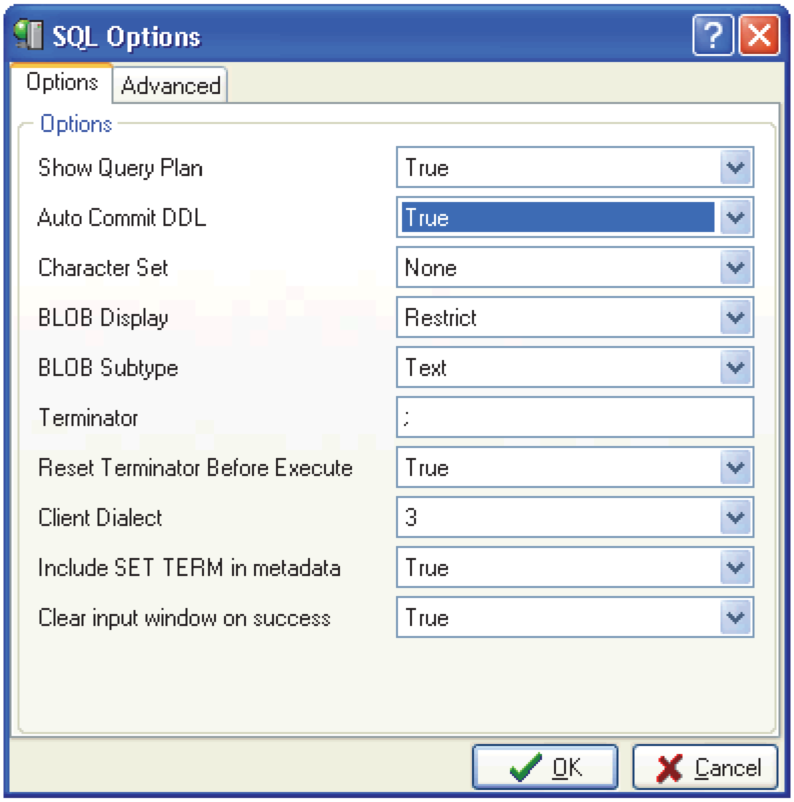
InterBase provides an Autocommit feature that automatically commits any DDL statement when you execute it. The Autocommit feature does not apply to DML statements such as INSERT, UPDATE, DELETE, and SELECT. The Autocommit feature is enabled by default in InterBase.

Checking Session Settings

To check the status of the Autocommit feature and other session settings, follow these steps:

1 Click SQL to open the Interactive SQL dialog.

2 Choose **Edit > Options** to display the SQL Options dialog.



Show Query Plan	True
Auto Commit DDL	True
Character Set	None
BLOB Display	Restrict
BLOB Subtype	Text
Terminator	;
Reset Terminator Before Execute	True
Client Dialect	3
Include SET TERM in metadata	True
Clear input window on success	True

3 Check to see that the Auto Commit DDL is set to **True**. If not, enable the feature and click **OK**.

Creating Domains

A *domain* is a customized column definition that you use to create tables. When you create a table, you specify the characteristics of each column in the table. Often, across the tables in a database, there will be several columns that have the same characteristics. Rather than entering the same complex definition for each column, you can create a name for the collection of characteristics. This named set of column characteristics is called a domain. You can use this domain name in a column definition rather than typing out the full definition.

NOTE: If you are going to use domains in your column definitions, you must create the domains before you use them in table definitions.

Column characteristics include:

- Datatype
- Default value: a literal value, NULL, or the name of the current user (USER)
- Nullability: NOT NULL prohibits NULL values in the column (columns are nullable by default)
- CHECK constraints: checks that the value being entered meets specified criteria

- Character set and optional collation order (output sort order for CHAR and VARCHAR columns)
- When you specify a column, only a column name and datatype are required. All other characteristics are optional.

Connecting to the TUTORIAL database

1 If you are not already connected to the Tutorial database, in IBConsole, log on as the TUTOR user to the Local InterBase server.

- Select the Local Server in the left pane and then choose **Server > Login** to display the Server Login dialog.
- The user name TUTOR is already in the first field. Enter the password "tutor4ib" and click **Login**.

2 To connect to the TUTORIAL database highlight the Tutorial.gdb icon in the left pane and choose **Database > Connect**.

Entering Metadata Statements

In this exercise, you use the CREATE DOMAIN statement to create domains that you will use later to specify column datatypes.

Create some Domains

In the following exercise you will define four domains. The first three specify only a datatype. The fourth one is more complex. In each case, the domain will be useful for several different columns, not just the column for which it is named.

NOTE: You should be connected to the TUTORIAL database when you begin this exercise.

1 Click on the SQL icon (SQL) and type the following code in the SQL Statement Area to define a domain called FIRSTNAME that has a datatype of VARCHAR(15).

```
CREATE DOMAIN FIRSTNAME AS VARCHAR(15)
```

2 Execute the statement: click ⚡?, press F5, or choose **Query > Execute**.

Now create two more domains, LASTNAME and EMPNO. Execute each statement separately before entering the next one.

```
CREATE DOMAIN LASTNAME AS VARCHAR(20)

CREATE DOMAIN EMPNO AS SMALLINT
```

3 Next, enter and execute the following code to define a domain for department numbers. The domain is defined as a three-character string. In addition to the datatype, it includes check constraints to ensure that the department number is either "000", alphabetically between "0" and "999", or NULL. Pay attention to parentheses and quotes as you enter this:

```
CREATE DOMAIN DEPTNO AS CHAR(3)
   CHECK (VALUE = '000'
      OR (VALUE > '0' AND VALUE <= '999')
      OR VALUE IS NULL)
```

TIP: When you're typing an SQL statement that has parentheses, take a moment to count the left parentheses and the right parentheses and make sure that there are the same number of each. Mismatched parentheses are a major source of errors in SQL code. In the example above, there are nested parentheses: the CHECK clause is enclosed in parentheses because it contains three parts ("A OR B OR C") and the second part of the clause has parentheses because it also contains multiple parts ("A AND B").

The CREATE DOMAIN statement above is divided into several lines to make it easy for users to follow the syntax. InterBase ignores the line breaks when parsing the statement. Enter the whole statement before executing it.

You don't need to commit your work, because CREATE statements are DDL (data definition language) statements. You turned *Autocommit DDL* on in the Session Settings earlier in this tutorial, so all these DDL statements have been committed automatically.

NOTE: You can manually commit your SQL statements, making them permanent by selecting Transactions > Commit. This commits any uncommitted DML and DDL statements.

Data Definition Files

Since you've already created several domains, you can use an SQL script—also called a *data definition file*—to create the rest of the domains. A data definition file is a text file that contains SQL statements. It can be executed in IBConsole and is typically created with a text editor such as Notepad.

It is often convenient to create a data definition file rather than typing each statement directly into IBConsole, because the text editor provides you with an editing environment and the script provides a reusable record of what was entered. In practice, most data definition is accomplished using data definition files.

TIP: Every SQL script must begin with a CONNECT statement. The CONNECT statement specifies a database name including the complete path, a user name, and password. The SQL scripts that accompany this tutorial begin with the following CONNECT statement:

```
CONNECT 'C:\Documents and Settings\All Users\Application
```

```
Data\Embarcadero\Interbase\gds_db\tutorial.gdb'
USER 'TUTOR' PASSWORD 'tutor4ib'
```

If this is not the correct information for you, you must edit each SQL file and make the CONNECT string correct. The SQL scripts are text files that you can modify in any text editor. If you use an application that saves by default in a proprietary format, be sure to save the files as text.

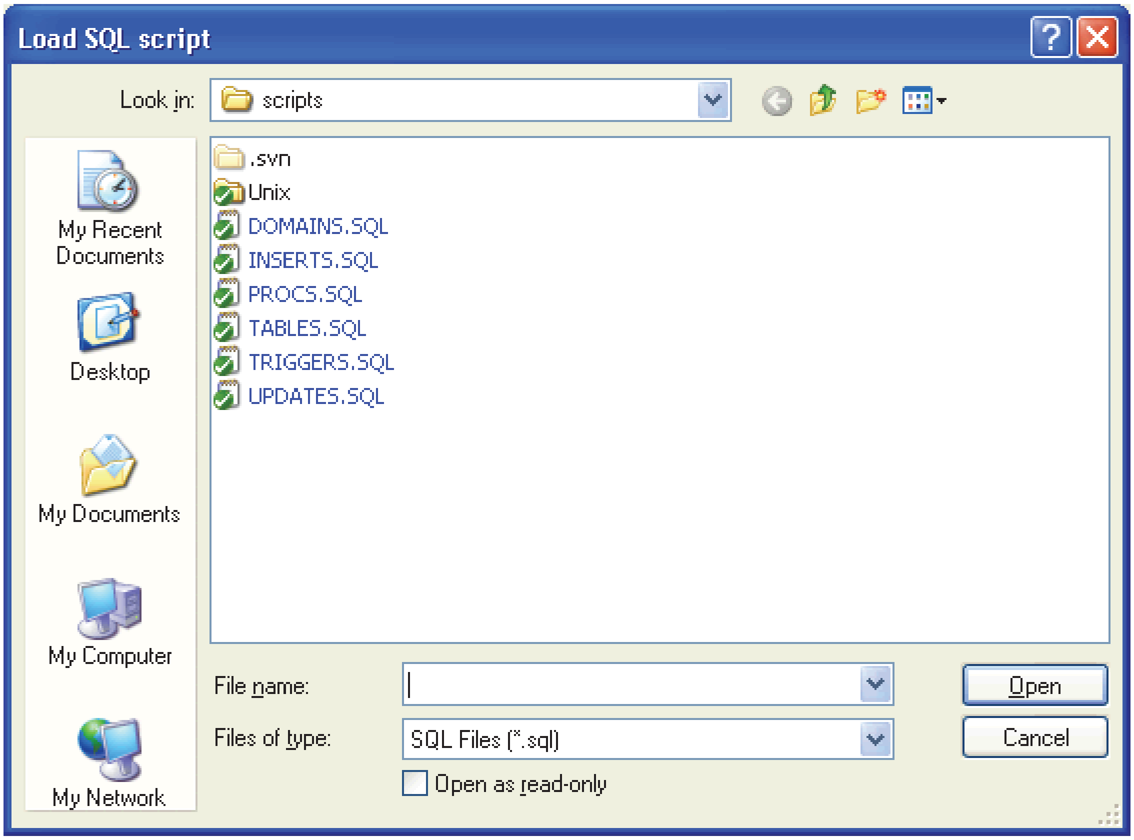
Running an SQL Script

Before running an ISQL script, it is good practice to open the file in a text editor and check that the CONNECT statement provides the correct server, database, user name, and password. Edit the information if necessary.

The remaining domain definitions that are needed for your TUTORIAL database are in the **Domains.sql** data definition file. In this section of the tutorial, you execute that file to create the remaining domain definitions.

Run the Domains.sql Script

1. Open **Domains.sql** in a text editor and make sure that the CONNECT statement specifies the correct path, database name, user name, and password.
2. Notice that the file contains many descriptive passages that are commented out. The convention for comments is exactly like that for the C language: comments begin with `/*` and end with `*/`.
3. Look through the file and notice that each SQL statement ends with a semicolon (`;`). Semicolons are required at the end of each statement in a data definition file. They are not required when you type statements directly into IBConsole.
4. Choose **Query > Load Script**. The **Load SQL Script** dialog displays:



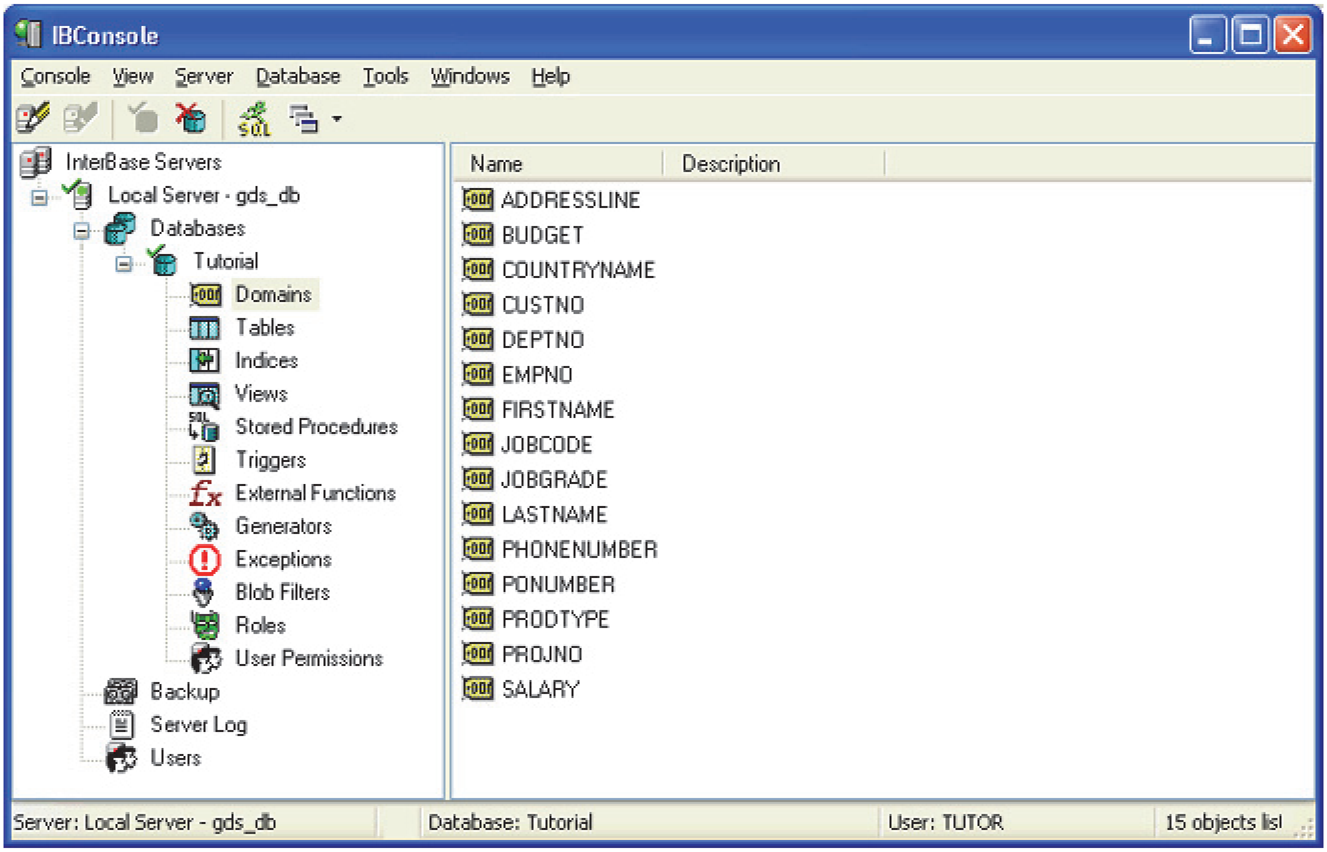
5 Navigate to the location where you have stored the tutorial SQL files, highlight **Domains.sql** and click **Open**.

6 Choose **Query > Execute**.

Troubleshooting: If you receive a message stating that there are errors, read the log file, in this case **Domains.log**, which is located in the \scripts directory. Also check that the CONNECT information (database, user, and password) in the script file is correct.

7 To confirm that the domains now exist, click on Domains in the left pane of IBConsole.

You should see all the domains defined for the database displayed in the right pane of IBConsole. You should see the following domains:



Name	Description
ADDRESSLINE	
BUDGET	
COUNTRYNAME	
CUSTNO	
DEPTNO	
EMPNO	
FIRSTNAME	
JOBCODE	
JOBGRADE	
LASTNAME	
PHONENUMBER	
PONUMBER	
PRODTYPE	
PROJNO	
SALARY	

Server: Local Server - gds_db | Database: Tutorial | User: TUTOR | 15 objects lis

NOTE: You can choose to show or hide the systems data (for example the RDB$ domains) by choosing **IBConsole > View > System Data**.

CREATING TABLES

A table is a data structure consisting of an unordered set of rows, each containing a specific number of columns. Conceptually, a database table is like an ordinary table. Much of the power of relational databases comes from defining the *relations* among the tables.

The CREATE TABLE statement has the following general form:

CREATE TABLE tablename (*colname1 characteristics*[, *colname2 characteristics*, …][, *tableconstraint* …])

- *Characteristics* must include a datatype and can also include several other things. See “Creating Domains” on page 6 for a list of column characteristics.
- A *tableconstraint* can be a CHECK, UNIQUE, FOREIGN KEY, or PRIMARY KEY constraint on one or more columns.

For the full syntax of the CREATE TABLE statement, see the Language Reference.

In the following steps, you create three of the ten tables for the TUTORIAL database.

Creating the Country table

The first table—the *Country* table—has two columns. The column definitions are separated by commas. For each column, the first word is the column name and the following words are characteristics. The first column, *country*, has the COUNTRYNAME domain and in addition is NOT NULL and a primary key. Primary keys are discussed in a following section of this tutorial.

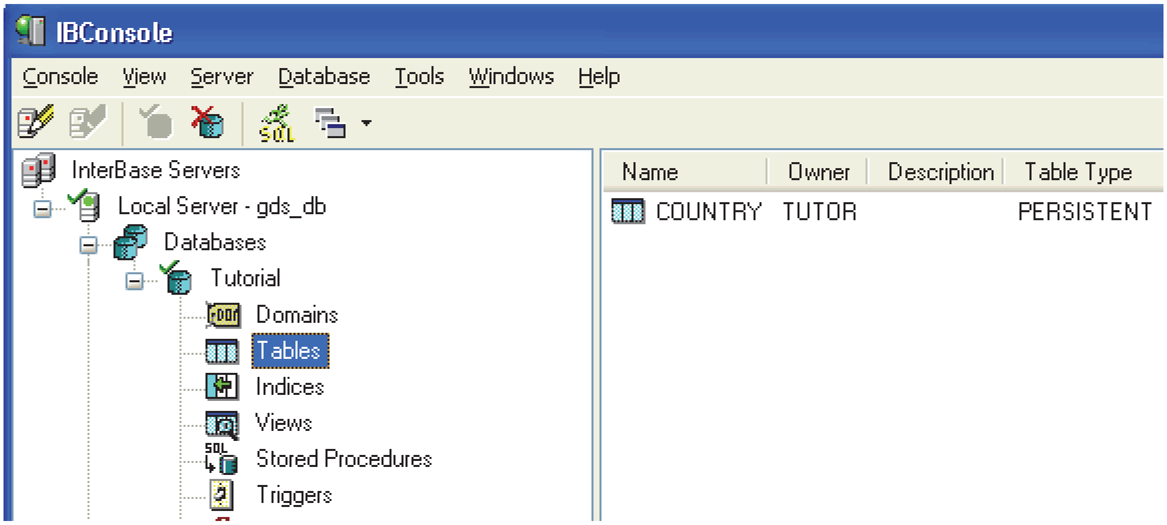
1 If you are not already connected, in IBConsole, connect to tutorial.gdb as TUTOR.

2 In the Interactive SQL dialog enter the following CREATE TABLE statement. The layout below is for ease of reading; the line endings are not required:

```
CREATE TABLE Country (
   country COUNTRYNAME NOT NULL PRIMARY KEY,
   currency VARCHAR(10) NOT NULL)
```

Notice that the collection of column definitions is surrounded by parentheses and that the columns are separated by commas.

3 Execute the statement (click , press F5 or choose **Query > Execute**). If you entered the code without errors, the tables now exist in the database in IBConsole. Choose **Tables** from the left pane.



Name	Owner	Description	Table Type
COUNTRY	TUTOR		PERSISTENT

Creating the Department table

Next, you create the *Department* table. This table has only two columns to begin with. Later, you use the ALTER TABLE command to add to it. Type in the complete SQL statement and then execute it:

```
CREATE TABLE Department (
   dept_no DEPTNO NOT NULL PRIMARY KEY,
   department VARCHAR(25) NOT NULL UNIQUE)
```

The *dept_no* column is the primary key for the table and is therefore UNIQUE. Primary keys are discussed on page 14. Notice that the *department* column value must be unique and that neither column can be null.

Creating the Job table

Now you create the more complex *Job* table. This definition includes CHECK constraints, PRIMARY KEY and FOREIGN KEY constraints and a BLOB datatype for storing descriptive text. The text following the code discusses these new elements.

1 Enter the following CREATE TABLE statement in the Interactive SQL statement area. Type in the whole statement and then execute it.

```
   CREATE TABLE Job (job_code JOBCODE NOT NULL,
      job_grade JOBGRADE NOT NULL,
      job_country COUNTRYNAME NOT NULL,
      job_title VARCHAR(25) NOT NULL,
      min_salary SALARY NOT NULL,
      max_salary SALARY NOT NULL,
      job_requirement BLOB SUB_TYPE TEXT SEGMENT SIZE 400,
      language_req VARCHAR(15)[1:5],
   CONSTRAINT pkjob PRIMARY KEY (job_code, job_grade, job_country),
   CONSTRAINT fkjob FOREIGN KEY (job_country) REFERENCES Country
(country),
   CHECK (min_salary < max_salary))
```

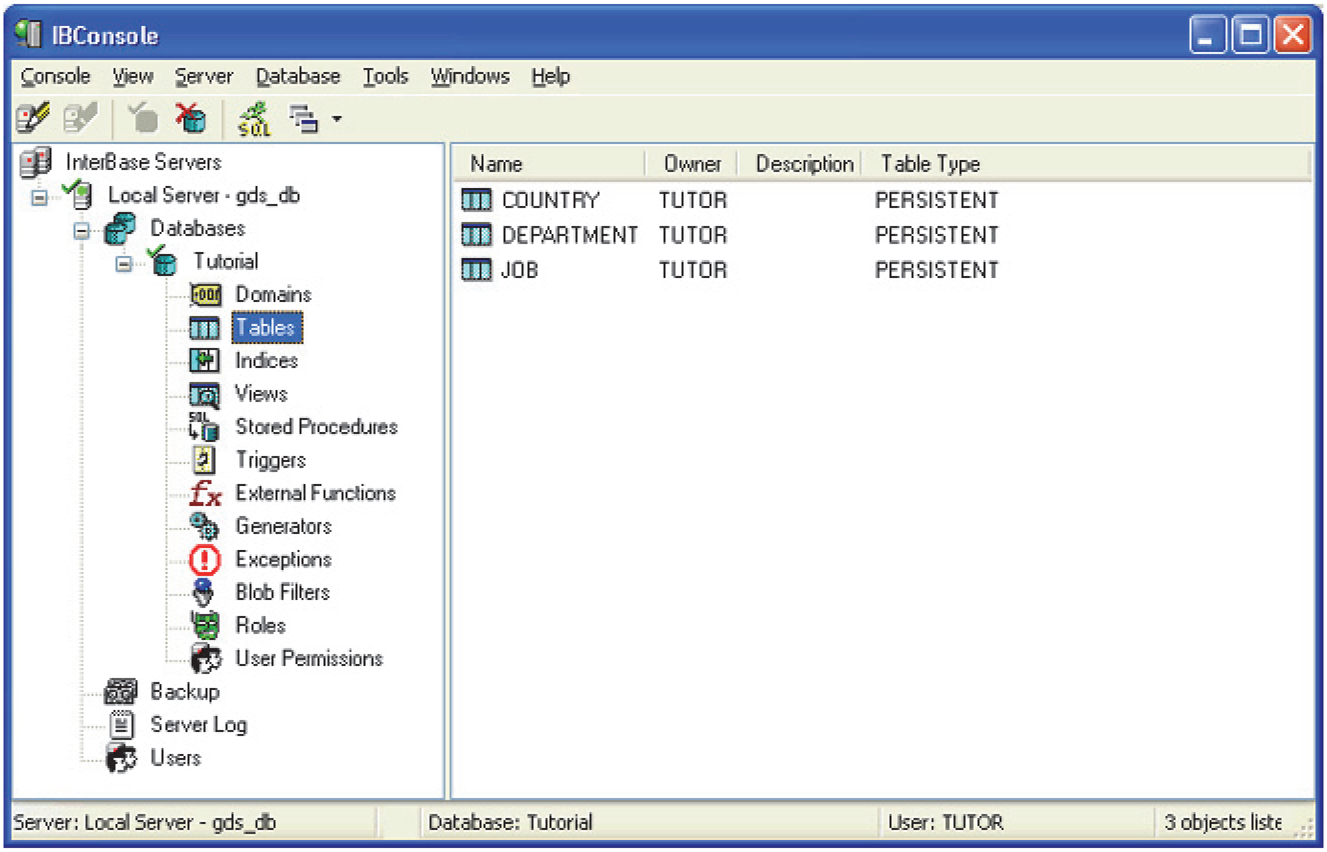
The CHECK constraint at the end checks that the minimum salary is less than the maximum salary.

The three-column primary key guarantees that the combination of the three columns identifies a unique row in the table.

The foreign key checks that any country listed in the *Job* table also exists in the *Country* table.

The BLOB datatype used for the *job_requirement* column is a dynamically sizable datatype that has no specified size and encoding. It is used to store large amounts of data such as text, images, sounds, and other multimedia content.

2 To check that the tables now exist in the database, in IBConsole, choose Tables from the left pane.



Name	Owner	Description	Table Type
COUNTRY	TUTOR		PERSISTENT
DEPARTMENT	TUTOR		PERSISTENT
JOB	TUTOR		PERSISTENT

Server: Local Server - gds_db | Database: Tutorial | User: TUTOR | 3 objects liste

Primary Keys and Foreign Keys

The *Job* table definition includes a primary key and a foreign key.

- A *primary key* is a column or group of columns that uniquely identify a row. Every table should have a primary key. And a table cannot have more than one primary key. The PRIMARY KEY characteristic can be specified as part of a column definition, as in the first column of the *Country* table, or it can be specified as a separate clause of the CREATE TABLE statement, as in the statement that creates the *Job* table. The primary key in the *Job* table is made up of three columns: *job_code*, *job_grade*, and *job_country*. While a value can appear more than once in any of these columns taken individually, the fact that they are collectively a primary key means that the three values taken together cannot occur in more than one row.

- A *foreign key* is a column or set of columns in one table whose values *must* have matching values in the primary key of another (or the same) table. A foreign key is said to *reference* its primary key. Foreign keys are a mechanism for maintaining data integrity. In the *Job* table, for example, any country listed in the *job_country* column must also exist in the *Country* table. By stating that the *job_country* column of the *Job* table is a foreign key that references the *country* column of the *Country* table, you are guaranteeing this, because InterBase will return an error if a value is entered the *job_country* column that does not have a matching entry in the *country* column of the *Country* table.

  You can declare a constraint such as UNIQUE, FOREIGN KEY, or PRIMARY KEY either as part of a column definition, or as a *table constraint* following the column definitions. The syntax varies slightly depending on which you choose. See the Language Reference for details.

  You declared the primary key constraint as part of a column definition in the *Country* and *Department* tables. For the *Job* table, you declared the primary key, foreign key, and check constraints at the table level. Functionally, the effect is the same.

Cascading Referential Integrity Constraints

When you create a foreign key, you are saying that the value in the foreign key must also exist in the primary key that it references. What happens if later, the value in the referenced primary key changes or is deleted? The *cascading referential integrity constraints* let you specify what should happen. Your choices are to take no action, to propagate (cascade) the change to the foreign key column, to set the foreign key to its default, or to set it to NULL.

If you are specifying the foreign key as part of the column definition, the syntax is this:

```
CREATE TABLE table_name (column_name datatype FOREIGN KEY
   REFERENCES other_table(columns)
      [ON UPDATE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
      [ON DELETE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}],
   [, more columns defs])
```

If you are specifying the foreign key as a table-level constraint, the syntax is nearly the same except that you have to identify the column for which it is being defined, so the syntax becomes:

```
CREATE TABLE table_name (column_defs,
   FOREIGN KEY (column_name) REFERENCES other_table(columns)
      [ON UPDATE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
      [ON DELETE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}],
   [, more table constraints])
```

A little later, you will use ALTER TABLE to add columns and table constraints to the *Department* table, including some cascading referential integrity constraints.

Naming Constraints

When you declare a constraint at either the column level or the table level, you have the option of naming the constraint using the optional CONSTRAINT keyword, followed by a constraint name. When a constraint is named, you can drop it using the ALTER TABLE statement. In the *Job* table definition, two of the constraints have names “pkjob” and “fkjob”), but the CHECK constraint does not have a name (although it *could* have). When you alter the *Department* table a little later, you will add two named constraints.

COMPUTED COLUMNS

When you are creating a table, you can define columns whose value is based on the values of one or more other columns in the table. The computation can include any arithmetic operations that are appropriate to the datatypes of the columns.

- Open the script **Tables.sql** and look at the following column definition for the *Employee* table:

  ```
  full_name COMPUTED BY (last_name || ', ' || first_name)
  ```

  The value of the *full-name* column consists of the value in the same row of the *last_name* column plus a comma plus the value of the *first_name* column.

BACKING UP A DATABASE

This is a good time to back up your database, because you've finished entering some tables. In the next part of the tutorial, you run a script to create more tables. Throughout this tutorial, you will be instructed to back up your database frequently. That way, if you run into difficulties, you can restore the last correct version and try again.

In a production database, a full backup and restore performs several functions:

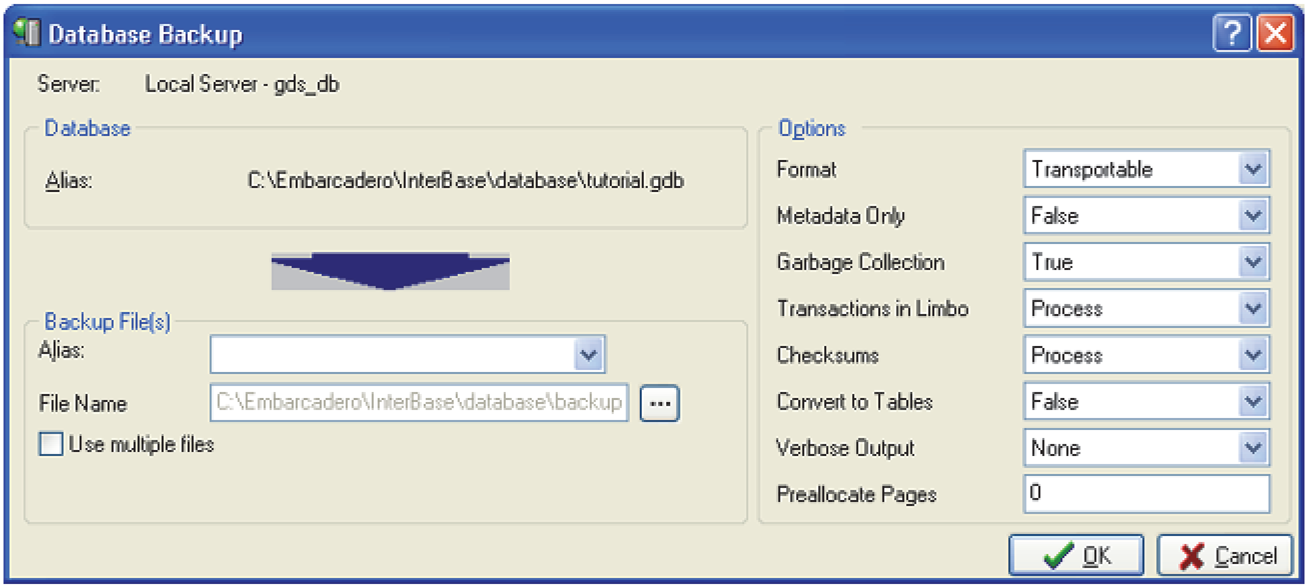
- It preserves your data by making a copy of both the data and the data structures (metadata).
- It improves database performance by balancing indexes and performing garbage collection on outdated records.
- It reclaims space occupied by deleted records, and packs the remaining data.
- When you restore, it gives you the option of changing the database page size and of distributing the database among multiple files or disks.

Important If you restore a database to a name that is already in use, be sure that no users are connected to it at the time you restore.

Backing up your Database

Before you begin, create a subdirectory called backups. If you are using the recommended directory path for this tutorial, your backups would be in `C:\Documents and Settings\All Users\Application Data\Embarcadero\InterBase\gds_db\backups`.

1. If it's not already open, open IBConsole by choosing it from the InterBase folder on the Start menu.
2. Log in as TUTOR to the server where your TUTORIAL database is located. (See "Connecting to a Server from IBConsole" on page 4 if you've forgotten how to do this.)
3. Right-click the Tutorial database and choose **Backup/Restore > Backup** to display the Database Backup dialog.



Server: Local Server - gds_db

Database

Alias: C:\Embarcadero\InterBase\database\tutorial.gdb

Backup File(s)

Alias:

File Name C:\Embarcadero\InterBase\database\backup

□ Use multiple files

Options

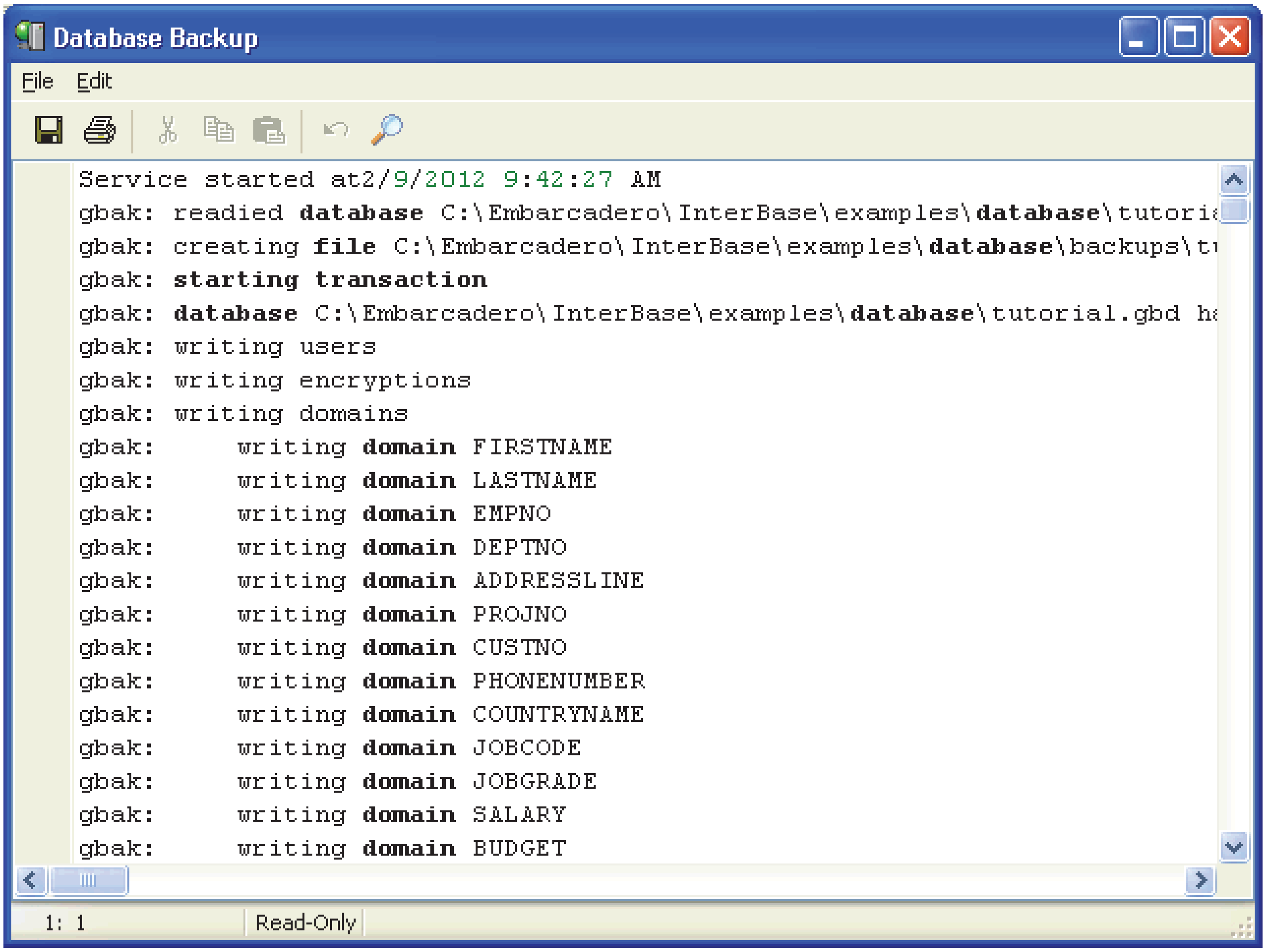
Format	Transportable
Metadata Only	False
Garbage Collection	True
Transactions in Limbo	Process
Checksums	Process
Convert to Tables	False
Verbose Output	None
Preallocate Pages	0

4 In the Alias field enter **Tutorial**

By convention, backups have a .gbk extent, but it is not required. From the Verbose Output, select **To Screen** from the drop-down list in order to see a detailed description of what InterBase does when it backs up a database

5 Browse to the location of your database backups (`C:\Documents and Settings\All Users\Application Data\Embarcadero\Interbase\gds_db\backups`) and enter **tutorial1.gbk.**

6 Click **Save** and then click **OK**. The backup actions are recorded in the Database Backup dialog. You also get a message that “Database backup completed.”



```
Service started at2/9/2012 9:42:27 AM
gbak: readied database C:\Embarcadero\InterBase\examples\database\tutoria
gbak: creating file C:\Embarcadero\InterBase\examples\database\backups\tu
gbak: starting transaction
gbak: database C:\Embarcadero\InterBase\examples\database\tutorial.gbd ha
gbak: writing users
gbak: writing encryptions
gbak: writing domains
gbak:     writing domain FIRSTNAME
gbak:     writing domain LASTNAME
gbak:     writing domain EMPNO
gbak:     writing domain DEPTNO
gbak:     writing domain ADDRESSLINE
gbak:     writing domain PROJNO
gbak:     writing domain CUSTNO
gbak:     writing domain PHONENUMBER
gbak:     writing domain COUNTRYNAME
gbak:     writing domain JOBCODE
gbak:     writing domain JOBGRADE
gbak:     writing domain SALARY
gbak:     writing domain BUDGET
```

7 The database backup file (tutorial1gbk) has been created in the following folder:

```
C:\Documents and Settings\All Users\Application
Data\Embarcadero\InterBase\gds_db\backups
```

Creating Tables with a Script

You’ve created several tables manually now and begin to understand what’s involved. To avoid having to type in all of the table definitions, you should now run the **Tables.sql** script.

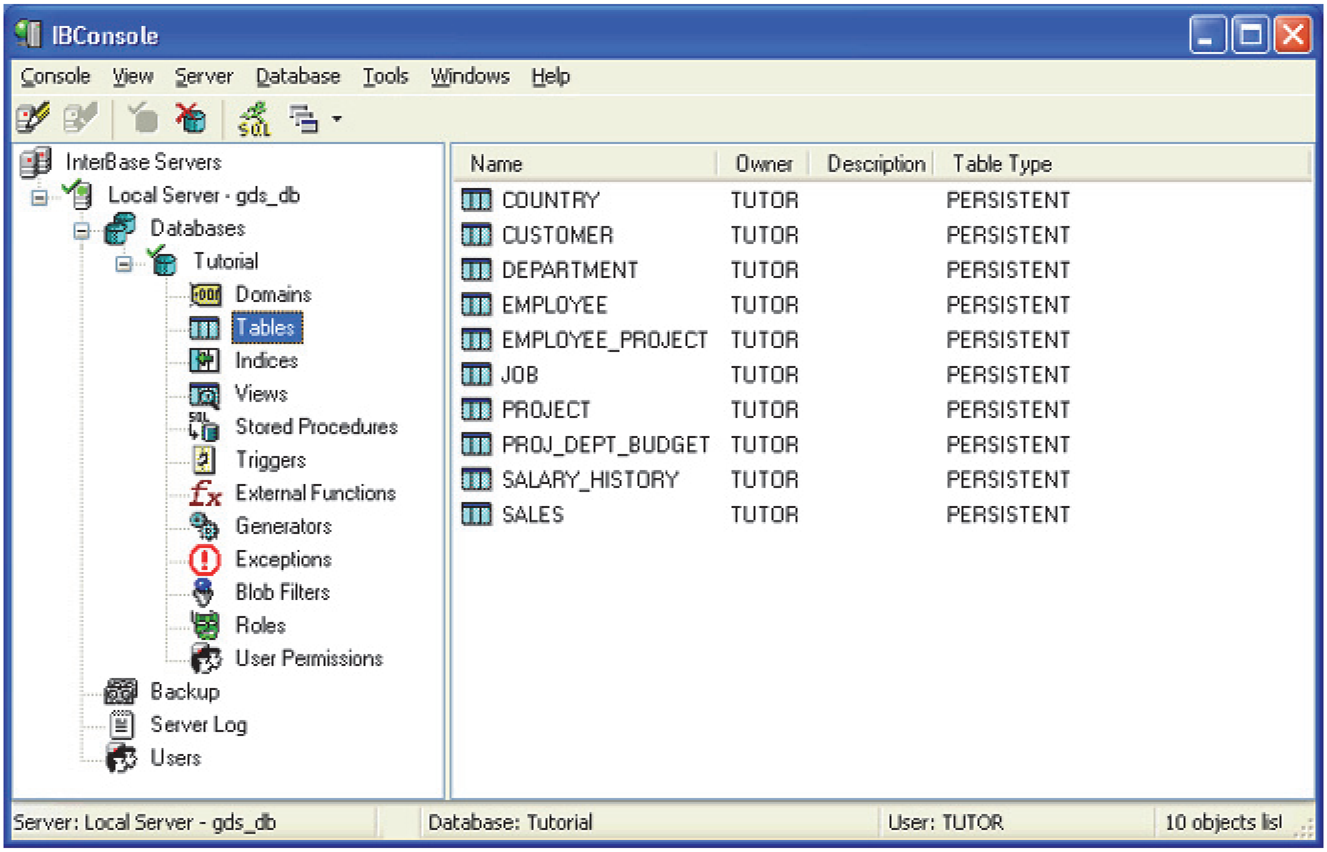
Running the Tables.sql Script

Before you leave the topic of tables, look over the remaining table definitions in **Tables.sql** to be sure that you understand them. Pay particular attention to the *Employee* table, which is complex and is central to this database. Notice, in particular, the complex CHECK constraint on the *salary* column in that table. It states that the salary entered for an employee has to be greater than the minimum salary for the employee's job (specified by *job_code*, *job_grade*, and *job_country*) and less than the corresponding maximum.

1. In the Interactive SQL dialog, load the **Tables.sql** script to enter the remaining table definitions into the TUTORIAL database by choosing **Query > Load Script.**

   As before, check first that the database path, user name, and password are correct in the CONNECT statement at the beginning of the file. Then choose **Query > Execute**.

2. Click on Tables in the left pane of IBConsole to check that you now have ten new tables in the TUTORIAL database.



Name	Owner	Description	Table Type
COUNTRY	TUTOR		PERSISTENT
CUSTOMER	TUTOR		PERSISTENT
DEPARTMENT	TUTOR		PERSISTENT
EMPLOYEE	TUTOR		PERSISTENT
EMPLOYEE_PROJECT	TUTOR		PERSISTENT
JOB	TUTOR		PERSISTENT
PROJECT	TUTOR		PERSISTENT
PROJ_DEPT_BUDGET	TUTOR		PERSISTENT
SALARY_HISTORY	TUTOR		PERSISTENT
SALES	TUTOR		PERSISTENT

Server: Local Server - gds_db   Database: Tutorial   User: TUTOR   10 objects lisl

NOTE: Time to back up again: If you have successfully run **Tables.sql**, this is a good time to back up your database to **Tutorial2.gbk**.

Troubleshooting

If you made any typing mistakes when you were entering the domain definitions, you'll get an error message when you run the **Tables.sql** script or when you are defining the tables manually. InterBase posts a message.

When executing a script, check the script's log file for error information. The log file is created in the same directory as the script file. When you are defining tables manually, look in the SQL Output area for more information. The SQL Output area echoes the contents of the script.

In either case, if there were problems with a particular table, the SQL code for that table is followed by an error message such as the following:

```
Statement failed, SQLCODE = -607

Dynamic SQL Error

-SQL error code = -607

-Invalid command

-Specified domain or source column does not exist
```

To understand the problem, follow these steps:

1. Read the error text. In this case, it says that the specified domain does not exist. You probably made an error in typing the domain name.
2. Click on Domains in the left pane of IBConsole. InterBase displays the names of all domains in the right pane.
3. You defined four domains by hand: FIRSTNAME, LASTNAME, EMPNO, and DEPTNO. Look for each of these in the list of domains and make sure that their names are spelled correctly. It's likely that you will find one that is misspelled.
4. Drop the incorrect domain by entering and executing the following command:

   ```
   DROP DOMAIN domainname
   ```

5. Recreate the domain using the CREATE DOMAIN statement.
6. Run the **Tables.sql** script again.

If this isn't the problem, continue with these steps:

1. Look right above the message text to see which table has the problem. Note which domains are used in that table. Do they include any of the four domains that you entered by hand?
2. Click on Domains in the left pane of IBConsole. InterBase displays the names of all domains in the right pane. Right-click on the domain in the right pane and select **Properties**. Click on the Metadata tab and InterBase displays the domain's definition in the SQL Output Area.
3. Compare the displayed definition with the definition given in this document (the one that you typed). Continue checking each of the four hand-entered domains until you find one that has a problem.

4 Drop the domain as described in step 4 above, and then re-enter it correctly. Run the **Tables.sql** script again.

Viewing an Object Definition

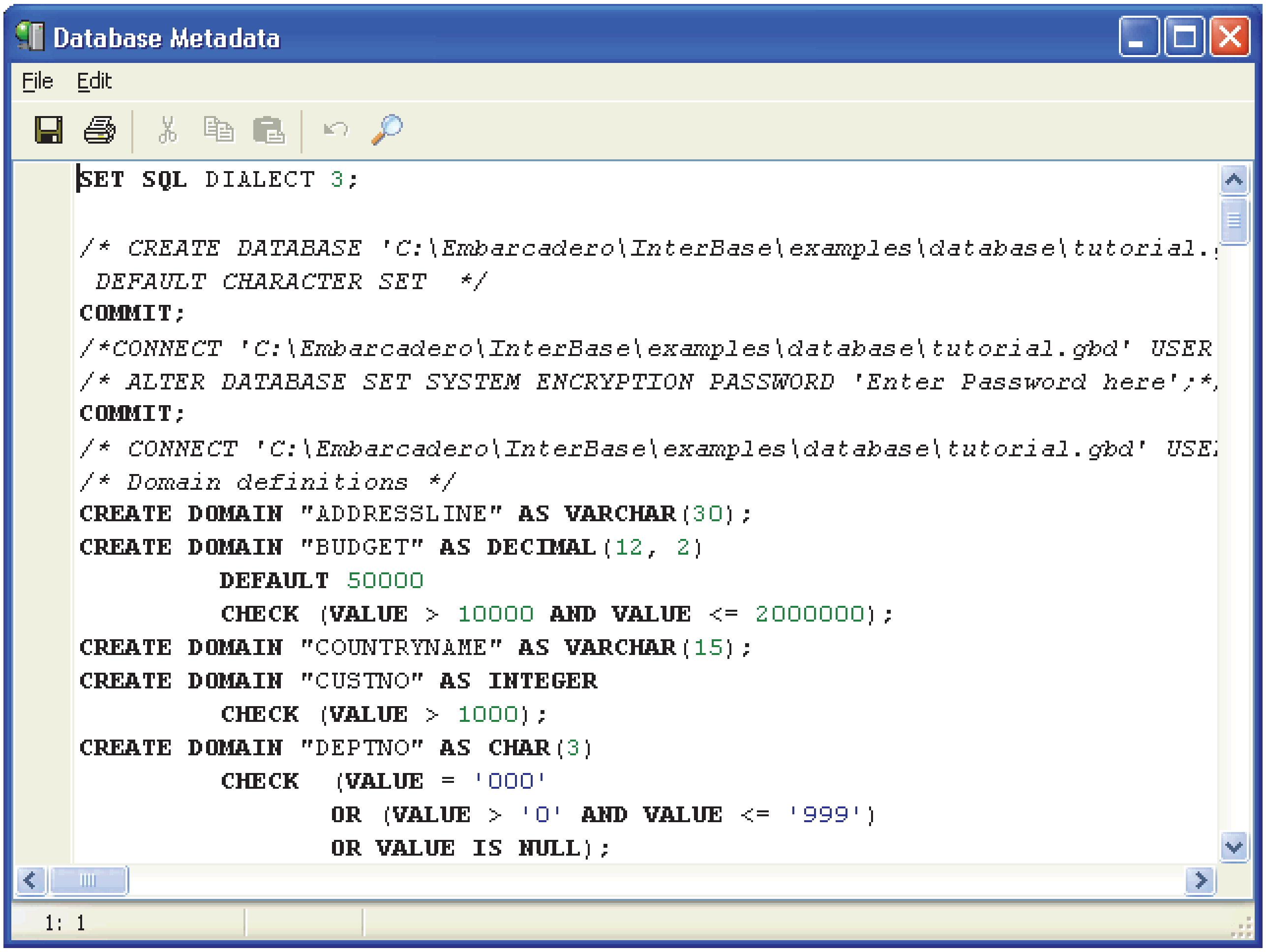
You can see the definition of any object in a database using the Metadata tab. Now you use it in a different way to get information about a specific object.

Viewing the Definition of the Department Table

To refresh your memory of the current *Department* table definition, follow these steps:

1 In IBConsole click **Tables** in the left pane.

2 Click Department and choose **Database > View Metadata** and the metadata for the `Department` table is displayed.



```
SET SQL DIALECT 3;

/* CREATE DATABASE 'C:\Embarcadero\InterBase\examples\database\tutorial.
 DEFAULT CHARACTER SET  */
COMMIT;
/*CONNECT 'C:\Embarcadero\InterBase\examples\database\tutorial.gbd' USER
/* ALTER DATABASE SET SYSTEM ENCRYPTION PASSWORD 'Enter Password here';*
COMMIT;
/* CONNECT 'C:\Embarcadero\InterBase\examples\database\tutorial.gbd' USE
/* Domain definitions */
CREATE DOMAIN "ADDRESSLINE" AS VARCHAR(30);
CREATE DOMAIN "BUDGET" AS DECIMAL(12, 2)
         DEFAULT 50000
         CHECK (VALUE > 10000 AND VALUE <= 2000000);
CREATE DOMAIN "COUNTRYNAME" AS VARCHAR(15);
CREATE DOMAIN "CUSTNO" AS INTEGER
         CHECK (VALUE > 1000);
CREATE DOMAIN "DEPTNO" AS CHAR(3)
         CHECK  (VALUE = '000'
                OR (VALUE > '0' AND VALUE <= '999')
                OR VALUE IS NULL);
```

ALTERING TABLES

You can change the structure of existing tables with the ALTER TABLE statement. In the previous section of the tutorial, you created a simple *Department* table. Now you can use the ALTER TABLE statement to alter to this table. The syntax for altering a table, in simplified form, is:

```
ALTER TABLE table_name operation [, operation]

where each operation is one of the following:
   ADD column
   ADD tableconstraint
   ALTER [column] col_name <alt_col_clause>
   DROP column
   DROP CONSTRAINT constraintname
```

Notice that you can drop a constraint only if you gave it a name at the time you created it.

Altering the Department Table

You now add five new columns (*head_dept*, *mngr_no*, *budget*, *location*, and *phone_no*) and two foreign key constraints to the *Department* table that you created earlier.

1 In Interactive SQL, type the following code and then execute it:

```
ALTER TABLE Department
   ADD head_dept DEPTNO,
   ADD mngr_no EMPNO,
   ADD budget BUDGET,
   ADD location VARCHAR(15),
   ADD phone_no PHONENUMBER DEFAULT '555-1234',
   ADD FOREIGN KEY (mngr_no)
      REFERENCES Employee (emp_no) ON DELETE CASCADE ON UPDATE
CASCADE,
   ADD CONSTRAINT fkdept FOREIGN KEY (head_dept)
      REFERENCES Department (dept_no) ON DELETE CASCADE ON UPDATE
CASCADE
```

2 In IBConsole, click on the Department table once again and then **Database > View Metadata** to see the new table definition.

MORE TROUBLESHOOTING

If you receive error messages when you are altering tables or inserting data, use the Metadata command as your resource.

- Show the definition for each table that you entered by hand and compare the output to the SQL code that this document instructs you to enter.
- When you find a problem, you can either drop the table and recreate it, or use ALTER TABLE to drop a column and then add the column again with the correct definition. If you misspelled the name of the table itself, you must drop the table and recreate it.

- The DROP TABLE statement has the following syntax:

```
DROP TABLE tablename
```

- To change a column definition, first drop it using the ALTER TABLE statement:

```
ALTER TABLE tablename DROP columnname
```

- Then add the column back in using the ALTER TABLE statement again:

```
ALTER TABLE tablename ADD columnname columndef
```

If you made any typing errors when creating the domains and tables, you will get errors when you try to insert data by hand or to run the **Inserts.sql** and **Update.sql** scripts. If you follow the steps above, you will be able to fix your errors and run the scripts successfully.

The remainder of the tutorial is less demanding, in that it focuses on the SELECT command. Once you detect and fix any errors in the domain and table definitions, you will get the correct results from your SELECT statements.

CREATING VIEWS

A view is a virtual table that contains selected rows and columns from one or more tables or views. InterBase stores only the definition of a view. The contents of a view are essentially pointers to data in the underlying tables. When you create a view, you are not copying data from the source tables to the view. You are looking at the original data.

A view often functions as a security device, because you can give people permissions on a view but not on the underlying tables. Thus, the people can access a defined part of the data (the part defined in the view), but the rest of the data remains private.

In the following exercise, you use the CREATE VIEW statement to create a phone list by choosing the employee number, first name, last name, and phone extension from the *Employee* table and the employee's location and department phone number from the *Department* table. Views are frequently created to store an often-used query or set of queries in the database.

You can select from a view just as you can from a table. Other operations are more restricted. See "Working with Views" in the Data Definition Guide for more on views.

Creating the Phone_list View

1 Enter the following statement to create the *Phone_list* view from selected columns in the *Employee* and *Department* tables.

```
         CREATE VIEW Phone_list AS
            SELECT emp_no, first_name, last_name, phone_ext, location,
      phone_no
            FROM Employee, Department
            WHERE Employee.dept_no = Department.dept_no
```

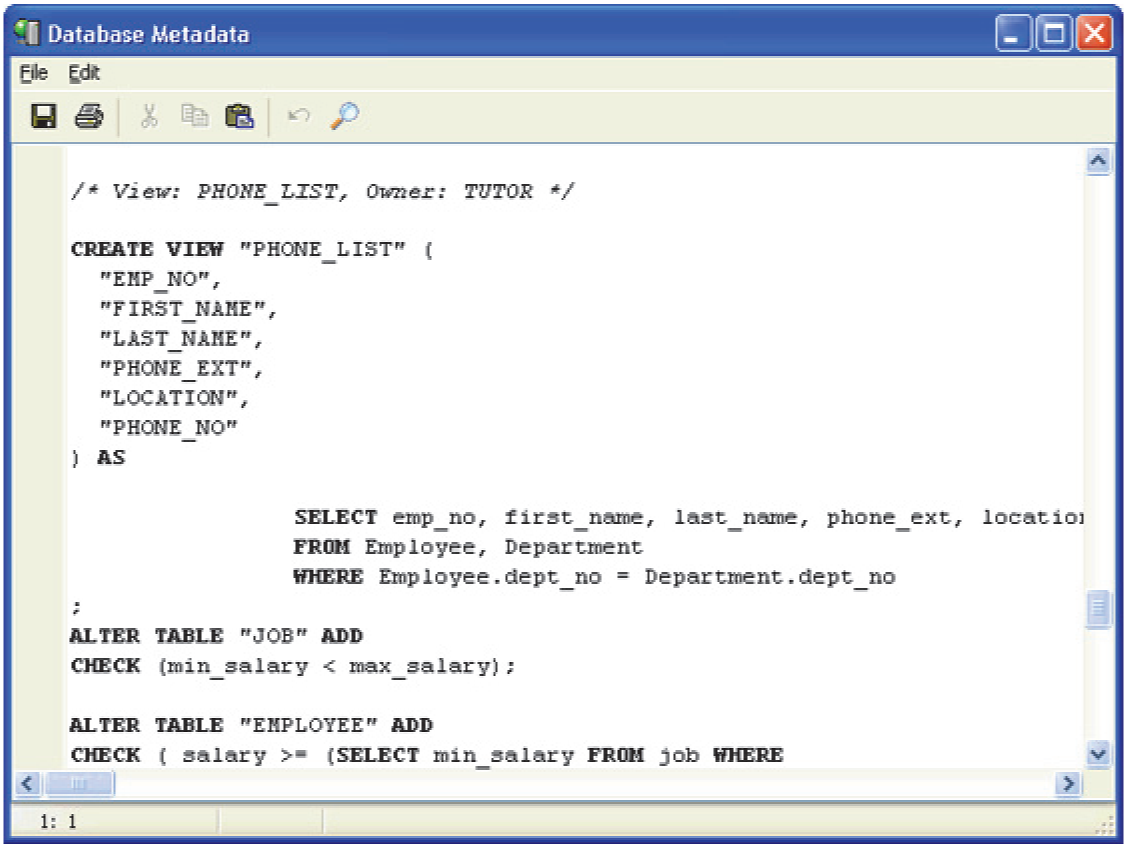
The WHERE clause tells InterBase how to connect the rows: the *dept_no* column in the *Department* table is a foreign key that references the *dept_no* column in the *Employee* table. Both columns are UNIQUE and NOT NULL, so the *dept_no* value in a *Department* row uniquely identifies a row in the *Employee* table. (In case you're wondering, the *dept_no* column in the *Employee* table is UNIQUE because all primary key columns automatically acquire the UNIQUE property.)

Notice that when the same column name appears in two tables in a query, you reference the columns by giving both the table name and the column name, joined by a period:

```
table_name.column_name
```

2 Now look at the structure of the *Phone_list* view by selecting the EMPLOYEE table and then selecting **Database > View Metadata**. You should see the following output:

```
/*  View: PHONE_LIST, Owner: TUTOR */

CREATE VIEW PHONE_LIST ( EMP_NO, FIRST_NAME, LAST_NAME, PHONE_EXT,
LOCATION, PHONE_NO ) AS SELECT emp_no, first_name, last_name,
phone_ext, location, phone_no    FROM Employee, Department
WHERE Employee.dept_no = Department.dept_no;
```



```
/* View: PHONE_LIST, Owner: TUTOR */

CREATE VIEW "PHONE_LIST" (
  "EMP_NO",
  "FIRST_NAME",
  "LAST_NAME",
  "PHONE_EXT",
  "LOCATION",
  "PHONE_NO"
) AS

                SELECT emp_no, first_name, last_name, phone_ext, locatio
                FROM Employee, Department
                WHERE Employee.dept_no = Department.dept_no
;
ALTER TABLE "JOB" ADD
CHECK (min_salary < max_salary);

ALTER TABLE "EMPLOYEE" ADD
CHECK ( salary >= (SELECT min_salary FROM job WHERE
```

Creating Indexes

An index is based on one or more columns in a table. It orders the contents of the specified columns and stores that information on disk in order to speed up access to those columns. Although they improve the performance of data retrievals, indexes also take up disk space and can slow inserts and updates, so they are typically used on frequently queried columns. Indexes can also enforce uniqueness and referential integrity constraints.

InterBase automatically generates indexes on UNIQUE and PRIMARY KEY columns. See the Data Definition Guide for more information about constraints.

You use the CREATE INDEX statement to create an index. The simplified syntax is as follows:

```
CREATE INDEX name ON table (columns)
```

Optionally, you can add one or more of the ASCENDING, DESCENDING, or UNIQUE keywords following the CREATE INDEX keywords.

Creating the namex Index

Define an index for the *Employee* table, by entering the following code:

```
CREATE INDEX namex ON Employee (last_name, first_name)
```

This statement defines an index called *namex* for the *last_name* and *first_name* columns in the *Employee* table.

Preventing Duplicate Index Entries

To define an index that eliminates duplicate entries, include the UNIQUE keyword in CREATE INDEX. After a UNIQUE index is defined, users cannot insert or update values in indexed columns if the same values already exist there.

For unique indexes defined on multiple columns, such as *prodtypex* in the example below, the same value can be entered within individual columns, but the combination of values entered in all columns of the index must be unique for each row

You cannot create a UNIQUE index on columns that already contain non-unique values.

Creating a UNIQUE Index

Create a unique index named *prodtypex*, on the *Project* table by entering the following:

```
CREATE UNIQUE INDEX prodtypex ON Project (product, proj_name)
```

Specifying Index Sort Order

By default, SQL stores an index in ascending order. To make a descending sort on a column or group of columns more efficient, use the DESCENDING keyword to define the index.

Creating the budgetx DESCENDING Index

Enter and execute the following code to create an index called *budgetx* that is in descending order:

```
CREATE DESCENDING INDEX budgetx ON Department (budget)
```

Modifying Indexes

To change an index definition—which columns are indexed, sort order, or UNIQUE requirement—you must first drop the index and then create a new index.

Altering the namex Index

Begin by viewing the current definition of the *namex* index.

1. Since *namex* was created on the Employee table, click Tables in the left pane of IBConsole.
2. Double-click **Employee**, and in the **Tutorial - Properties for: EMPLOYEE** dialog, click Indexes ( ).
3. Select the Metadata tab to view the definition and then you can exit this Properties dialog.

In the following steps, you redefine the *namex* index that you created earlier to include the UNIQUE keyword.

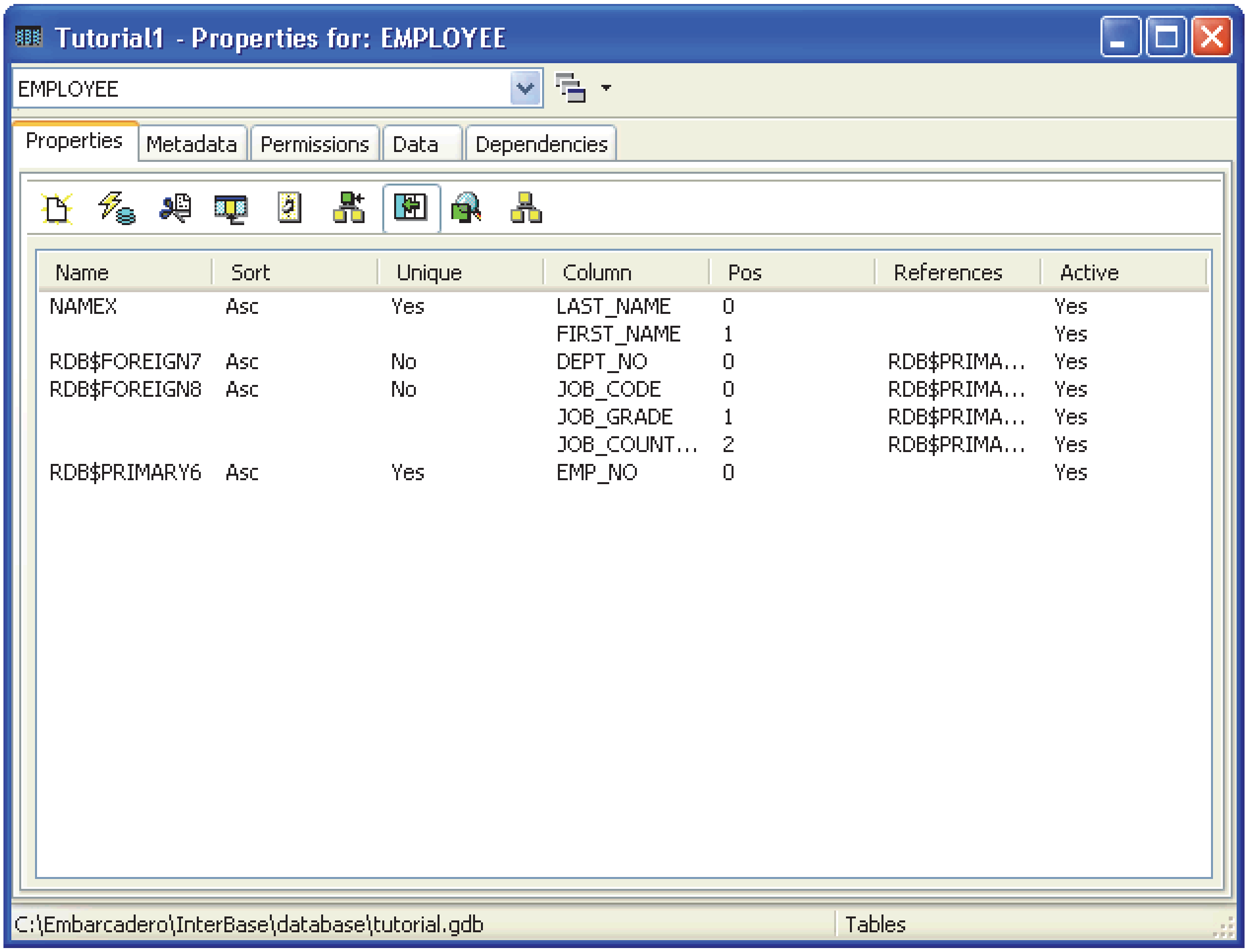
1 In the ISQL enter and execute the following DROP INDEX statement:

```
DROP INDEX namex
```

2 Enter and execute the following line to redefine *namex* so that it includes the UNIQUE keyword:

```
CREATE UNIQUE INDEX namex ON Employee (last_name, first_name)
```

3 Once again, double-click **Employee**, and in the **Tutorial - Properties for: EMPLOYEE** dialog, click Indexes ( ).



Name	Sort	Unique	Column	Pos	References	Active
NAMEX	Asc	Yes	LAST_NAME	0		Yes
			FIRST_NAME	1		Yes
RDB$FOREIGN7	Asc	No	DEPT_NO	0	RDB$PRIMA...	Yes
RDB$FOREIGN8	Asc	No	JOB_CODE	0	RDB$PRIMA...	Yes
			JOB_GRADE	1	RDB$PRIMA...	Yes
			JOB_COUNT...	2	RDB$PRIMA...	Yes
RDB$PRIMARY6	Asc	Yes	EMP_NO	0		Yes

C:\Embarcadero\InterBase\database\tutorial.gdb Tables

Time to back up: If you have successfully altered the *Department* table definition, created the *phone_list* view, created the three indexes, and altered the *namex* index, this is a good time to back up your database to **Tutorial3.gbk**.

ADVANCE TO PART III

Click here to advance to the next section: Part III: Populating the Database.

PART III: POPULATING THE DATABASE

In Part II you created the structure of your database: domains, tables, a view, and three indexes. This chapter walks you through exercises where you will use the INSERT statement to *populate* (add data to) the database that you created in previous steps. Then you can use UPDATE and DELETE statements to manipulate the data.

INSERTING DATA

The INSERT statement is the mechanism by which you store one or more rows of data in an existing table. In its simplest form, the syntax is:

```
INSERT INTO table_name [(columns)] VALUES (values)
```

If you don't specify column names, InterBase inserts the supplied values into columns in the order in which they were defined, and there must be as many values as there are columns in the table. When you specify columns, you supply the values in the order you name the columns. Columns not specified are given default values or NULL values, depending on the column definitions.

The values supplied can be constants or can be calculated. In embedded SQL, they can also be variables.

An important variation of this syntax is one that allows you to add rows to a table by selecting rows from another table. The two tables must have columns occurring in the same order for this to work. The syntax for this form is:

> INSERT INTO *table_name* (*columns*) SELECT *columns* FROM *table_name* WHERE *conditions*

See the Language Reference for a full description of INSERT.

Inserting Data Using Column Values

1 Enter and execute the following code to add a row to the *Country* table:

```
INSERT INTO Country(country, currency) VALUES ('USA', 'Dollar')
```

Reminder: Anything you type inside the quotation marks is case sensitive.

2 Enter and execute the following line to add a row to the *Department* table:

```
INSERT INTO Department

    (dept_no, department, head_dept, budget, location, phone_no)

VALUES ('000', 'Corporate Headquarters', NULL, 1000000,
'Monterey',
            '(408) 555-1234')
```

Notice that strings are all enclosed in single quotes, while numeric values are not. The department number and default phone number, for example, are strings, not numeric values.

3 The next row of data for the *Department* table is similar to the previous one. To simplify entry, click the Previous Script button. This displays the previous query in the SQL Statement Area.

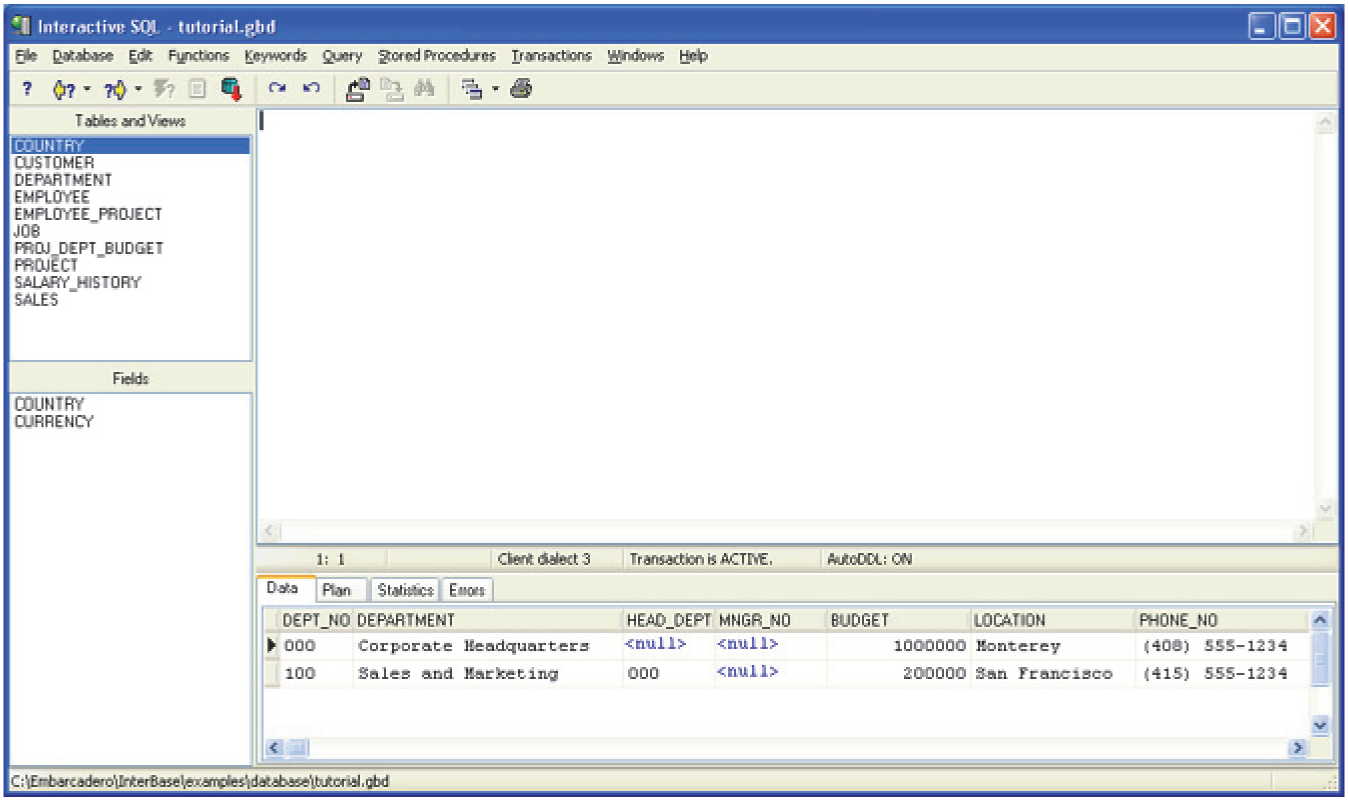
4 Now substitute into the previous query so that it reads as follows and execute the statement.

```
         INSERT INTO Department
                (dept_no, department, head_dept, budget, location, phone_no)
           VALUES ('100', 'Sales and Marketing', '000', 200000, 'San
      Francisco',
                   '(415) 555-1234')
```

Notice that the new value for head_dept is a string, not a numeric value.

5 Check the accuracy of your insertions by entering and executing each of the following statements in turn. Examine the output to make sure it matches the instructions above.

```
SELECT * from Country

SELECT * from Department
```



DEPT_NO	DEPARTMENT	HEAD_DEPT	MNGR_NO	BUDGET	LOCATION	PHONE_NO
000	Corporate Headquarters	<null>	<null>	1000000	Monterey	(408) 555-1234
100	Sales and Marketing	000	<null>	200000	San Francisco	(415) 555-1234

In Part IV of this Quick Start, you will learn more about the important SELECT statement.

▸Reading in the remaining data

1. To read the remaining data into the *Country*, *Job*, *Department*, and *Employee* tables, open **Inserts.sql** in a text editor, make sure that the CONNECT statement has the correct information, and load it into the database using **Execute**.

2. To confirm that data has been entered into each table enter and execute each following statement in turn.

   ```
   SELECT * FROM Country
   ```

   There should be 14 entries in the *Country* table. If this one is correct, the others probably are, too. Now run three more SELECT statements. Remember, you must execute each one before proceeding to the next.

   ```
   SELECT * FROM Job

   SELECT * FROM Employee

   SELECT * FROM Department
   ```

   Once again it is time to back up If you have successfully entered three INSERTs and run the **Inserts.sql** script, this is a good time to back up your database to **Tutorial4.gbk**.

UPDATING DATA

You use UPDATE statements to change values for one or more rows of data in existing tables.

USING UPDATE

A simple update has the following syntax:

```
UPDATE table

SET column = value

WHERE condition
```

The UPDATE statement changes values for columns specified in the SET clause; columns not listed in the SET clause are not changed. To update more than one column, list each column assignment in the SET clause, separated by a comma. The WHERE clause determines which rows to update. If there is no WHERE clause, all rows are updated.

For example, the following statement would increase the salary of salespeople by $2,000, by updating the *salary* column of the *Employee* table for rows where the value in the *job_code* column is “sales.” (Don’t do this yet.)

UPDATE Employee

```
SET salary = salary + 2000
```

```
WHERE job_code = 'Sales'
```

Executing and Committing

- For the rest of this tutorial, execute each statement after entering it. You will no longer be explicitly instructed to do so.
- In addition, execute a COMMIT statement after executing a DML statement (INSERT, DELETE, UPDATE, and SELECT). DDL statements—CREATE, ALTER, and DROP—don't need manual commits because you have enabled Auto Commit DDL in IBConsole's Interactive SQL Preferences.

Updating Data in the Employee Table

To make a more specific update, make the WHERE clause more restrictive. Enter the following code to increase the salaries only of salespeople hired before January 1, 1992:

```
UPDATE Employee

SET salary = salary + 2000

WHERE job_code = 'Sales' AND hire_date < '01-JAN-1992'
```

A WHERE clause is not required for an update. If the previous statements did not include a WHERE clause, the update would increase the salary of all employees in the *Employee* table.

Running the Updates.sql Script

1. Open the **Updates.sql** file in a text editor and look it over. It contains UPDATE statements that set values for the *mngr_no* column in the *Department* table, it updates the *Customer* table by setting the status of two customers to "on hold" by entering an asterisk in the *on_hold* column. Close the file when you have finished examining it.
2. Choose **Query > Execute** and run the **Updates.sql** file.

Time to back up: If you have successfully run the **Updates.sql** script and performed the manual update, this is a good time to back up your database to **Tutorial5.gbk**.

Updating to a NULL Value

Sometimes data needs to be updated before all the new values are available. You can indicate unknown data by setting values to NULL. This works only if a column is *nullable*, meaning that it is not defined as NOT NULL.

Suppose that in the previous example, the department number of salespeople hired before 1992 is changing but the new number is not yet known. You would update salaries and department numbers as follows:

```
UPDATE Employee

SET salary = salary + 2000, dept_no = NULL
```

```
WHERE job_code = 'Sales'
AND hire_date < '01-Jan-1992'
```

Using a Subquery to Update

The search condition of a WHERE clause can be a subquery. Suppose you want to change the manager of the department that Katherine Young current manages. One way to do this is to first determine Katherine Young's department number (don't do this yet):

```
SELECT dept_no FROM Employee

WHERE full_name = 'Young, Katherine'
```

This query returns "623" as the department. Then, using 623 as the search condition in an UPDATE, you change the department with the following statement:

```
UPDATE Department

SET mngr_no = 107

WHERE dept_no = '623'
```

A more efficient way to perform the update is to combine the two previous statements using a subquery. A subquery is one in which a SELECT clause is used within the WHERE clause to determine which rows to update.

Updating Department Using a Subquery

1 So that you can see the results of this exercise, begin by entering the following query to show you the manager number of Katherine's department before you make the update:

```
SELECT mngr_no FROM department WHERE dept_no = '623'
```

This returns 15. (If you select *first_name* and *last_name* from the *Employee* table where *emp_no* equals 15, you will see that the manager of department 623 is Katherine herself.)

2 Enter and execute the following UPDATE statement with a subquery to simultaneously find out Katherine's department number and assign a new manager number to that department:

```
UPDATE Department

SET mngr_no = 107

WHERE dept_no = (SELECT dept_no FROM Employee

WHERE full_name = 'Young, Katherine')
```

The rows returned by the SELECT statement within the parentheses are the rows that the UPDATE statement acts on.

3 Now run the query in step 1 again to see the change. The manager of department 623 is manager number 107, rather than 15.

4 This isn’t a change we want to keep, so enter and execute the following statement to reinstate Katherine Young as manager of department 623:

```
UPDATE Department SET mngr_no = 15 WHERE dept_no = '623'
```

Deleting Data

To remove one or more rows of data from a table, use the DELETE statement. A simple DELETE has the following syntax:

```
DELETE FROM table

WHERE condition
```

As with UPDATE, the WHERE clause specifies a search condition that determines the rows to delete. Search conditions can be combined or can be formed using a subquery.

Important The DELETE statement does not require a WHERE clause. However, if you do not include a WHERE clause, you delete *all* the rows of a table.

Deleting a Row from Sales (and put it back)

In this exercise, you first look to see what orders are older than a certain date. Then you delete those sales from the *Sales* table and check to see that they are gone.

1 Enter the following SELECT statement to see what sales were ordered prior to 1992:

```
SELECT * FROM Sales WHERE order_date < '31-DEC-1991'
```

There should be only one order returned. Notice that the SELECT statement requires that you specify columns. You can also use “*” to specify all columns.

2 Commit your data to this point by selecting **Transaction > Commit** (or F9).

3 Enter the following DELETE statement. To make it easier, you can display the previous SELECT statement and substitute DELETE for “SELECT *”. You can use the Previous Statement ⇦? button to display previous statements:

```
DELETE FROM Sales
   WHERE order_date < '31-DEC-1991'
```

Notice that the DELETE statement does not take any column specification. That’s because it deletes all columns for the rows you have specified.

4 Now repeat your original SELECT query. There should be no rows returned.

5 However, you just realized that you didn’t want to delete that data after all. Fortunately, you committed previous work before executing this statement, so choose **Transaction > Rollback** and click OK at the prompt. This “undoes” all statements that were executed since the last Commit.

6 Perform the SELECT again to see that the deleted row is back.

Time to back up: Now that you have created your database and its tables and finished inserting and updating data, this is a good time to back up your database to Tutorial6.gbk.

Deleting more Precisely

You can restrict deletions further by combining search conditions. For example, enter the following statement to delete records of everyone in the sales department hired before January 1, 1992:

```
DELETE FROM Employee
   WHERE job_code = 'Sales'
      AND hire_date < '01-Jan-1992'
```

You can try entering this statement, but you'll get an error because there's a foreign key column in the *Employee_project* table that references the *Employee* table. If you were to delete these rows, some values in the *Employee_project* table would no longer have matching values in the *Employee* table, violating the foreign key constraint that says any employee who has a project must also have an entry in the *Employee* table.

In addition, you can use subqueries to delete data, just as you use them to update. The following statement would delete all rows from the *Employee* table where the employees were in the same department as Katherine Young.

```
DELETE FROM Employee
      WHERE dept_no = (SELECT dept_no FROM Employee
         WHERE full_name = 'Young, Katherine')
```

Again, you cannot actually execute this statement because it would violate foreign key constraints on other tables.

Advance to Part IV

Click here to advance to the next section: Part IV: Retrieving Data

PART IV: RETRIEVING DATA

The SELECT statement lies at the heart of SQL because it is how you retrieve the information you have stored. There is no use in creating and populating data structures if you cannot get the data out again in usable form. You've seen some simple forms of the SELECT statement in earlier exercises. In this part of the tutorial you get further practice with the SELECT statement.

OVERVIEW OF SELECT

Part III presented the simplest form of the SELECT statement. The full syntax is much more complex. Take a minute to look at the entry for SELECT in the Language Reference Guide. Much of SELECT's power comes from its rich syntax.

In this chapter, you learn a core version of the SELECT syntax:

```
SELECT [DISTINCT] columns

FROM tables

WHERE <search_conditions>

[GROUP BY column [HAVING <search_condition>]]

[ORDER BY <order_list>]
```

The SELECT syntax above has seven main keywords. A keyword and its associated information is called a *clause*. The clauses above are:

Clause	**Description**
SELECT *columns*	Lists columns to retrieve
DISTINCT	Optional keyword that eliminates duplicate rows
FROM *tables*	Identifies the tables to search for values
WHERE <*search_conditions*>	Specifies the search conditions used to limit retrieved rows to a subset of all available rows
GROUP BY *column*	Groups rows retrieved according the value of the specified column
HAVING <*search_conditions*>	Specifies search condition to use with GROUP BY clause
ORDER BY <*order_list*>	Orders the output of a SELECT statement by the specified columns

Table 1: Seven Important SELECT Clauses

You have already used SELECT statements to retrieve data from single tables. However, SELECT can also retrieve data from multiple tables, by listing the table names in the FROM clause, separated by commas.

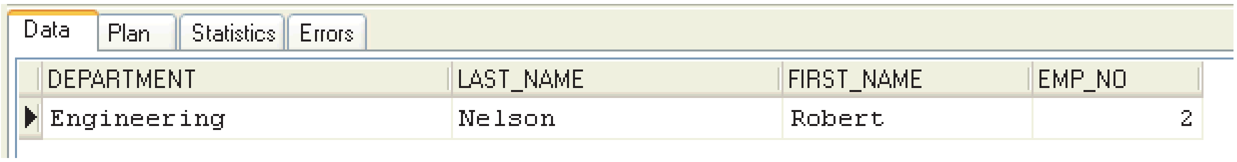
Retrieving Data from Two Tables at Once

In this example, you want to know the name and employee number of the person who manages the Engineering department. The Department table contains the manager number (*mngr_no*) for each department. That manager number matches an employee number (*emp_no*) in the Employee table, which has a first and last name in the record with the employee number. You link the corresponding records of the two tables by using the WHERE clause to specify the foreign key of one (*mngr_no*) as equal to the primary key (*emp_no*) of the other. Since the primary key is guaranteed to be unique, you are specifying a unique row in the employee table. Neither key has to be part of the SELECT clause. In this example, the referenced primary key is part of the SELECT clause but the foreign key is not.

To get the information described above, execute the following SQL statement in IBConsole:

```
SELECT department, last_name, first_name, emp_no

FROM Department, Employee

WHERE department = 'Engineering' AND mngr_no = emp_no
```

This statement retrieves the following information:



DEPARTMENT	LAST_NAME	FIRST_NAME	EMP_NO
Engineering	Nelson	Robert	2

REMOVING DUPLICATE ROWS WITH DISTINCT

Columns often contain duplicate entries (assuming that they do not have PRIMARY KEY or UNIQUE constraints on them). Sometimes you want to see only one instance of each value in a column. The DISTINCT keyword gives you exactly that.

Selecting One of Each

1 Suppose you want to retrieve a list of all the valid job codes in the TUTORIAL database. Begin by entering this query:

```
SELECT job_code FROM Job
```

As you can see, the results of this query are rather long, and some job codes are repeated a number of times. What you really want is a list of job codes where each value returned is distinct from the others. To eliminate duplicate values, use the DISTINCT keyword.

2 Re-enter the previous query with the DISTINCT keyword:

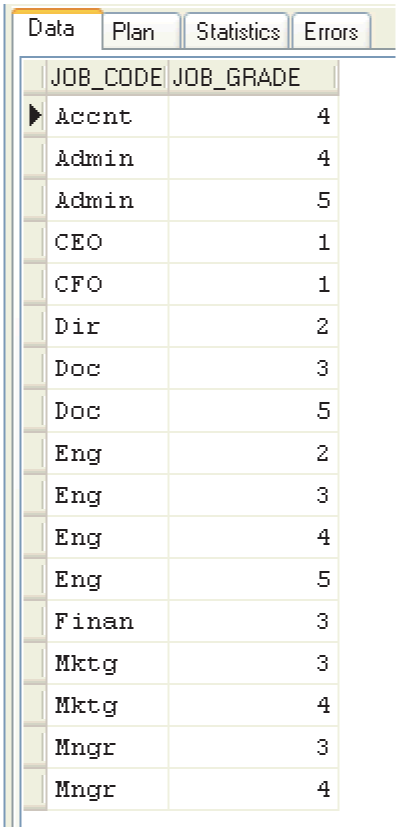
```
SELECT DISTINCT job_code FROM Job
```

This produces the desired results: each job code is listed only once in the results.

3 What happens if you specify another column when using DISTINCT? Enter the following SELECT statement:

```
SELECT DISTINCT job_code, job_grade FROM Job
```

This query returns:



JOB_CODE	JOB_GRADE
Accnt	4
Admin	4
Admin	5
CEO	1
CFO	1
Dir	2
Doc	3
Doc	5
Eng	2
Eng	3
Eng	4
Eng	5
Finan	3
Mktg	3
Mktg	4
Mngr	3
Mngr	4

DISTINCT applies to all columns listed in a SELECT statement. In this case, duplicate job codes are retrieved. However, DISTINCT treats the job code and job grade together, so the *combination* of values is distinct.

USING THE WHERE CLAUSE

The WHERE clause follows the SELECT and FROM clauses. It must precede the ORDER BY clause if one is used. The WHERE clause tests data to see whether it meets certain conditions, and the SELECT statement returns only the rows that meet the WHERE condition. The WHERE clause lies at the heart of database usage, because it is the point at which you state exactly what you want. It seems complex at first glance, but the complexity exists to allow you to be precise in your requests for data.

Using WHERE

1 Enter the following statement to return only rows for which "Green" is the value in the *last_name* column.

```
SELECT last_name, first_name, phone_ext

FROM Employee

WHERE last_name = 'Green'
```

The query should return one row:

```
Green                T.J.            218
```

2 Now display this statement again by clicking ⇦? and change the equal sign to a greater than sign. This retrieves rows for which the last name is alphabetically greater than (after) "Green." There should be 29 rows.

Something extra: To make the results more readable, execute the last query once again, but add an ORDER BY clause. This is just a preview: the ORDER BY clause is discussed starting on page 20.

```
SELECT last_name, first_name, phone_ext

FROM Employee

WHERE last_name > 'Green'

ORDER BY last_name
```

Search Conditions

The text following the WHERE keyword is called a *search condition*, because a SELECT statement searches for rows that meet the condition. Search conditions consist of a column name (such as "last_name"), an operator (such as "="), and a value (such as "Green"). Thus, WHERE clauses have the following general form:

```
WHERE column_name operator value
```

In general, *column_name* is the column name in the table being queried, *operator* is a comparison operator (Table 2), and *value* is a value or a range of values compared against the column. Table 3 describes the kinds of values you can specify.

Comparison Operators

Search conditions use the following operators. Note that for two-character operators, there is no space between the operators.

Operator	Description
Comparison operators	Used to compare data in a column to a value in the search condition. Examples include <, >, <=, >=, =, !=, and <>. Other operators include BETWEEN, CONTAINING, IN, IS NULL, LIKE, and STARTING WITH.
Arithmetic operators	Used to calculate and evaluate search condition values. The operators are +, –, *, and /.
Logical operators	Used to combine search conditions or to negate a condition. The keywords are NOT, AND, and OR.

Table 2: Search Condition Operators

Search Condition Values

The values in a search condition can be literal, or they can be calculated (derived). In addition, the value can be the return value of a subquery. A subquery is a nested SELECT statement.

Values that are text literals must be placed in quotes. The approaching standard will require single quotes. Numeric literals must *not* be quoted.

Important String comparisons are case sensitive.

Types of Values	Description
Literal values	Numbers and text strings whose value you want to test literally; for example, the number 1138 or the string 'Smith'
Derived values	Functions and arithmetic expressions; for example salary * 2 or last_name || first_name
Subqueries	A nested SELECT statement that returns one or more values. The returned values are used in testing the search condition.

Table 3: Types of values used in search conditions

When a row is compared to a search condition, one of three values is returned:

- *True*: A row meets the conditions specified in the WHERE clause.
- *False*: A row does not meet the conditions specified in the WHERE clause.
- *Unknown*: A field in the WHERE clause contains an NULL state that could not be evaluated.

Find the Deadbeats

Execute the following SELECT statement into the SQL Statement Area of IBConsole to query the *Sales* table for all the customers who ordered before January 1, 1994, received their shipment, and still haven’t paid. Notice that there are three search conditions in the WHERE clause, which are joined together with the AND operator.

```
SELECT * from Sales

WHERE order_date < '1-JAN-1994' AND order_status = 'shipped' AND
paid = 'n'
```

You should get two rows, one for PO number V93F3088 and one for PO number V93N5822.

Negation

You can negate any expression with the negation operators !, ^, and ~. These operators are all synonyms for NOT.

Suppose you just want to find what customers are not in the United States. Execute the following SELECT statement:

```
SELECT customer, country FROM Customer

WHERE NOT country = 'USA'
```

You should get a list of ten customers.

There are other ways to achieve exactly this result. To prove to yourself that these all produce the same results as the previous query, execute each of the following forms of it:

```
SELECT customer, country FROM Customer

WHERE country != 'USA'

SELECT customer, country FROM Customer

WHERE country ~= 'USA'

SELECT customer, country FROM Customer

WHERE country ^= 'USA'
```

Pattern Matching

Besides comparing values, search conditions can also test character strings for a particular pattern. If data is found that matches a given pattern, the row is retrieved.

Wildcards Use a percent sign (%) to match zero or more characters. Use an underscore (_) to match a single character.

Table 4 gives examples of some common patterns. Only CONTAINING is not case sensitive.

WHERE	Matches
last_name LIKE '%q%'	Last names containing at least one "q"
last_name STARTING WITH 'Sm'	Last names beginning with the letters "Sm."
last_name CONTAINING 'q'	Last name contains at least one "q," either uppercase or lowercase.
last_name BETWEEN 'M' AND 'T'	Last name beginning with letters M through S

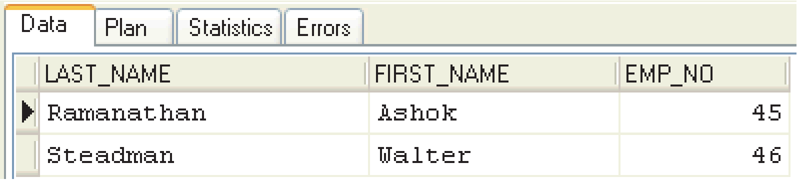
Table 4: Pattern Matching Examples

Find what's LIKE a Value

1 LIKE is case sensitive and takes wildcards. Execute this statement to find all employees whose last name ends in "an":

```
SELECT last_name, first_name, emp_no FROM Employee

WHERE last_name LIKE '%an'
```

The result set should look like this:



LAST_NAME	FIRST_NAME	EMP_NO
Ramanathan	Ashok	45
Steadman	Walter	46

2 Now enter the following SELECT statement to find employees whose last names begin with "M", have exactly two more characters and then a "D", followed by anything else in the remainder of the name:

```
SELECT last_name, first_name, emp_no FROM Employee

WHERE last_name LIKE 'M__D%'
```

This returns

MacDonald	Mary S.	85

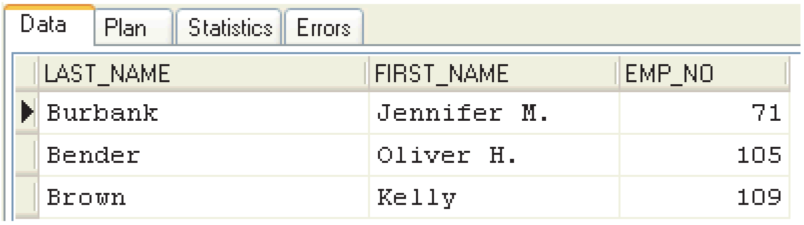
but it would ignore names like McDonald.

3 The "%" matches zero or more characters. The following query returns rows for last names Burbank, Bender, and Brown.

```
SELECT last_name, first_name, emp_no FROM Employee
```

```
          WHERE last_name LIKE 'B%r%'
```

This returns:



LAST_NAME	FIRST_NAME	EMP_NO
Burbank	Jennifer M.	71
Bender	Oliver H.	105
Brown	Kelly	109

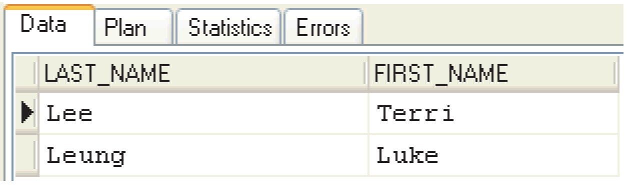
Find Things STARTING WITH

The STARTING WITH operator tests whether a value starts with a particular character or sequence of characters. STARTING WITH is case sensitive, but does not support wildcard characters.

1 Execute the following statement to retrieve two employee last names that start with "Le":

```
SELECT last_name, first_name FROM Employee

WHERE last_name STARTING WITH 'Le'
```

The result set is:

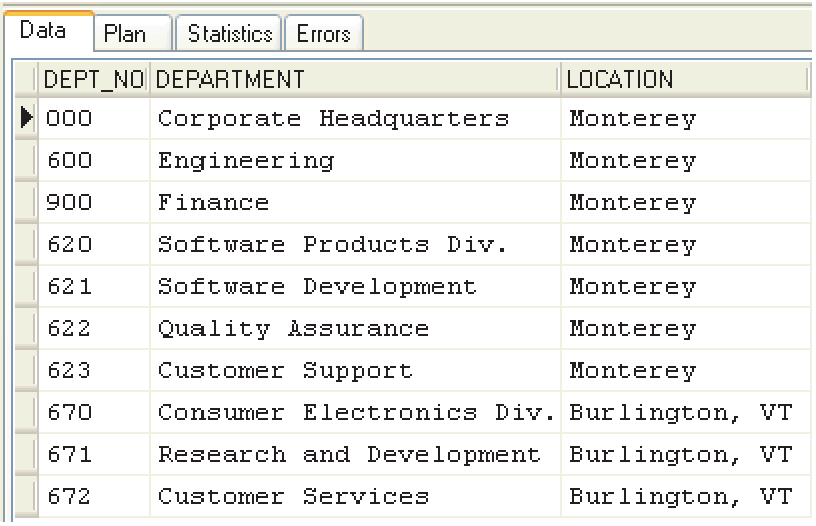


LAST_NAME	FIRST_NAME
Lee	Terri
Leung	Luke

2 To negate the STARTING WITH operator, precede it with the logical operator NOT. (*Note*: That's a "one" in the quotes at the end of the statement.)

```
SELECT dept_no, department, location from department
WHERE dept_no NOT starting with '1'
```

This query should return the following 10 rows:



DEPT_NO	DEPARTMENT	LOCATION
000	Corporate Headquarters	Monterey
600	Engineering	Monterey
900	Finance	Monterey
620	Software Products Div.	Monterey
621	Software Development	Monterey
622	Quality Assurance	Monterey
623	Customer Support	Monterey
670	Consumer Electronics Div.	Burlington, VT
671	Research and Development	Burlington, VT
672	Customer Services	Burlington, VT

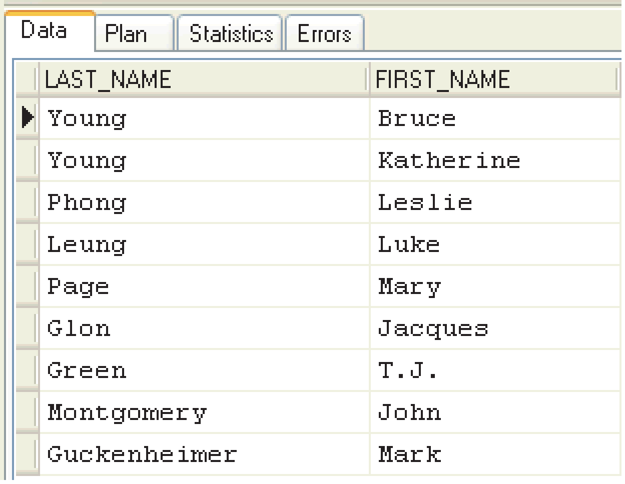
Find Something CONTAINING a Value

The CONTAINING operator is similar to STARTING WITH, except it matches strings containing the specified string *anywhere* within the string. CONTAINING is *not* case sensitive and does not support wildcards.

1 Execute the following statement to find last names that have a “g” or “G” anywhere in them.

```
SELECT last_name, first_name FROM Employee

WHERE last_name CONTAINING 'G'
```

You should get the following result set:



LAST_NAME	FIRST_NAME
Young	Bruce
Young	Katherine
Phong	Leslie
Leung	Luke
Page	Mary
Glon	Jacques
Green	T.J.
Montgomery	John
Guckenheimer	Mark

Now execute the same query, except substitute a lower-case “g.” You should get exactly the same result set.

```
SELECT last_name, first_name FROM Employee
```

```
WHERE last_name CONTAINING 'g'
```

Testing for an Unknown Value

Another type of comparison tests for the absence or presence of a value. Use the IS NULL operator to test whether a value is unknown. To test for the *presence* of any value, use IS NOT NULL.

Testing for NULL

1 Execute the following query to retrieve the names of employees who do not have phone extensions:

```
SELECT last_name, first_name, phone_ext FROM Employee

WHERE phone_ext IS NULL
```

The query should return rows for last names Sutherland, Glon, and Osborne.

2 Now execute the statement using IS NOT NULL to retrieve the names of employees who *do* have phone extensions:

```
SELECT last_name, first_name, phone_ext FROM Employee

WHERE phone_ext IS NOT NULL
```

There should be 39 rows in the result set.

Comparing Against a Range or List of Values

The previous sections present comparison operators that test against a single value. The BETWEEN and IN operators test against multiple values.

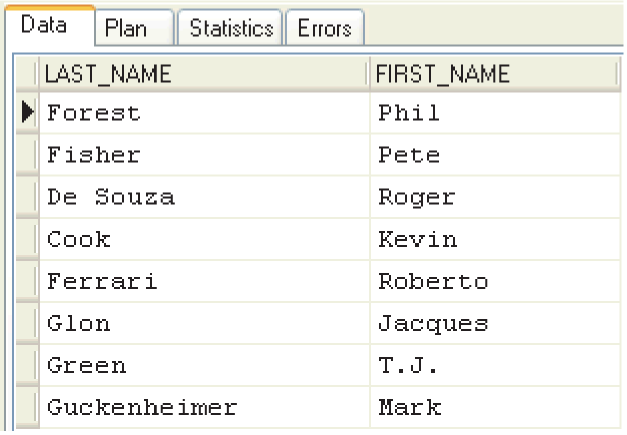
BETWEEN tests whether a value falls within a range. The BETWEEN operator is case-sensitive and does not require wildcards.

Finding Something BETWEEN Values

1 Execute the following query to find all the last names that start with letters between C and H. Notice that the query does not include names that begin with the final value (“H”). This is because BETWEEN finds values that are less than or equal to the terminating value. A name that begins with the letter but includes other letters is greater than H. If there were someone whose last name was just “H”, the query would return it.

```
SELECT last_name, first_name FROM Employee

WHERE last_name BETWEEN 'C' AND 'H'
```

The result set is:



LAST_NAME	FIRST_NAME
Forest	Phil
Fisher	Pete
De Souza	Roger
Cook	Kevin
Ferrari	Roberto
Glon	Jacques
Green	T.J.
Guckenheimer	Mark

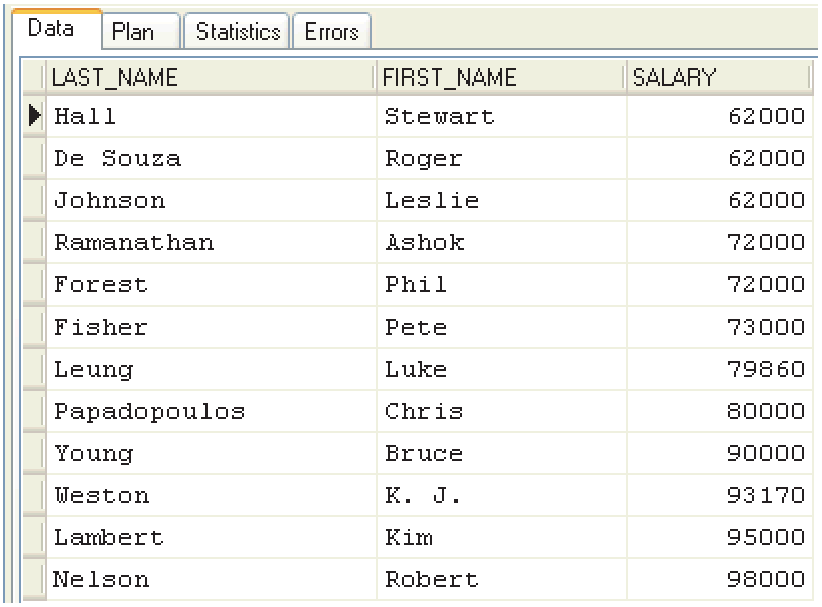
2 To demonstrate that BETWEEN is case sensitive, repeat the previous query using lower-case letters. There are no names returned.

```
SELECT last_name, first_name FROM Employee

WHERE last_name BETWEEN 'c' AND 'h'
```

3 Execute the following query to retrieve names of employees whose salaries are between $62,000 and $98,000, inclusive:

```
SELECT last_name, first_name, salary FROM Employee

WHERE salary BETWEEN 62000 AND 98000
ORDER BY salary
```

The result set should return 13 rows, with salaries that include both the low figure and the high figure in the range. (See page 20 for a discussion of the ORDER BY clause.)



LAST_NAME	FIRST_NAME	SALARY
Hall	Stewart	62000
De Souza	Roger	62000
Johnson	Leslie	62000
Ramanathan	Ashok	72000
Forest	Phil	72000
Fisher	Pete	73000
Leung	Luke	79860
Papadopoulos	Chris	80000
Young	Bruce	90000
Weston	K. J.	93170
Lambert	Kim	95000
Nelson	Robert	98000

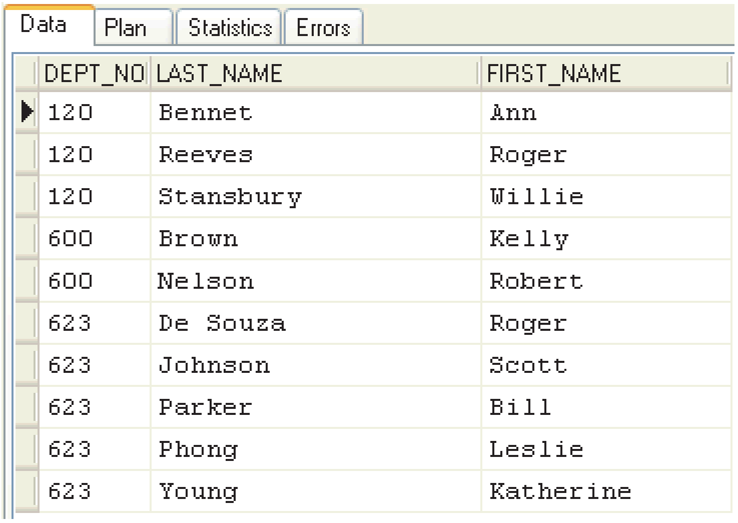
Finding What's IN

The IN operator searches for values matching one of the values in a list. The values in the list must be separated by commas, and the list must be enclosed in parentheses. Use NOT IN to search for values that do not occur in a set

Execute the following query to retrieve the names of all employees in departments 120, 600, and 623:

```
SELECT dept_no, last_name, first_name FROM Employee

WHERE dept_no IN (120, 600, 623)

ORDER BY dept_no, last_name
```

The returns the following result set:



DEPT_NO	LAST_NAME	FIRST_NAME
120	Bennet	Ann
120	Reeves	Roger
120	Stansbury	Willie
600	Brown	Kelly
600	Nelson	Robert
623	De Souza	Roger
623	Johnson	Scott
623	Parker	Bill
623	Phong	Leslie
623	Young	Katherine

LOGICAL OPERATORS

You can specify multiple search conditions in a WHERE clause by combining them with the logical operators AND or OR.

Finding Rows that Match Multiple Conditions

When AND appears between search conditions, both conditions must be true for a row to be retrieved. For example, execute this query to find employees in a particular department who were hired after January 1, 1992:

```
SELECT dept_no, last_name, first_name, hire_date

FROM Employee

WHERE dept_no = 623 AND hire_date > '01-Jan-1992'
```

It should return two rows, one each for employees Parker and Johnson.

Finding Rows that Match at Least One Condition

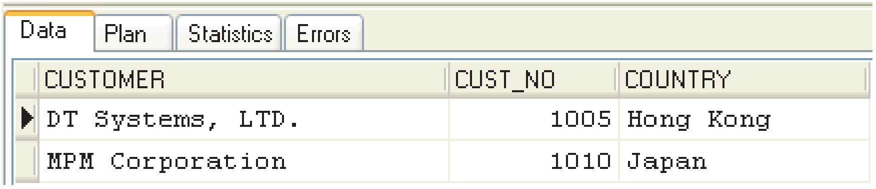
Use OR between search conditions where you want to retrieve rows that match at least one of the conditions.

1 Click the Previous Query button to display your last query. Change AND to OR and execute the new query. Notice that the results are dramatically different; the query returns 25 rows.

2 As a more likely example of the OR operator, execute the following query to find customers who are in either Japan or Hong Kong:

```
SELECT customer, cust_no, country FROM Customer

WHERE country = 'Japan' OR country = 'Hong Kong'
```

The result set should look like this:



CUSTOMER	CUST_NO	COUNTRY
DT Systems, LTD.	1005	Hong Kong
MPM Corporation	1010	Japan

Controlling the Order of Evaluation

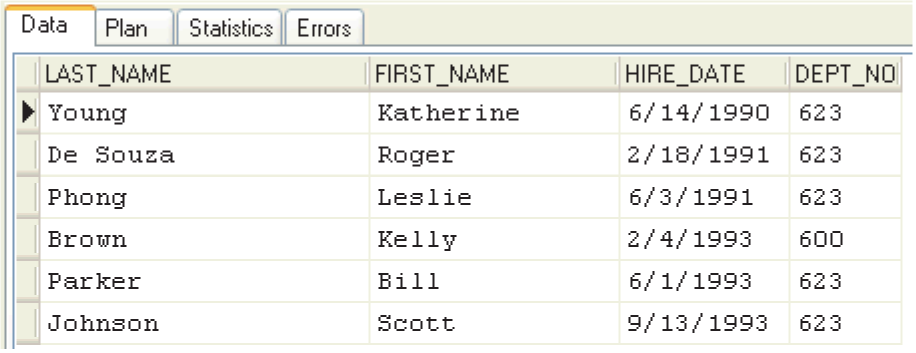
When entering compound search conditions, you must be aware of the order of evaluation of the conditions. Suppose you want to retrieve employees in department 623 or department 600 who have a hire date later than January 1, 1992.

Trying a Compound Condition

Try executing this query:

```
SELECT last_name, first_name, hire_date, dept_no

FROM Employee

WHERE dept_no = 623 OR dept_no = 600 AND hire_date > '01-JAN-1992'
```

As you can see, the results include employees hired earlier than you want:



LAST_NAME	FIRST_NAME	HIRE_DATE	DEPT_NO
Young	Katherine	6/14/1990	623
De Souza	Roger	2/18/1991	623
Phong	Leslie	6/3/1991	623
Brown	Kelly	2/4/1993	600
Parker	Bill	6/1/1993	623
Johnson	Scott	9/13/1993	623

This query produces unexpected results because AND has *higher precedence* than OR. This means that the expressions on either side of AND are tested before those associated with OR. In the example as written, the search conditions are interpreted as follows:

```
( WHERE dept_no = 623 )

OR

( WHERE dept_no = 600 AND hire_date > '01-JAN-1992' )
```

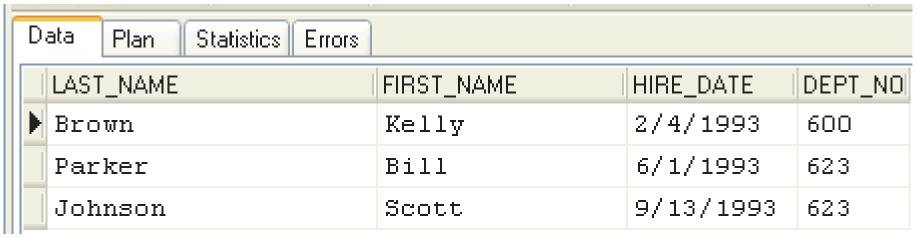
The restriction on the hire date applies only to the second department. Employees in department 623 are listed regardless of hire date.

Using a Compound Condition Successfully

Use parentheses to override normal precedence. In the exercise below, place parentheses around the two departments so they are tested against the AND operator as a unit. Redisplay your last query and add parentheses, so that your query is interpreted correctly:

```
SELECT last_name, first_name, hire_date, dept_no

FROM Employee

WHERE (dept_no = 623 OR dept_no = 600)

AND hire_date > '01-JAN-1992'
```

This displays the results you want:



LAST_NAME	FIRST_NAME	HIRE_DATE	DEPT_NO
Brown	Kelly	2/4/1993	600
Parker	Bill	6/1/1993	623
Johnson	Scott	9/13/1993	623

Order of precedence is not just an issue for AND and OR. All operators are defined with a precedence level that determines their order of interpretation. You can study precedence levels in detail by reading any number of books about SQL, but in general, the following rule of thumb is all you need to remember.

TIP: Always use parentheses to group operations in complex search conditions.

Using Subqueries

Subqueries are a special case of the WHERE clause, but they are an important tool and deserve a discussion of their own.

Recall that in a WHERE clause, you provide a column name, a comparative operator, and a value. WHERE tests the column contents against the value using the operator. You can use a SELECT statement in place of the value portion of a WHERE clause. This internal SELECT clause is the *subquery*. InterBase executes the SELECT subquery and uses its result set as the value for the WHERE clause.

Suppose, for example, that you want to retrieve a list of employees who work in the same country as a particular employee whose ID is 144. If you don't use a subquery, you would first need to find out what country this employee works in:

```
SELECT job_country FROM Employee

WHERE emp_no = 144
```

This query returns "USA." With this information, you would issue a second query to find a list of employees in the USA, the same country as employee number 144:

```
SELECT emp_no, last_name FROM Employee

WHERE job_country = 'USA'
```

Using a subquery permits you to perform both queries in a single statement.

Using a Subquery to Find a Single Item

You can obtain the same result by combining the two queries:

```
SELECT emp_no, last_name FROM Employee

WHERE job_country = (SELECT job_country FROM Employee WHERE emp_no
= 144)
```

In this case, the subquery retrieves a single value, "USA." The main query interprets "USA" as a value to be tested by the WHERE clause. The subquery must return a single value because the WHERE clause is testing for a single value ("="); otherwise the statement produces an error.

The result set for this query is a list of 33 employee numbers and last names. These are the employees who work in the same country as employee number 144.

MULTIPLE-RESULT SUBQUERIES

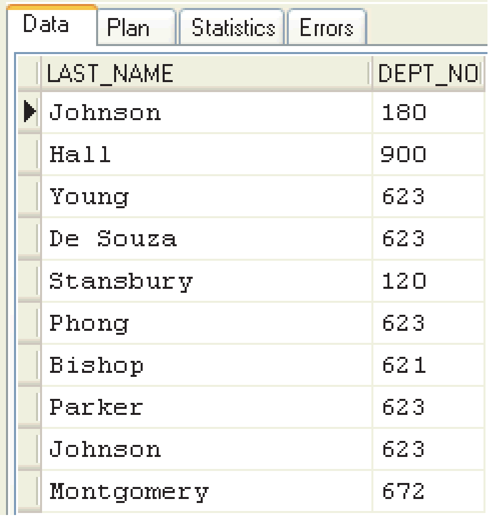
If a subquery returns more than one value, the WHERE clause that contains it must use an operator that tests against more than one value. IN is such an operator.

Using a Subquery to Find a Collection of Items

Execute the following example to retrieve the last name and department number of all employees whose salary is equal to that of someone in department 623. It uses a subquery that returns all the salaries of employees in department 623. The main query selects each employee in turn and checks to see if the associated salary is in the result set of the subquery.

```
SELECT last_name, dept_no FROM Employee

WHERE salary IN (SELECT salary FROM Employee WHERE dept_no = 623)
```

The result set should look like this:



LAST_NAME	DEPT_NO
Johnson	180
Hall	900
Young	623
De Souza	623
Stansbury	120
Phong	623
Bishop	621
Parker	623
Johnson	623
Montgomery	672

CONDITIONS FOR SUBQUERIES

The following table summarizes the operators that compare a value on the left of the operator to the results of a subquery on the right of the operator:

Operator	Purpose
ALL	Returns true if a comparison is true for all values returned by a subquery.
ANY or SOME	Returns true if a comparison is true for at least one value returned by a subquery.
EXISTS	Determines if at least one value exists in the resut set returned by a subquery.
SINGULAR	Determines if exactly one value exists in the result set returned by a subquery.

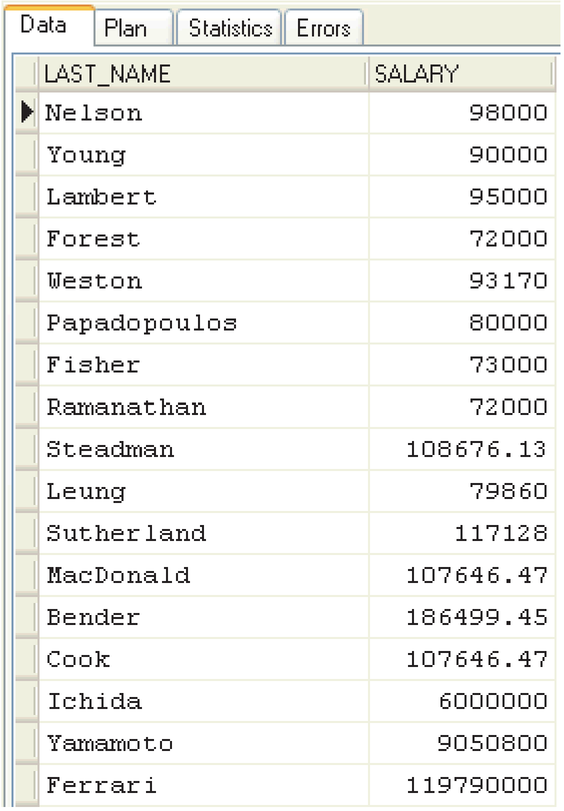
Table 5: InterBase Comparison Operators that Require Subqueries

Using ALL

The IN operator tests only against the *equality* of a list of values. What if you want to test some relationship other than equality? For example, suppose you want to find out who earns more than the people in department 623. Enter the following query:

```
SELECT last_name, salary FROM Employee

WHERE salary > ALL

(SELECT salary FROM Employee WHERE dept_no = 623)
```

The result set should look like this:



LAST_NAME	SALARY
Nelson	98000
Young	90000
Lambert	95000
Forest	72000
Weston	93170
Papadopoulos	80000
Fisher	73000
Ramanathan	72000
Steadman	108676.13
Leung	79860
Sutherland	117128
MacDonald	107646.47
Bender	186499.45
Cook	107646.47
Ichida	6000000
Yamamoto	9050800
Ferrari	119790000

This example uses the ALL operator. The statement tests against *all* values in the subquery and retrieves the row if the salary is greater. The manager of department 623 can use this output to see which company employees earn more than his or her employees.

Using ANY, EXISTS, and SINGULAR

Instead of testing against all values returned by a subquery, you can rewrite the example to test for at least one value:

```
SELECT last_name, salary FROM Employee

WHERE salary > ANY(SELECT salary FROM Employee WHERE dept_no =
623)
```

This statement retrieves 34 rows for which *salary* is greater than any of the values from the subquery. The ANY keyword has a synonym, SOME. The two are interchangeable.

Two other subquery operators are EXISTS and SINGULAR.

- For a given value, EXISTS tests whether *at least one* qualifying row meets the search condition specified in a subquery. EXISTS returns either true or false, even when handling NULL values.
- For a given value, SINGULAR tests whether *exactly one* qualifying row meets the search condition specified in a subquery.

Using Aggregate Functions

SQL provides aggregate functions that calculate a single value from a group of values. A group of values is all data in a particular column for a given set of rows, such as the job code listed in all rows of the JOB table. Aggregate functions may be used in a SELECT clause, or anywhere a value is used in a SELECT statement.

The following table lists the aggregate functions supported by InterBase:

Function	**What It Does**
AVG(*value*)	Returns the average value for a group of rows
COUNT(*value*)	Counts the number of rows that satisfy the WHERE clause
MIN(*value*)	Returns the minimum value in a group of rows
MAX(*value*)	Returns the maximum value in a group of rows
SUM(*value*)	Adds numeric values in a group of rows

Table 6: Aggregate Functions Supported by InterBase

Practicing with Aggregate Functions

1 Suppose you want to know how many different job codes are in the *Job* table. Enter the following statement:

```
SELECT COUNT(job_code) FROM Job
```

The result count is 31

However, this is not what you want, because the query included duplicate job codes in the count.

2 To count only the unique job codes, use the DISTINCT keyword as follows:

```
SELECT COUNT(DISTINCT job_code) FROM Job
```

This produces the correct result of 14.

3 Enter the following to retrieve the average budget of departments from the *Department* table:

```
SELECT AVG(budget) FROM Department
```

The result average is 648095.23

4 A single SELECT can retrieve multiple aggregate functions. Enter the following statement to retrieve the number of employees, the earliest hire date, and the total salary paid to all employees:

```
SELECT COUNT(emp_no), MIN(hire_date), SUM(salary)
```

```
FROM Employee
```

The result sum is: a count of 42, a minimum hire date of 12/28/1988, and a salary sum of 37860791.52.

NOTE: The value in the sum column may vary, depending on which exercises you have done and whether you have done some of them more than once.

Important If a value involved in an aggregate calculation is NULL or unknown, the function ignores the entire row to prevent wrong results. For example, when calculating an average over fifty rows, if ten rows contain a NULL value, then the average is taken over forty values, not fifty.

5 To see for yourself that aggregate functions ignore NULL rows, perform the following test: first, look at all the rows in the *Department* table:

```
SELECT dept_no, mngr_no FROM Department
```

Notice that there are 21 rows, but four of them have NULLs in the *mngr_no* column.

6 Now count the rows in *mngr_no*:

```
SELECT COUNT(mngr_no) FROM Department
```

The result is 17, not 21. COUNT did not count the NULL rows.

Grouping and Ordering Query Results

Rows are not stored in any particular order in a database. So when you execute a query, you may find that the results are not organized in any useful way. The ORDER BY clause lets you specify how the returned rows should be ordered. You can use the GROUP BY clause to group the results of aggregate functions.

Using ORDER BY to Arrange Rows

You can use the ORDER BY clause to organize the data that is returned from your queries. You can specify one or more columns by name or by ordinal number. The syntax of the ORDER BY clause is:

```
ORDER BY [col_name | int] [ASC[ENDING] | DESC[ENDING]] [, …]
```

Notice that you can specify more than one column. As an alternative to naming the columns, you can provide an integer that references the order in which you named the columns in the query. The second column that you named in the SELECT can be referenced as 2.

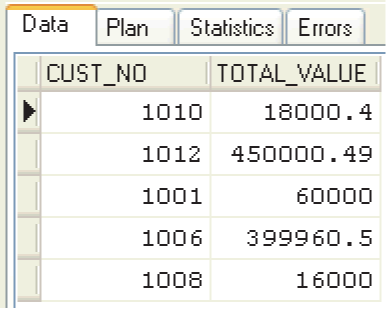
By default, InterBase uses ASCENDING order, so you only need to specify the order if you want it to be DESCENDING.

Practicing with ORDER BY

1 Execute the following query (you did this one earlier when you worked with conditions for subqueries).

```
SELECT cust_no, total_value FROM Sales

WHERE total_value > 10000
```

There's no particular order to the returned rows. The result set should look like this:



CUST_NO	TOTAL_VALUE
1010	18000.4
1012	450000.49
1001	60000
1006	399960.5
1008	16000

2 Execute the same query, but order the results by the *cust_no* column:

```
SELECT cust_no, total_value FROM Sales

WHERE total_value > 10000

ORDER BY cust_no
```

Notice that the result set now has the *cust_no* column in ascending order. Ascending order is the default.

3 Order the result set by the total value of the sales:

```
SELECT cust_no, total_value FROM Sales

WHERE total_value > 10000

ORDER BY total_value
```

4 Execute the query above, but order the result set by the descending order of the *total_value* column:

```
SELECT cust_no, total_value FROM Sales

WHERE total_value > 10000

ORDER BY total_value DESC
```

5 To see the effect of ordering by more than one column, execute the following query:

```
SELECT last_name, first_name, phone_ext FROM Employee
```

```
ORDER BY last_name DESC, first_name
```

Notice that there are 42 rows with the last names are in descending order, as requested, and the first names in ascending order, the default.

USING THE GROUP BY CLAUSE

You use the optional GROUP BY clause to organize data retrieved from aggregate functions. When you issue a query (a SELECT statement) that has both aggregate (AVG, COUNT, MIN, MAX, or SUM) and non-aggregate columns, you *must* use GROUP BY to group the result set by each of the nonaggregate columns. The three following rules apply:

- Each column from which you are doing a nonaggregate SELECT must appear in the GROUP BY clause
- The GROUP BY clause can reference only columns that appear in the SELECT clause
- Each SELECT clause in a query can have only one GROUP BY clause

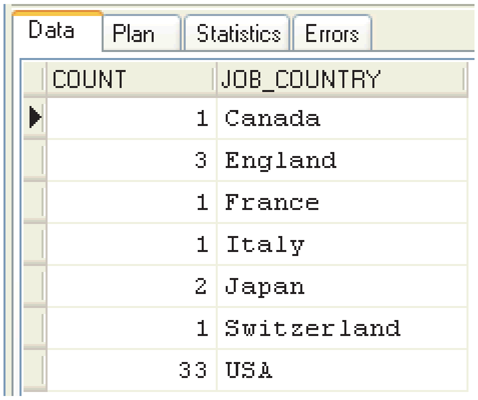
A group is defined as the subset of rows that match a distinct value in the columns of the GROUP BY clause.

Grouping the Result Set of Aggregate Functions

1 Execute the following query to find out how many employees there are in each country:

```
SELECT COUNT(emp_no), job_country FROM Employee

GROUP BY job_country
```

The result set should look like this:



COUNT	JOB_COUNTRY
1	Canada
3	England
1	France
1	Italy
2	Japan
1	Switzerland
33	USA

Using the HAVING Clause

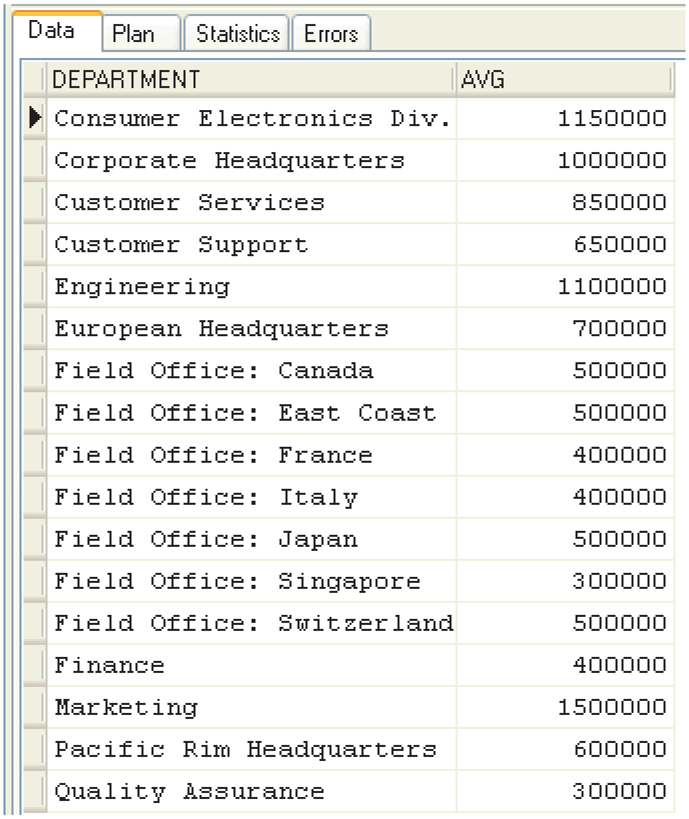
Just as a WHERE clause reduces the number of rows returned by a SELECT clause, the HAVING clause can be used to reduce the number of rows returned by a GROUP BY clause. Like the WHERE clause, a HAVING clause has a search condition. However, in a HAVING clause, the search condition typically corresponds to an aggregate function used in the SELECT clause.

Controlling your Query with GROUP BY and HAVING

Issue the following query to list the departments that have an average budget of over $60,000 and order the result set by department.

```
SELECT Department, AVG(budget) FROM department

GROUP BY Department

HAVING AVG(budget) >60000

ORDER BY Department
```

The result set should look like this:



DEPARTMENT	AVG
Consumer Electronics Div.	1150000
Corporate Headquarters	1000000
Customer Services	850000
Customer Support	650000
Engineering	1100000
European Headquarters	700000
Field Office: Canada	500000
Field Office: East Coast	500000
Field Office: France	400000
Field Office: Italy	400000
Field Office: Japan	500000
Field Office: Singapore	300000
Field Office: Switzerland	500000
Finance	400000
Marketing	1500000
Pacific Rim Headquarters	600000
Quality Assurance	300000

Ordering by an Aggregate Column

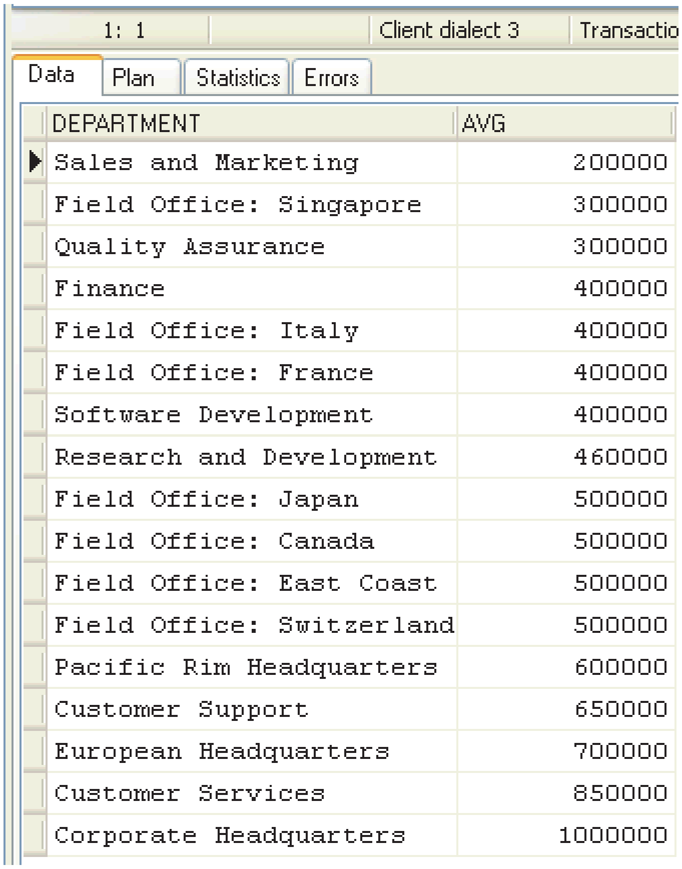
But what if you want to list the result set by the average budget in the previous query? In that query, it wouldn't work to say "ORDER BY budget" because the query generates a two-column result set in which the first column is named "department" but the second column, which is generated by the aggregate function, doesn't have a name. The ORDER BY clause is actually referencing the columns of the result set. In order to request that the result set be ordered by the results of an aggregate function, you must reference the ordinal column number. (Look back at the ORDER BY syntax on page 20 and notice that it begins "`ORDER BY [col_name | int]`". You must reference the column by its integer number.

Ordering Result Set by the Results of an Aggregate Function

Display your last query, but change the ORDER BY clause as follows, to order by the second column:

```
SELECT department, AVG(budget) FROM department

GROUP BY Department

HAVING AVG(budget) >60000

ORDER BY 2
```

Now the result set should have the second column in ascending order.



DEPARTMENT	AVG
Sales and Marketing	200000
Field Office: Singapore	300000
Quality Assurance	300000
Finance	400000
Field Office: Italy	400000
Field Office: France	400000
Software Development	400000
Research and Development	460000
Field Office: Japan	500000
Field Office: Canada	500000
Field Office: East Coast	500000
Field Office: Switzerland	500000
Pacific Rim Headquarters	600000
Customer Support	650000
European Headquarters	700000
Customer Services	850000
Corporate Headquarters	1000000

Joining Tables

Joins enable a SELECT statement to retrieve data from two or more tables in a database. The tables are listed in the FROM clause. The optional ON clause can reduce the number of rows returned, and the WHERE clause can further reduce the number of rows returned.

From the information in a SELECT that describes a join, InterBase builds a table that contains the results of the join operation, the *results table*, sometimes also called a *dynamic* or *virtual table*.

InterBase supports two types of joins: inner joins and outer joins.

Inner joins link rows in tables based on specified join conditions and return only those rows that match the join conditions. If a joined column contains a NULL value for a given row, that row is not included in the results table. Inner joins are the more common type because they restrict the data returned and show a clear relationship between two or more tables.

Outer joins link rows in tables based on specified join conditions but return rows whether they match the join conditions or not. Outer joins are useful for viewing joined rows in the context of rows that do not meet the join conditions.

Correlation Names

Once you begin to query multiple tables, it becomes important to identify unambiguously what table each column is in. The standard syntax for doing this is to state the table name followed by a period and the column name:

```
table_name.col_name
```

In complex queries, this can get very tedious, so InterBase permits you to state a shorter version of the table name in the FROM clause of a join. This short name is called a *correlation name* or an *alias*. You will see many examples of correlation names in the next few pages. The form is as follows:

```
SELECT a.col, b.col FROM table_1 a, table_2 b

ON a.some_col = b.some_col

WHERE a.conditional_col <condition>

…
```

Notice the FROM clause, where *table_1* is given the correlation name of *a* and *table_2* is named *b*. These abbreviated names are used even in the initial select list.

Important If you include a subquery in a join, you must assign new correlation names to any tables that appeared in the main query.

Inner Joins

There are three types of inner joins:

- *Equi-joins* link rows based on common values or equality relationships in the join columns.
- Joins that link rows based on comparisons other than equality in the join columns. There is not an officially recognized name for these joins, but for simplicity's sake they can be categorized as *comparative* joins, or *non-equi-joins*.
- *Reflexive* or *self-joins* compare values within a column of a single table.

To specify a SELECT statement as an inner join, list the tables to join in the FROM clause, and list the columns to compare in the ON clause. Use the WHERE clause to restrict which rows are retrieved. The simplified syntax is:

```
SELECT <columns> | *

FROM left_table [INNER] JOIN right_table
ON left_table.col = right_table.col

[WHERE <searchcondition>]
```

There are several things to note about this syntax:

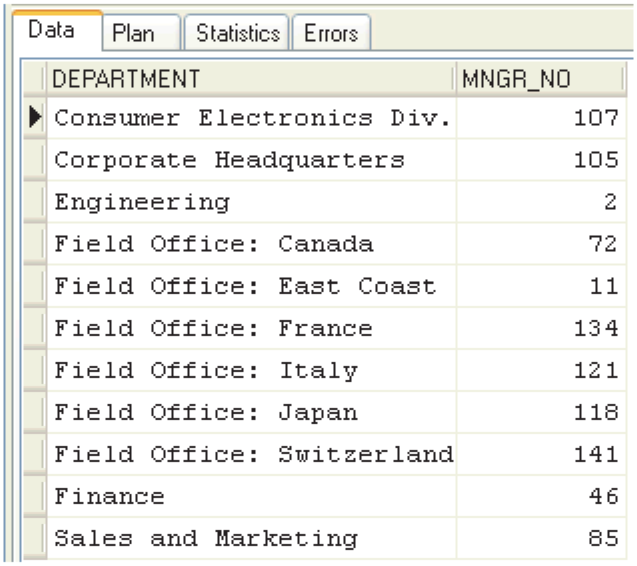
- The INNER keyword is optional because INNER is the default join type. If you want to perform an outer join, you must state the OUTER keyword explicitly.
- The FROM statement often specifies the correlation names
- FROM table1 t1 JOIN table2 t2
- The operator in the ON clause doesn't have to be equality. It can be any of the comparison operators, such as !=, >, >=, or <>.

Performing Inner Joins

1 Enter the following query to list all department managers and their departments where the manager earns more than 80,000. (This isn't stated as dollars because some of the employee salaries are in other currencies.)

```
SELECT D.department, D.mngr_no

FROM Department D INNER JOIN Employee E

ON D.mngr_no = E.emp_no

WHERE E.salary > 80000

ORDER BY D.department
```

The result set should look like this:



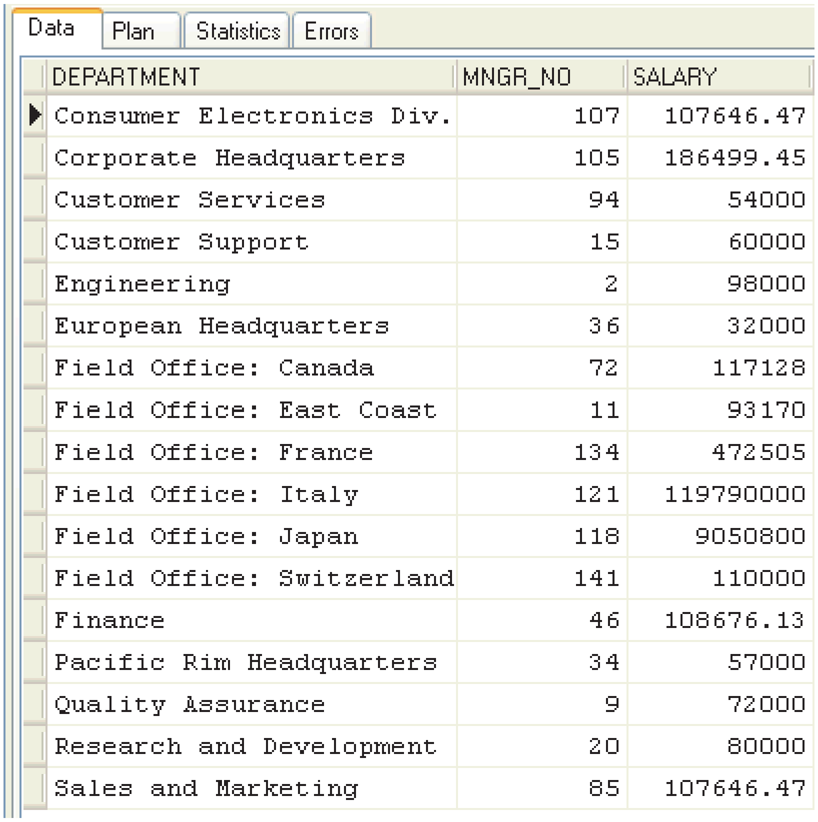
DEPARTMENT	MNGR_NO
Consumer Electronics Div.	107
Corporate Headquarters	105
Engineering	2
Field Office: Canada	72
Field Office: East Coast	11
Field Office: France	134
Field Office: Italy	121
Field Office: Japan	118
Field Office: Switzerland	141
Finance	46
Sales and Marketing	85

2 The next query uses a subquery to display all departments and department managers where the manager's salary is at least 20% of a department's total salary:

```
SELECT D.department, D.mngr_no, E.salary

FROM Department D JOIN Employee E

ON D.mngr_no = E.emp_no

WHERE E.salary*5 >= (SELECT SUM(S.salary) FROM Employee S
WHERE D.dept_no = S.dept_no)

ORDER BY D.department
```

The subquery sums the salaries for one department at a time and hands out the result to be compared to the manager's salary (multiplied by 5).

The result set should look like this:



DEPARTMENT	MNGR_NO	SALARY
Consumer Electronics Div.	107	107646.47
Corporate Headquarters	105	186499.45
Customer Services	94	54000
Customer Support	15	60000
Engineering	2	98000
European Headquarters	36	32000
Field Office: Canada	72	117128
Field Office: East Coast	11	93170
Field Office: France	134	472505
Field Office: Italy	121	119790000
Field Office: Japan	118	9050800
Field Office: Switzerland	141	110000
Finance	46	108676.13
Pacific Rim Headquarters	34	57000
Quality Assurance	9	72000
Research and Development	20	80000
Sales and Marketing	85	107646.47

NOTE: Joins are not limited to two tables. There is theoretically no limit to how many tables can be joined in one statement, although on the practical level of time and resources, 16 is usually considered the workable maximum.

Outer Joins

Outer joins produce a results table containing columns from every row in one table and a subset of rows from another table. Outer join syntax is very similar to that of inner joins.

```
SELECT col [, col …] | *

FROM left_table {LEFT | RIGHT | FULL} OUTER JOIN
right_table ON joincondition

[WHERE <searchcondition>]
```

The *joincondition* is of the form `left_table.col = right_table.col` where the equality operator can be replaced by any of the comparison operators.

With outer joins, you must specify the type of join to perform. There are three types:

- A *left outer join* retrieves all rows from the left table in a join, and retrieves any rows from the right table that match the search condition specified in the ON clause.

- A *right outer join* retrieves all rows from the right table in a join, and retrieves any rows from the left table that match the search condition specified in the ON clause.
- A *full outer join* retrieves all rows from both the left and right tables in a join regardless of the search condition specified in the ON clause.

Outer joins are useful for comparing a subset of data in the context of all data from which it is retrieved. For example, when listing the employees that are assigned to projects, it might be interesting to see the employees that are not assigned to projects, too.

Practicing with Joins

The following outer join retrieves employee names from the *Employee* table and project IDs from the *Employee_project* table. It retrieves all the employee names, because this is a *left* outer join, and *Employee* is the left table.

```
SELECT e.full_name, p.proj_id

FROM Employee e LEFT OUTER JOIN Employee_project p

ON e.emp_no = p.emp_no

ORDER BY p.proj_id
```

This should produce a list of 48 names. Notice that some employees are not assigned to a project; the *proj_id* column displays `<null>` for them.

3 Now reverse the order of the tables and execute the query again.

```
SELECT e.full_name, p.proj_id

FROM Employee_project p LEFT OUTER JOIN Employee e

ON e.emp_no = p.emp_no

ORDER BY p.proj_id
```

This produces a list of 28 names. Notice that you get different results because this time the left table is *Employee_project*. The left outer join is only required to produce all the rows of the *Employee_project* table, not all of the *Employee* table.

4 As a last experiment with joins, repeat the query in Exercise 2 (the one you just did), but this time do a *right* outer join. Before you execute the query, think about it for a moment. What do you think this query will return?

```
SELECT e.full_name, p.proj_id

FROM Employee_project p RIGHT OUTER JOIN Employee e

ON e.emp_no = p.emp_no

ORDER BY p.proj_id
```

You should get the same result set as in Exercise 1. Did you realize that performing a right outer join on tables B JOIN A is the same as a left outer join on tables A JOIN B?

FORMATTING DATA

This section describes three ways to change data formats:

- Using CAST to convert datatypes
- Using the string operator to concatenate strings
- You can convert characters to uppercase

USING CAST TO CONVERT DATATYPES

Normally, only similar datatypes can be compared in search conditions, but you can work around this by using CAST. Use the CAST clause in search conditions to translate one datatype into another. The syntax for the CAST clause is:

```
CAST (<value> | NULL AS datatype)
```

For example, the following WHERE clause uses CAST to translate a CHAR datatype, INTERVIEW_DATE, to a DATE datatype. This conversion lets you compare INTERVIEW_DATE to another DATE column, *hire_date*:

```
. . . WHERE hire_date = CAST(interview_date AS DATE)
```

You can use CAST to compare columns in the same table or across tables. CAST allows the conversions listed in the following table:

From datatype	To datatype
NUMERIC	CHARACTER, DATE
CHARACTER	NUMERIC, DATE
DATE	CHARACTER, NUMERIC

Table 7: Compatible Datatypes for CAST

USING THE STRING OPERATOR IN SEARCH CONDITIONS

The string operator, also referred to as a *concatenation operator*, ||, joins two or more character strings into a single string. The strings to be joined can be the result set of a query or can be quoted strings that you supply. The operator is the pipe character, typed twice.

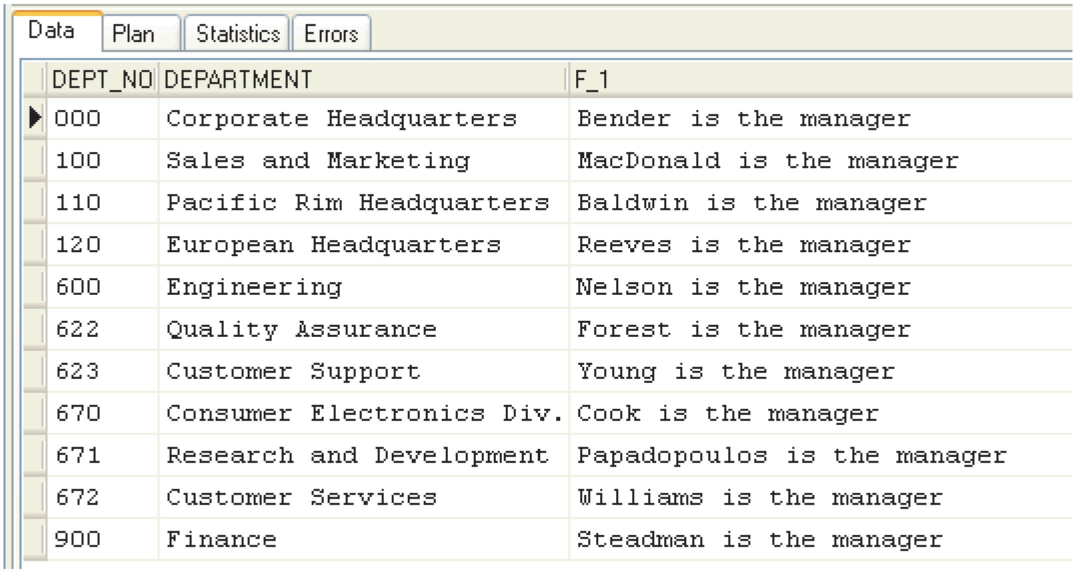
Using the String Operator to Join Strings

1 Execute the following SELECT statement to concatenate the result of the query with the additional text “ is the manager.” Remember to have a space as the first character of the string. The query returns the manager names for all department that are not field offices.

```
SELECT D.dept_no, D.department, E.last_name || ' is the manager'

FROM Department D, Employee E
```

```
WHERE D.mngr_no = E.emp_no AND D.department NOT CONTAINING 'Field'

ORDER BY D.dept_no
```

You should see the following result:

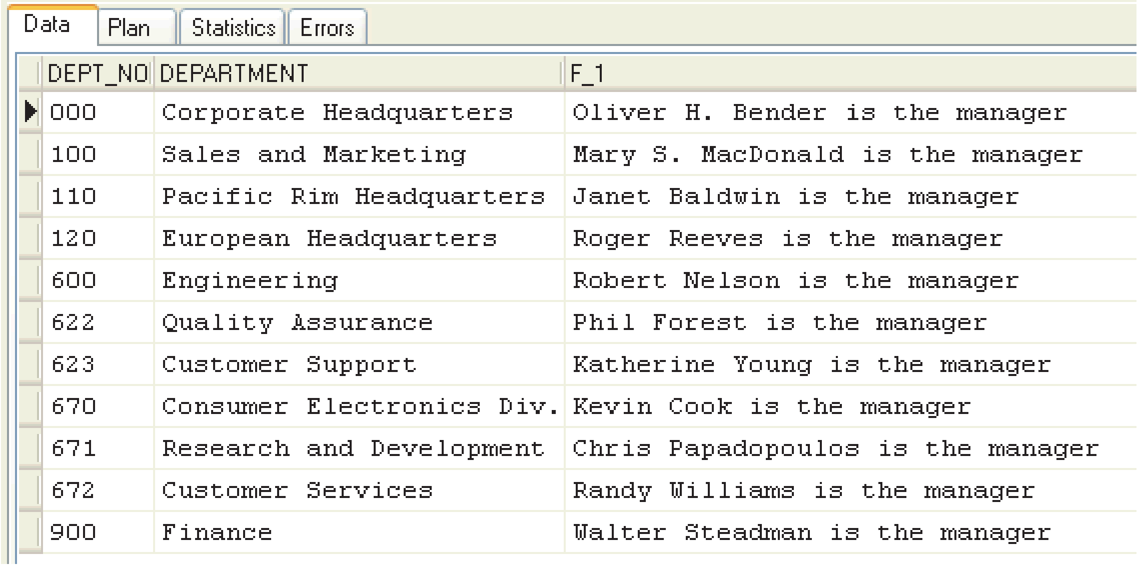


DEPT_NO	DEPARTMENT	F_1
000	Corporate Headquarters	Bender is the manager
100	Sales and Marketing	MacDonald is the manager
110	Pacific Rim Headquarters	Baldwin is the manager
120	European Headquarters	Reeves is the manager
600	Engineering	Nelson is the manager
622	Quality Assurance	Forest is the manager
623	Customer Support	Young is the manager
670	Consumer Electronics Div.	Cook is the manager
671	Research and Development	Papadopoulos is the manager
672	Customer Services	Williams is the manager
900	Finance	Steadman is the manager

2 You can concatenate as many strings as you like. The following query is a slight variation on the previous one: it concatenates the first name to the other output strings:

```
SELECT D.dept_no, D.department, E.first_name || ' ' || E.last_name
||

' is the manager'

FROM Department D, Employee E

WHERE D.mngr_no = E.emp_no AND D.department NOT CONTAINING 'Field'

ORDER BY D.dept_no
```

Notice that in order to get a space between the first and last names, you have to concatenate a string that consists solely of a space. The result set should look like this:



DEPT_NO	DEPARTMENT	F_1
000	Corporate Headquarters	Oliver H. Bender is the manager
100	Sales and Marketing	Mary S. MacDonald is the manager
110	Pacific Rim Headquarters	Janet Baldwin is the manager
120	European Headquarters	Roger Reeves is the manager
600	Engineering	Robert Nelson is the manager
622	Quality Assurance	Phil Forest is the manager
623	Customer Support	Katherine Young is the manager
670	Consumer Electronics Div.	Kevin Cook is the manager
671	Research and Development	Chris Papadopoulos is the manager
672	Customer Services	Randy Williams is the manager
900	Finance	Walter Steadman is the manager

Converting to Uppercase

The UPPER function converts character values to uppercase. For example, you could include a CHECK constraint that ensures that all column values are entered in uppercase when defining a table column or domain. The following CREATE DOMAIN statement uses the UPPER function to guarantee that column entries are all upper case:

```
CREATE DOMAIN PROJNO

AS CHAR(5)

CHECK (VALUE = UPPER (VALUE));
```

Advance to Part V

Click here to advance to the next section: Part V: Advanced Topics

PART V: ADVANCED TOPICS

This chapter provides examples of some advanced DDL features, including:

- Granting and revoking access privileges
- Creating and using triggers
- Creating and using stored procedures

ACCESS PRIVILEGES

Initially, only a table's creator, its *owner*, and the SYSDBA user have access to a table. On UNIX servers that have a superuser, or a user with root privileges, those users also have access to all database objects.

You can grant other users the right to look at or change your tables by assigning *access privileges* using the GRANT statement. Table 1 lists the available access privileges:

Privilege	Access
ALL	SELECT, DELETE, INSERT, UPDATE, and REFERENCES; note that ALL does not include the EXECUTE privilege
SELECT	Read data
DELETE	Delete data
INSERT	Write new data
UPDATE	Modify existing data
EXECUTE	Execute or call a stored procedure
REFERENCES	Reference a primary key with a foreign key
ROLE	All privileges assigned to the role

Table 1: SQL Access Privileges

The GRANT statement assigns access privileges for a table or view to specified users, roles, or procedures. The REVOKE statement removes previously granted access privileges.

ASSIGNING PRIVILEGES WITH GRANT

The GRANT statement can grant one or more privileges to one or more users. The privileges can be to one or more complete tables or can be restricted to certain columns of the tables. Only UPDATE and REFERENCES privileges can be assigned at the column level.

Granting Access to Whole Tables

The following statement grants one privilege on the *Department* table to one user:

```
GRANT SELECT ON Department TO EMIL
```

The following statement assigns two privileges (INSERT and UPDATE) on the *Department* table to three users:

```
GRANT INSERT, UPDATE ON Department TO EMIL, RAVI, HELGA
```

To grant privileges to everyone, use the PUBLIC keyword. The following statement grants all privileges except EXECUTE on the *Department* table to anyone who connects to the database:

```
GRANT ALL ON Department to PUBLIC
```

Granting Access to Columns

In the previous examples, users were granted access to entire tables. Often, however, you may want to grant access only to certain columns of a table. The following statement assigns UPDATE privilege to all users for the *contact* and *phone* columns in the *Customers* table:

This is a very brief introduction to an important topic: security. For more information about granting access, see the Data Definition Guide.

Revoking Privileges

The REVOKE statement removes access privileges that were granted with GRANT. The following statement removes the insert and update privileges on the *Department* table that were granted to Emil, Ravi, and Helga in an earlier example.

```
REVOKE INSERT, UPDATE ON Department FROM EMIL, RAVI, HELGA
```

USING ROLES TO CONTROL SECURITY

MarketPlace® is compliant with entry- and mid-level SQL92 standards regarding roles. A *role* is a named group of privileges.

In practice, a company might want to grant a particular collection of privileges to its sales people and a different collection of privileges to its accounting staff. The privileges list in each case might be quite complex. Without roles, you would have to use a lengthy and detailed GRANT statement each time a new sales or accounting person joined the company. The role feature avoids that.

Implementing roles is a four-step process:

Step 1: Define the role.

```
CREATE ROLE role_name
```

Step 2: Grant privileges to the role

```
GRANT {one or more of INSERT, UPDATE, DELETE, SELECT, REFERENCES,
       EXECUTE}
   ON table_name to role_name

GRANT UPDATE (col_name1, col_name 2) ON table_name TO role_name
```

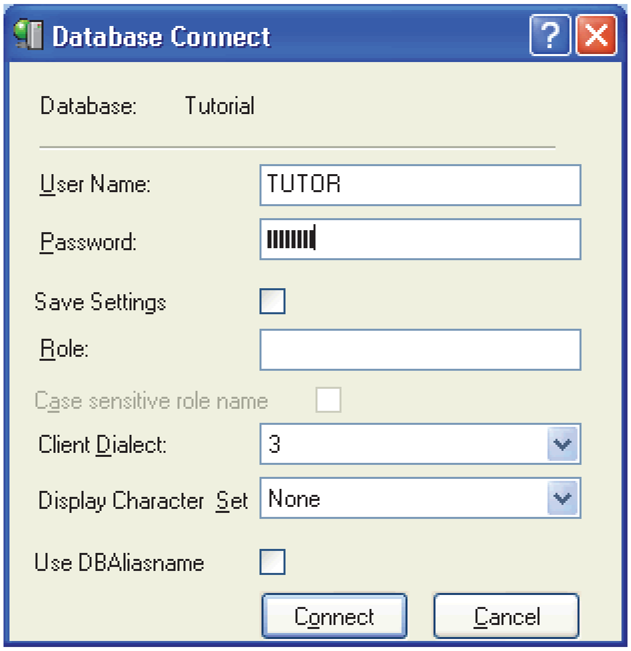
When the access is restricted to certain columns, as in the second line, only UPDATE and REFERENCES can be granted. EXECUTE must always be granted in a separate statement.

Step 3: Grant the role to users

```
GRANT role_name TO user_name1, user_name2
```

The users have now have all the privileges that were granted to the *role_name* role. But there's an additional step they must take before those privileges are available to them. They must specify the role when the connect to a database.

Step 4: Specify the role when connecting to a database.



Database: Tutorial

User Name: TUTOR

Password: ||||||||

Save Settings □

Role:

Case sensitive role name □

Client Dialect: 3

Display Character Set: None

Use DBAliasname □

Committing your Work

If you have not committed your work in IBConsole lately, do so now (**Transaction > Commit**). That way, if anything goes astray, you can roll back your work to this point (**Transaction > Rollback**).

Using a Role to Control Access

This exercise takes you through all four steps of implementing a role. You begin by creating another user and trying to access one of your tables when you are connected as this new user, in order to experience InterBase's security at work. Then you create the Salesperson role, assign some privileges to it, assign the role to your new user, and finally, repeat the access that failed earlier to experience that your new user now has the necessary permissions.

1 Logout from the server.

2 Disconnect from the Tutorial database by selecting **Database > Disconnect.**

3 Login to the server as SYSDBA (password "masterkey) by choosing **Server > Login**

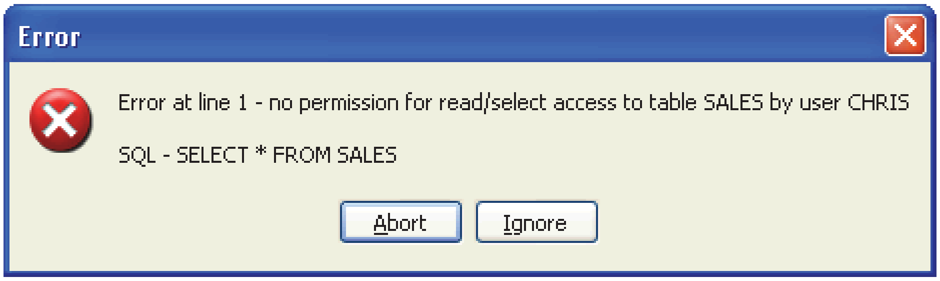
4 Create a new user called CHRIS with a password of chris4ib. Refer to "Creating a New User" on page 6 if you've forgotten how to create a user.

5 Highlight the Tutorial database icon in the left pane and choose **Database > Connect As** to connect to the database as Chris:

6 Execute the following SELECT statement:

```
SELECT * FROM SALES
```

NOTE: InterBase issues an error statement, because all those tables you've created in this tutorial belong to user TUTOR. User CHRIS doesn't have permission to do anything at all with them.



Error at line 1 - no permission for read/select access to table SALES by user CHRIS

SQL - SELECT * FROM SALES

7 Now disconnect from CHRIS and reconnect to the TUTORIAL database again, this time as TUTOR. Leave the Role field empty.

8 Create a role called SALESPEOPLE.

```
CREATE ROLE SALESPEOPLE
```

9 Execute the following GRANT statements to assign privileges to the SALESPERSON role.

NOTE: Remember that you must execute each GRANT statement before entering the next one.

```
   GRANT SELECT, UPDATE, INSERT ON Sales to SALESPEOPLE
   GRANT update (contact_first, contact_last, phone_no) ON Customer
to SALESPEOPLE
```

10 Grant the SALESPEOPLE role to user CHRIS.

```
GRANT SALESPEOPLE TO CHRIS
```

11 Connect to the TUTORIAL database as user CHRIS.

12 Now enter the same query that failed in Step 5. This time, InterBase retrieves all the rows in the *Sales* table, because CHRIS now has the required permissions, thanks to the role.

13 Now reconnect to the database as user TUTOR.

TRIGGERS AND STORED PROCEDURES

Stored procedures and triggers are part of a database's metadata and are written in *stored procedure and trigger language*, an InterBase extension to SQL. Procedure and trigger language includes SQL data manipulation statements and some powerful extensions, including IF … THEN … ELSE, WHILE … DO, FOR SELECT … DO, exceptions, and error handling.

- A *stored procedure* is a program that can be called by applications or from IBConsole.

Stored procedures can be invoked directly from applications, or can be substituted for a table or view in a SELECT statement. They can receive input parameters from and return values to the calling application.

- A *trigger* is a self-contained routine that is associated with a table. A trigger definition specifies an action to perform when a specified event, such as an update, insert, or delete occurs on the table. A trigger is never called directly by an application or user. Instead, when an application or user attempts to perform the action stated in the trigger definition, the trigger automatically executes, or *fires*.

See the Data Definition Guide for a full explanation of stored procedures and triggers.

TRIGGERS

Triggers have a great variety of uses, but in general, they permit you to automate tasks that would otherwise have to be done manually. You can use them to define actions that should occur automatically when data is inserted, updated, or deleted in a particular table. Triggers are a versatile tool, with a wide range of uses.

The triggers defined in the TUTORIAL database perform the following actions:

- The *set_emp_no* trigger generates and inserts unique employee numbers when a row is inserted into the *Employee* table and the *set_cust_no* trigger does the same for customer numbers in the *Customer* table.
- The *save_salary_change* trigger maintains a record of employees' salary changes.
- The *new_order* trigger posts an event when a new row is inserted into the *Sales* table.

Using SET TERM

In command-line `isql` and SQL scripts, the trigger statement must be preceded by a SET TERM statement that defines what characters will terminate the trigger statement, since the SQL statements in the body of a trigger must each end with a semicolon (;). The double exclamation mark (!!) is a common choice for this terminator.

In environments where the terminator was changed using the SET TERM statement, the trigger code should be followed by another SET TERM statement to change the terminator back to a semicolon.

The following example changes the terminator to !!, but ends with the current terminator (semicolon), because that is the one in effect for this statement:

```
SET TERM !!;
```

Changing the Terminator in IBConsole

To change the terminator in IBConsole, choose **Edit >| Preferences** and click on the Interactive SQL tab. Enter a double exclamation mark (!!), or another terminator if you prefer, in the Terminator field.

The Structure of Triggers

- The CREATE TRIGGER keywords are followed by the trigger *name* and the *table* for which the trigger is defined.
- The next line determines when the trigger fires. Choices are before or after an insert, deletion, or update. If more than one trigger is defined for a particular point—such as AFTER INSERT—you can add a *position number* that specifies the sequence in which the trigger should fire.
- If the trigger uses local variables, they are declared next.
- The variables, if any, are followed by SQL statements that determine the behavior of the trigger. These statements are bracketed between the keywords BEGIN and END. Each of these SQL statements ends with a semicolon.
- If you are working in `isql` or writing a script, the END keyword is followed by the terminator that was defined by the SET TERM statement.

The syntax for a trigger looks like this.

```
CREATE TRIGGER trigger_name FOR Table_name
{BEFORE | AFTER} {INSERT | DELETE | UPDATE} [POSITION NUMBER]
AS
[DECLARE VARIABLE variable_name datatype;]
BEGIN
   statements in InterBase procedure and trigger language
END
```

See **Triggers.sql** for an example. The *Language Reference*, *Data Definition Guide*, and *Programmer's Guide* all contain more discussion of triggers.

Generators: Generating Unique Column Values

There are many cases in which table columns require unique, sequential values. The *emp_no* column of the *Employee* table is a good example. Without a trigger, you would have to know what the last employee number is each time you add a row for a new employee, so that you could increment it by one to create the new employee number. This is cumbersome and error prone.

Triggers provide a way to automate this process, by using a handy database object called a *generator*. A generator is a named variable that is called and incremented through the **gen_id( )** function. The value of the generator is initialized with SET GENERATOR. After that, it generates the next incremental value each time **gen_id( )** is called. The **gen_id()** function takes a generator name and an increment as inputs.

Context Variables

Context variables are unique to triggers. Triggers are often used to change a value, and in the process of doing so, must temporarily store both the old and new values. The context variables, *Old* and *New*, are the mechanisms by which triggers do this. As you perform the exercises in this section, look for them in contexts such as `New.emp_no = gen_id(emp_no_gen, 1).` For more information about context variables, see the Data Definition Guide

Creating a Generator

1 Begin by checking the employee numbers in the *Employee* table, to confirm that the highest employee number currently in use is 145:

```
SELECT emp_no from Employee
   ORDER BY emp_no
```

NOTE: The statement above returns all the employee numbers so that you can confirm that 145 is the highest. The following statement produces the same information more efficiently:

```
SELECT max(emp_no) from Employee
```

2 Triggers often use generators, and the trigger you create in the next exercise is an example of one. Execute the following statement to create a generator called *emp_no_gen*.

```
CREATE GENERATOR emp_no_gen
```

3 Now initialize the generator to 145, the highest value currently in use.

```
SET GENERATOR emp_no_gen TO 145
```

Creating a Trigger that Generates a Value

1 The next statements define a trigger named *set_emp_no* that makes use of the *emp_no_gen* generator to generate unique sequential employee numbers and insert them into the *Employee* table.

```
CREATE TRIGGER set_emp_no FOR Employee
BEFORE INSERT AS
   BEGIN
   New.emp_no = gen_id(emp_no_gen, 1);
   END
```

This statement says that the *set_emp_no* trigger will fire before an insert operation, and that it will create a new value for *emp_no* by calling the gen_id() function on the *emp_no_gen* generator with an increment of 1.

2 To test the generator, execute the following INSERT statement:

```
INSERT INTO Employee (first_name, last_name, dept_no, job_code,
 job_grade,job_country, hire_date, salary, phone_ext)
   VALUES ('Reed', 'Richards', '671', 'Eng', 5, 'USA', '07/27/95',
      '34000', '444')
```

Notice that you did not include a value for the emp_no column in the INSERT statement. Look at the new record by entering

```
SELECT * from Employee WHERE last_name = 'Richards'
```

The employee number is 146. Remember that the highest employee number before you created the generator and inserted a new row was 145. The trigger has automatically assigned the new employee the next employee number.

3 If your INSERT ran without errors and your SELECT returns the correct result set, commit your work by selecting **Transactions > Commit**.

Finishing the Trigger Exercises

The remainder of this section on triggers takes you through the process of creating another generator and three more triggers. The text instructs you to enter them by hand in order to get more experience with them.

TIP: If you want to save time, you can use the **Query > Load Script** command to read in the **Triggers.sql** script in place of entering the remaining trigger and generator statements yourself. **Triggers.sql** defines another generator and a trigger named set_cust_no that assigns unique customer numbers. It defines two other triggers: save_salary_change and post_new_order.

Whether you choose to enter the remaining trigger statements yourself or to run the script, do take time to open **Triggers.sql** in a text editor and see that you understand the code in it. Notice that triggers in a script require the use of the SET TERM statement. (See "The Structure of Triggers" on page 6 for more about SET TERM.)

If you choose to work through this section manually instead of running the script, commit your work after creating and testing each trigger or stored procedure.

More Practice with Generators and Triggers

1 The next trigger that you will create uses the *cust_no_gen* generator.

NOTE: Execute each statement in turn to create and initialize this generator:

```
CREATE GENERATOR cust_no_gen

SET GENERATOR cust_no_gen to 1015
```

Remember, these are two separate statements, and you must execute each one before entering the next.

2 Now execute the following CREATE TRIGGER statement to create the *set_cust_no* trigger.

```
CREATE TRIGGER set_cust_no FOR Customer
   BEFORE INSERT AS
   BEGIN
      new.cust_no = gen_id(cust_no_gen, 1);
   END
```

3 To test this trigger, first confirm that the highest customer number is 1015.

```
SELECT max(cust_no) WHERE customer =
```

4 Then insert the following row:

```
INSERT INTO Customer (customer, contact_first, contact_last,
   phone_no, address_line1, address_line2, city, state_province,
      country, postal_code, on_hold)
   VALUES ('Big Rig', 'Henry', 'Erlig', '(701) 555-1212', '100 Big
Rig Way',
      NULL, 'Atlanta', 'GA', 'USA', '70008', NULL)
```

Now perform the following SELECT to confirm that the new customer number is, as you expect, 1016:

```
SELECT cust_no FROM Customer WHERE customer = 'Big Rig'
```

Creating a Trigger to Maintain Change Records

Enter the following CREATE TRIGGER statement to create the *save_salary_change* trigger, which maintains a record of changes to employees’ salaries in the *Salary_history* table:

```
CREATE TRIGGER save_salary_change FOR Employee
   AFTER UPDATE AS
   BEGIN
      IF (Old.salary <> New.salary) THEN
      INSERT INTO Salary_history
      (emp_no, change_date, UPDATER_ID, old_salary, percent_change)
      VALUES (
         OLD.emp_no,
         'NOW',
         USER,
         OLD.salary, (NEW.salary - OLD.salary) * 100 / OLD.salary
      );
   END
```

This trigger fires AFTER UPDATE of the *Employee* table. It compares the value of the *salary* column before the update to the value after the update and if they are different, it enters a record in *Salary_history* that consists of the employee number, date, previous salary, and percentage change in the salary.

Notice that when the values to be entered in the *Salary_history* table are to be taken from the *Employee* table, they are always preceded by the *Old* or *New* context variable. That is because InterBase creates two versions of a record during the update process, and you must specify which version the value is to come from.

In addition, note that this example makes use of two other InterBase features: it inserts the current date into a column of DATE datatype by supplying the string `'NOW'` in single quotes, and it inserts the name of the user who is currently connected to the database by supplying the keyword USER.

Update an employee record and change the salary to see how this trigger works.

Creating a Trigger that Posts an Event

Execute the following CREATE TRIGGER statement to create a trigger, *post_new_order*, that posts an event named "new_order" whenever a record is inserted into the *Sales* table.

```
CREATE TRIGGER post_new_order FOR Sales
AFTER INSERT AS
BEGIN
   POST_EVENT 'new_order';
END
```

An *event* is a message passed by a trigger or stored procedure to the InterBase event manager to notify interested applications of the occurrence of a particular condition. Applications that have registered interest in an event can pause execution and wait for the specified event to occur. For more information on events, see the Programmer's Guide.

The *post_new_order* trigger fires after a new record is inserted into the *Sales* table—in other words when a new sale is made. When this event occurs, interested applications can take action, such as printing an invoice or notifying the shipping department.

STORED PROCEDURES

Stored procedures are programs stored with a database's metadata that run on the server. Applications can call stored procedures to perform tasks, and you can also use stored procedures in IBConsole. See the Programmer's Guide for more information on calling stored procedures from applications.

There are two types of stored procedures:

- *Select procedures* that an application can use in place of a table or view in a SELECT statement. A select procedure must be defined to return one or more values (output parameters), or an error results. Since select procedures can return more than one row, they appear as a table or view to a calling program.
- *Executable procedures* that an application can call directly with the EXECUTE PROCEDURE statement. Executable procedures can perform a variety of tasks; they might or might not return values to the calling program.

Both kinds of procedures are defined with CREATE PROCEDURE and have essentially the same syntax. The difference is in how the procedure is written and how it is intended to be used.

Stored Procedure Syntax

A CREATE PROCEDURE statement is composed of a *header* and a *body*. The header contains:

- The *name* of the stored procedure, which must be unique among procedure, view, and table names in the database.
- An optional list of *input parameters* and their datatypes that a procedure receives from the calling program.
- If the procedure returns values to the calling program, the next item is the RETURNS keyword, followed by a list of *output parameters* and their datatypes.

The procedure body contains:

- An optional list of *local variables* and their datatypes.
- A *block* of statements in InterBase procedure and trigger language, bracketed by BEGIN and END. A block can itself include other blocks, so that there might be many levels of nesting.

The simplified syntax of a procedure looks like this:

```
CREATE PROCEDURE procedure_name
   [(input_var1 datatype[, input_var2 datatype …])]
   [RETURNS (output_var1 datatype[, output_var2 datatype …])]
   AS
      BEGIN
         statements in InterBase procedure and trigger language
      END
```

Like trigger definitions, procedure definitions in SQL scripts, embedded SQL, and command-line isql must be preceded by a SET TERM statement that sets the terminator to something other than a semicolon. When all procedure statements have been entered, the SET TERM statement must be used again to set the terminator back to a semicolon. See **Procs.sql** for an example. The *Language Reference*, *Data Definition Guide*, and *Programmer's Guide* all contain more information on stored procedures.

Creating a Simple SELECT Procedure

1 Execute the following code to create the *get_emp_proj* procedure. There is a useful convention of starting variable names with "v_" to help make the code readable, but it is not required. You can name variables anything you wish. The following code is a single SQL statement. Enter the whole thing and then execute it:

```
CREATE PROCEDURE get_emp_proj (v_empno SMALLINT)
RETURNS (project_id CHAR(5))
AS
BEGIN
   FOR SELECT proj_id
      FROM Employee_project
      WHERE emp_no = :v_empno
      INTO :project_id
   DO
      SUSPEND;
END
```

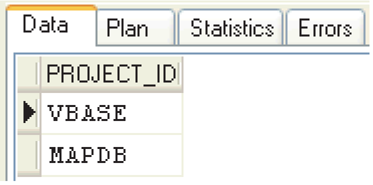
This is a select procedure that takes an employee number as its input parameter (*v_empno*, specified in parentheses after the procedure name) and returns all the projects to which the employee is assigned (*project_id*, specified after RETURNS). The variables are named in the header as *varname* and then referenced in the body as :*varname*.

It uses a FOR SELECT ... DO statement to retrieve multiple rows from the *Employee_project* table. This statement retrieves values just as a normal SELECT statement does, but retrieves them one at a time into the variable listed after INTO, and then performs the statements following DO. In this case, the only statement is SUSPEND, which suspends execution of the procedure and sends values back to the calling application (in this case, **InterBase Windows > SQL**).

2 See how the procedure works by entering the following query:

```
SELECT * FROM get_emp_proj(71)
```

This query looks at first as though there were a table named *get_emp_proj*, but you can tell that it's a procedure rather than a table because of the input parameter in parentheses following the procedure name. The results are:



PROJECT_ID
VBASE
MAPDB

These are the projects to which employee number 71 is assigned. Try it with some other employee numbers.

Create a More Complex Select Procedure

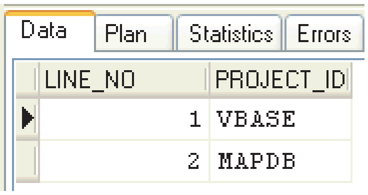
1 The next exercise starts with the code for the previous procedure and adds an output column that counts the line numbers. Execute the following code. (You can display the previous query if you wish and add the new code.)

```
CREATE PROCEDURE get_emp_proj2 (v_empno SMALLINT)
   RETURNS (line_no integer, project_id CHAR(5))
   AS
   BEGIN
   line_no = 0;
      FOR SELECT proj_id
         FROM Employee_project
         WHERE emp_no = :v_empno
         INTO :project_id
      DO
         BEGIN
            line_no = line_no+1;
            SUSPEND;
         END
   END
```

2 To test this new procedure, execute the following query:

```
SELECT * FROM get_emp_proj2(71)
```

You should see the following output:



LINE_NO	PROJECT_ID
1	VBASE
2	MAPDB

3 If your procedure returns the correct result set, commit your work.

Creating a Simple Executable Procedure

1 The executable procedure that you create in the next step of this exercise, *add_emp_proj*, makes use of an *exception*, a named error message, that you define with CREATE EXCEPTION. Execute the following SQL statement to create the UNKNOWN_EMP_ID exception:

```
CREATE EXCEPTION unknown_emp_id
   'Invalid employee number or project ID.'
```

Once defined, this exception can be *raised* in a trigger or stored procedure with the EXCEPTION clause. The associated error message is then returned to the calling application.

2 Execute the following statement to create the *add_emp_proj* stored procedure:

```
CREATE PROCEDURE add_emp_proj (v_empno SMALLINT, v_projid CHAR(5))
AS
BEGIN
   INSERT INTO Employee_project (emp_no, proj_id)
   VALUES (:v_empno, :v_projid);
   WHEN SQLCODE -530 DO
      EXCEPTION UNKNOWN_EMP_ID;
END
```

This procedure takes an employee number and project ID as input parameters and adds the employee to the specified project using an INSERT statement. The error-handling WHEN statement checks for SQLCODE -530, violation of foreign key constraint, and then raises the previously-defined exception when this occurs.

3 Practice using this procedure by executing the following SQL statement:

```
EXECUTE PROCEDURE add_emp_proj(20, 'DGPII')
```

To confirm that this worked, execute the following SELECT statement:

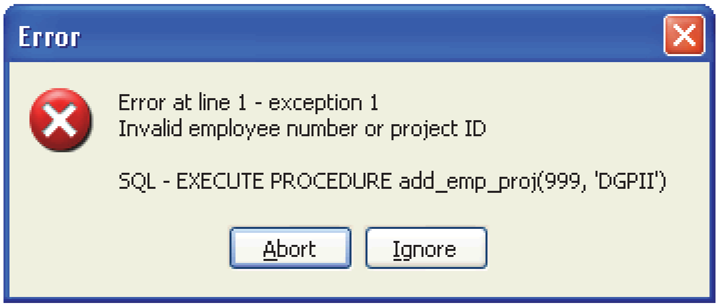
```
SELECT * FROM Employee_project where emp_no = 20
```

You should see that employee 20 is now assigned to both the DGPII project and the GUIDE project.

4 Now try adding a non-existent employee to a project, for example:

```
EXECUTE PROCEDURE add_emp_proj(999, 'DGPII')
```

The statement fails and the exception message displays on the screen.



Error at line 1 - exception 1
Invalid employee number or project ID

SQL - EXECUTE PROCEDURE add_emp_proj(999, 'DGPII')

Recursive Procedures

Stored procedures support *recursion*, that is, they can call themselves. This is a powerful programming technique that is useful in performing repetitive tasks across hierarchical structures such as corporate organizations or mechanical parts.

Creating a Recursive Procedure

In this exercise, you create a stored procedure called *dept_budget* that takes a department number as its input parameter and returns the budget of the department and all departments that are under it in the corporate hierarchy. It uses *local variables* declared with DECLARE VARIABLE statements. These variables are used only within the context of the procedure.

First, the procedure retrieves the budget of the department given as the input parameter from the *Department* table and stores it in the *total_budget* variable. Then it retrieves the number of departments reporting to that department using the COUNT aggregate function. If there are no reporting departments, it returns the value of *total_budget* with SUSPEND.

Using a FOR SELECT … DO loop, the procedure then retrieves the department number of each reporting department into the local variable *rdno*, and then recursively calls itself with

EXECUTE PROCEDURE dept_budget :rdno RETURNING_VALUES :sumb

This statement executes *dept_budget* with input parameter *rdno*, and puts the output value in *sumb*. Notice that when using EXECUTE PROCEDURE within a procedure, the input parameters are not put in parenthesis, and the variable into which to put the resultant output value is specified after the RETURNING_VALUES keyword. The value of *sumb* is then added to *total_budget*, to keep a running total of the budget. The result is that the procedure returns the total of the budgets of all the reporting departments given as the input parameter plus the budget of the department itself.

1 Execute the following SQL statement:

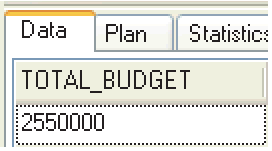
```
   CREATE PROCEDURE dept_budget (v_dno CHAR(3))
   RETURNS (total_budget NUMERIC(15, 2))
   AS
      DECLARE VARIABLE sumb DECIMAL(12, 2);
      DECLARE VARIABLE rdno CHAR(3);
      DECLARE VARIABLE cnt INTEGER;
   BEGIN
      total_budget = 0;
      SELECT budget FROM Department WHERE dept_no = :v_dno INTO
:total_budget;
      SELECT COUNT(budget)
         FROM Department
         WHERE head_dept = :v_dno
         INTO :cnt;
      IF (cnt = 0) THEN
         SUSPEND;
      FOR SELECT dept_no
         FROM Department
         WHERE head_dept = :v_dno
         INTO :rdno
```

```
        DO
        BEGIN
           EXECUTE PROCEDURE Dept_budget :rdno RETURNING_VALUES :sumb;
              total_budget = total_budget + sumb;
        END
     END
```

2 To find the total budget for department 620, including all its subdepartments, execute the following SQL statement:

```
EXECUTE PROCEDURE dept_budget(620)
```

The result is:



```
TOTAL_BUDGET
2550000
```

A Little about Datatype Conversion

Notice that the *dept_budget* procedure is defined to take a CHAR(3) as its input parameter, but that you can get away with giving it an integer (without quotes). This is because of InterBase's automatic type conversion, which converts datatypes, where possible, to the required datatype. It automatically converts the integer 620 to the character string "620". The automatic type conversion won't work for department number 000, however, because it would convert to the string "0", which is not a department number.

More Procedures

There are a number of other procedures, some quite complex, defined in **Procs.sql** for the TUTORIAL database. Now that you have a basic understanding of procedures, it will be worth your while to read them over so that you understand them. Then try using them. Notice that comments are often included within the text of a statement to make it easier for people to understand what the code is doing.

This completes the InterBase Update 4 Quick Start. For additional information on each of these topics, please refer to the InterBase Guides located at: http://docs.embarcadero.com/products/interbase/