InterBase 5 Server

Data Definition Guide

InterBase®

SOFTWARE CORPORATION

Table of Contents

Chapter 7   Working with Indexes

Chapter 8   Working with Views

Chapter 9   Working with Stored Procedures

Chapter 14 Character Sets and Collation Orders

Appendix A InterBase Document Conventions

List of Tables

List of Figures

1

Using the Data Definition Guide

The InterBase *Data Definition Guide* provides information necessary for creating a database and database objects with SQL. This chapter also describes:

- Who should read this book.
- Other InterBase documentation that will help you define a database.
- A brief overview of the contents of this book.

What is data definition?

An InterBase database is created and populated using SQL statements, which can be divided into two major categories: data definition language (DDL) statements and data manipulation language (DML) statements.

The underlying structures of the database—its tables, views, and indexes—are created using DDL statements. Collectively, the objects defined with DDL statements are known as *metadata*. *Data definition* is the process of creating, modifying, and deleting metadata. Conversely, DML statements are used to populate the database with data, and to manipulate existing data stored in the structures previously defined with DDL statements. The focus of this book is how to use DDL statements. For more information on using DML statements, see the *Language Reference*.

DDL statements that create metadata begin with the keyword CREATE, statements that modify metadata begin with the keyword ALTER, and statements that delete metadata begin with the keyword DROP. Some of the basic data definition tasks include:

- Creating a database (CREATE DATABASE).
- Creating tables (CREATE TABLE).
- Altering tables (ALTER TABLE).
- Dropping tables (DROP TABLE).

In InterBase, metadata is stored in system tables, which are a set of tables that is automatically created when you create a database. These tables store information about the structure of the database. All system tables begin with "RBD$". Examples of system tables include RDB$RELATIONS, which has information about each table in the database, and RDB$FIELDS, which has information on the domains in the database. For more information about the system tables, see the *Language Reference*.

IMPORTANT   You can directly modify columns of a system table, but unless you understand all of the interrelationships between the system tables, modifying them directly can adversely affect other system tables and disrupt your database.

Who should use this guide

The *Data Definition Guide* is a resource for programmers, database designers, and users who create or change an InterBase database or its elements.

This book assumes the reader has:

- Previous understanding of relational database concepts.

▪ Read the **isql** sections in the InterBase *Getting Started* book.

Related InterBase documentation

The *Language Reference* is the main reference companion to the *Data Definition Guide*. It supplies the complete syntax and usage for SQL data definition statements. For a complete list of books in the InterBase documentation set, see Appendix A, "InterBase Document Conventions."

Topics covered in this guide

The following table lists and describes the chapters in the *Data Definition Guide*:

Chapter	Description	SQL statements
Chapter 1, "Using the Data Definition Guide"	Overview of InterBase Data Definition features. Using **isql**, the SQL Data Definition Utility.	
Chapter 2, "Designing Databases"	Planning and designing a database. Understanding data integrity rules and using them in a database. Planning physical storage.	
Chapter 3, "Creating Databases"	Creating an InterBase database.	CREATE/ALTER/DROP DATABASE CREATE/ALTER/DROP SHADOW
Chapter 4, "Specifying Datatypes"	Choosing a datatype.	CREATE/ALTER TABLE CREATE/ALTER DOMAIN
Chapter 5, "Working with Domains"	Creating, altering, and dropping domains.	CREATE/ALTER/DROP DOMAIN
Chapter 6, "Working with Tables"	Creating and altering database tables, columns, and domains. Setting up referential integrity.	CREATE/ALTER/DROP TABLE
Chapter 7, "Working with Indexes"	Creating and dropping indexes.	CREATE/ALTER/DROP INDEX
Chapter 8, "Working with Views"	Creating and dropping views. Using WITH CHECK OPTION.	CREATE/DROP VIEW

TABLE 1.1   Chapter list for the *Data Definition Guide*

Chapter	Description	SQL statements
Chapter 9, "Working with Stored Procedures"	Using stored procedures. What you can do with stored procedures.	CREATE/ALTER/DROP PROCEDURE CREATE/ALTER/DROP EXCEPTION
Chapter 10, "Creating Triggers"	Using triggers. What you can do with triggers.	CREATE/ALTER/DROP TRIGGER CREATE/ALTER/DROP EXCEPTION
Chapter 11, "Declaring User-Defined Functions and BLOB Filters"	Defining user-defined functions and Blob filters.	DECLARE/DROP EXTERNAL FUNCTION DCELARE/DROP FILTER
Chapter 12, "Working with Generators"	Creating, setting, and resetting generators.	CREATE GENERATOR/SET GENERATOR
Chapter 13, "Planning Security"	Securing data and system catalogs with SQL: tables, views, triggers, and procedures.	GRANT, REVOKE
Chapter 14, "Character Sets and Collation Orders"	Specifying character sets and collation orders.	CHARACTER SET COLLATE
Appendix A, "InterBase Document Conventions"	Lists typefaces and special characters used in this book to describe syntax and identify object types.	

TABLE 1.1 Chapter list for the *Data Definition Guide* (*continued*)

Using isql

You can use **isql** to interactively create, update, and drop metadata, or you can input a file to **isql** that contains data definition statements, which is then executed by **isql** without prompting the user. It is usually preferable to use a data definition file because it is easier to modify the file than to retype a series of individual SQL statements, and the file provides a record of the changes made to the database.

The **isql** interface can be convenient for simple changes to existing data, or for querying the database and displaying the results. You can also use the interactive interface as a learning tool. By creating one or more sample databases, you can quickly become more familiar with InterBase.

Using a data definition file

A data definition file can include statements to create, alter, or drop a database, or any other SQL statement. To issue SQL statements through a data definition file, follow these steps:

1. Use a text editor to create the data definition file. Each DDL statement should be followed by a COMMIT to ensure its visibility to all subsequent DDL statements in the data definition file.

2. Save the file.

3. Input the file into **isql**. For information on how to input the data definition file using Windows ISQL, see the *Operations Guide*. For information on how to input the data definition file using command-line **isql**, see the *Operations Guide*.

2

Designing Databases

This chapter provides a general overview of how to design an InterBase database—it is not intended to be a comprehensive description of the principles of database design. This chapter includes:

- An overview of basic design issues and goals.
- A framework for designing the database.
- InterBase-specific suggestions for designing your database.
- Suggestions for planning database security.

Overview of design issues

A database describes real-world organizations and their processes, symbolically representing real-world objects as tables and other database objects. Once the information is organized and stored as database objects, it can be accessed by applications or a user interface displayed on desktop workstations and computer terminals.

The most significant factor in producing a database that performs well is good database design. Logical database design is an iterative process which consists of breaking down large, heterogeneous structures of information into smaller, homogenous data objects. This process is called *normalization*. The goal of normalization is to determine the natural relationships between data in the database. This is done by splitting a table into two or more tables with fewer columns. When a table is split during the normalization process, there is no loss of data because the two tables can be put back together with a join operation. Simplifying tables in this manner allows the most compatible data elements and attributes to be grouped into one table.

Database versus data model

It is important to distinguish between the *description* of the database, and the database itself. The description of the database is called the *data model* and is created at design time. The model is a template for creating the tables and columns; it is created before the table or any associated data exists in the database. The data model describes the logical structure of the database, including the data objects or entities, datatypes, user operations, relationships between objects, and integrity constraints.

In the relational database model, decisions about logical design are completely independent of the physical structure of the database. This separation allows great flexibility.

- **You do not have to define the physical access paths between the data objects at design time,** so you can query the database about almost any logical relationship that exists in it.

- **The logical structures that describe the database are not affected by changes in the underlying physical storage structures.** This capability ensures cross-platform portability. You can easily transport a relational database to a different hardware platform because the database access mechanisms defined by the data model remain the same regardless of how the data is stored.

- **The logical structure of the database is also independent of what the end-user sees.** The designer can create a customized version of the underlying database tables with *views*. A view displays a subset of the data to a given user or group. Views can be used to hide sensitive data, or to filter out data that a user is not interested in. For more information on views, see Chapter 8, "Working with Views."

Design goals

Although relational databases are very flexible, the only way to guarantee data integrity and satisfactory database performance is a solid database design—there is no built-in protection against poor design decisions. A good database design:

- **Satisfies the users' content requirements** for the database. Before you can design the database, you must do extensive research on the requirements of the users and how the database will be used.

- **Ensures the consistency and integrity of the data.** When you design a table, you define certain attributes and constraints that restrict what a user or an application can enter into the table and its columns. By validating the data before it is stored in the table, the database enforces the rules of the data model and preserves data integrity.

- **Provides a natural, easy-to-understand structuring of information**. Good design makes queries easier to understand, so users are less likely to introduce inconsistencies into the data, or to be forced to enter redundant data. This facilitates database updates and maintenance.

- **Satisfies the users' performance requirements**. Good database design ensures better performance. If tables are allowed to be too large, or if there are too many (or too few) indexes, long waits can result. If the database is very large with a high volume of transactions, performance problems resulting from poor design are magnified.

Design framework

The following steps provide a framework for designing a database:

1. Determine the information requirements for the database by interviewing prospective users.

2. Analyze the real-world objects that you want to model in your database. Organize the objects into entities and attributes and make a list.

3. Map the entities and attributes to InterBase tables and columns.

4. Determine an attribute that will uniquely identify each object.

5. Develop a set of rules that govern how each table is accessed, populated, and modified.

6. Establish relationships between the objects (tables and columns).

7. Plan database security.

The following sections describe each of these steps in more detail.

Analyzing requirements

The first step in the design process is to research the environment that you are trying to model. This involves interviewing prospective users in order to understand and document their requirements. Ask the following types of questions:

- Will your applications continue to function properly during the implementation phase? Will the system accommodate existing applications, or will you need to restructure applications to fit the new system?

- Whose applications use which data? Will your applications share common data?

- How do the applications use the data stored in the database? Who will be entering the data, and in what form? How often will the data objects be changed?

- What access do current applications require? Do your applications use only one database, or do they need to use several databases which might be different in structure? What access do they anticipate for future applications, and how easy is it be to implement new access paths?

- Which information is the most time-critical, requiring fast retrieval or updates?

Collecting and analyzing data

Before designing the database objects—the tables and columns—you need to organize and analyze the real-world data on a conceptual level. There are four primary goals:

- **Identify the major functions and activities of your organization.** For example: hiring employees, shipping products, ordering parts, processing paychecks, and so on.

- **Identify the objects of those functions and activities.** Building a business operation or transaction into a sequence of events will help you identify all of the entities and relationships the operation entails. For example, when you look at a process like "*hiring employees*," you can immediately identify entities such as the JOB, the EMPLOYEE, and the DEPARTMENT.

- **Identify the characteristics of those objects**. For example, the EMPLOYEE entity might include such information as EMPLOYEE_ID, FIRST_NAME, LAST_NAME, JOB, SALARY, and so on.

- **Identify certain relationships between the objects** For example, how do the EMPLOYEE, JOB, and DEPARTMENT entities relate to each other? The employee has one job title and belongs to one department, while a single department has many employees and jobs. Simple graphical flow charts help to identify the relationships.

FIGURE 2.1   Identifying relationships between objects
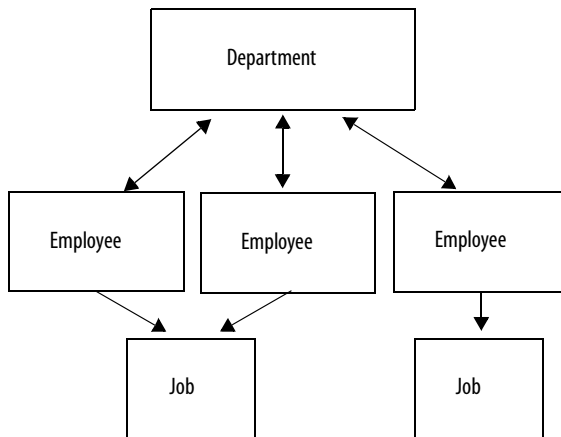
Identifying entities and attributes

Based on the requirements that you collect, identify the objects that need to be in the database—the entities and attributes. An *entity* is a type of person, object, or thing that needs to be described in the database. It might be an object with a physical existence, like a person, a car, or an employee, or it might be an object with a conceptual existence, like a company, a job, or a project. Each entity has properties, called *attributes*, that describe it. For example, suppose you are designing a database that must contain information about each employee in the company, departmental-level information, information about current projects, and information about customers and sales. The example below shows how to create a list of entities and attributes that organizes the required data.

Entities	Attributes
EMPLOYEE	Employee Number
	Last Name
	First Name
	Department Number
	Job Code
	Phone Extension
	Salary
DEPARTMENT	Department Number
	Department Name
	Department Head Name
	Department Head Employee Number
	Budget
	Location
	Phone Number
PROJECT	Project ID
	Project Name
	Project Description
	Team Leader
	Product

TABLE 2.1   List of entities and attributes

Entities	Attributes
CUSTOMER	Customer Number
	Customer Name
	Contact Name
	Phone Number
	Address
SALES	PO Number
	Customer Number
	Sales Rep
	Order Date
	Ship Date
	Order Status

TABLE 2.1   List of entities and attributes (*continued*)

By listing the entities and associated attributes this way, you can begin to eliminate redundant entries. Do the entities in your list work as tables? Should some columns be moved from one group to another? Does the same attribute appear in several entities? Each attribute should appear only once, and you need to determine which entity is the primary owner of the attribute. For example, DEPARTMENT HEAD NAME should be eliminated because employee names (FIRST NAME and LAST NAME) already exist in the EMPLOYEE entity. DEPARTMENT HEAD EMPLOYEE NUM can then be used to access all of the employee-specific information by referencing EMPLOYEE NUMBER in the EMPLOYEE entity. For more information about accessing information by reference, see "Establishing relationships between objects" on page 18.

The next section describes how to map your lists to actual database objects—entities to tables and attributes to columns.

Designing tables

In a relational database, the database object that represents a single entity is a *table*, which is a two-dimensional matrix of rows and columns. Each column in a table represents an attribute. Each row in the table represents a specific *instance* of the entity. After you identify the entities and attributes, create the *data model,* which serves as a logical design framework for creating your InterBase database. The data model maps entities and attributes to InterBase tables and columns, and is a detailed description of the database—the tables, the columns, the properties of the columns, and the relationships between tables and columns.

The example below shows how the EMPLOYEE entity from the entities/attributes list has been converted to a table.

EMP_NO	LAST_NAME	FIRST_NAME	DEPT_NO	JOB_CODE	PHONE_EXT	SALARY
24	Smith	John	100	Eng	4968	64000
48	Carter	Catherine	900	Sales	4967	72500
36	Smith	Jane	600	Admin	4800	37500

TABLE 2.2 **EMPLOYEE table**

Each row in the EMPLOYEE table represents a single employee. EMP_NO, LAST_NAME, FIRST_NAME, DEPT_NO, JOB_CODE, PHONE_EXT, and SALARY are the columns that represent employee attributes. When the table is populated with data, rows are added to the table, and a *value* is stored at the intersection of each row and column, called a field. In the EMPLOYEE table, "Smith" is a data value that resides in a single field of an employee record.

Determining unique attributes

One of the tasks of database design is to provide a way to uniquely identify each occurrence or instance of an entity so that the system can retrieve any single row in a table. The values specified in the table's primary key distinguish the rows from each other. A PRIMARY KEY or UNIQUE constraint ensures that values entered into the column or set of columns are unique in each row. If you try to insert a value in a PRIMARY KEY or UNIQUE column that already exists in another row of the same column, InterBase prevents the operation and returns an error.

For example, in the EMPLOYEE table, EMP_NO is a unique attribute that can be used to identify each employee in the database, so it is the primary key. When you choose a value as a primary key, determine whether it is inherently unique. For example, no two social security numbers or driver's license numbers are ever the same. Conversely, you should not choose a name column as a unique identifier due to the probability of duplicate values. If no single column has this property of being inherently unique, then define the primary key as a composite of two or more columns which, when taken together, are unique.

A unique key is different from a primary key in that a unique key is not the primary identifier for the row, and is not typically referenced by a foreign key in another table. The main purpose of a unique key is to force a unique value to be entered into the column. You can have only one primary key defined for a table, but any number of unique keys.

Developing a set of rules

When designing a table, you need to develop a set of rules for each table and column that establishes and enforces data integrity. These rules include:

- Specifying the datatype
- Choosing international character sets
- Creating a domain-based column
- Setting default values and NULL status
- Defining integrity constraints and cascading rules
- Defining CHECK constraints

Specifying a datatype

Once you have chosen a given attribute as a column in the table, you can choose a datatype for the attribute. The datatype defines the set of valid data that the column can contain. The datatype also determines which operations can be performed on the data, and defines the disk space requirements for each data item.

The general categories of SQL datatypes include:

- Character datatypes.
- Whole number (integer) datatypes.
- Fixed and floating decimal datatypes.
- A DATE datatype to represent date and time.

▪ A Blob datatype to represent unstructured binary data, such as graphics and digitized voice.

For more information about datatypes supported by InterBase, see Chapter 4, "Specifying Datatypes."

Choosing international character sets

When you create the database, you can specify a default character set. A default character set determines:

▪ What characters can be used in CHAR, VARCHAR, and BLOB text columns.

▪ The default collation order that is used in sorting a column.

The collation order determines the order in which values are sorted. The COLLATE clause of CREATE TABLE allows users to specify a particular collation order for columns defined as CHAR and VARCHAR text datatypes. You must choose a collation order that is supported for the column's given character set. The collation order set at the column level overrides a collation order set at the domain level.

Choosing a default character set is primarily intended for users who are interested in providing a database for international use. For example, the following statement creates a database that uses the ISO8859_1 character set, typically used to support European languages:

```
CREATE DATABASE "employee.gdb"
DEFAULT CHARACTER SET ISO8859_1;
```

You can override the database default character set by creating a different character set for a column when specifying the datatype. The datatype specification for a CHAR, VARCHAR, or BLOB text column definition can include a CHARACTER SET clause to specify a particular character set for a column. If you do not specify a character set, the column assumes the default database character set. If the database default character set is subsequently changed, all columns defined after the change have the new character set, but existing columns are not affected.

If you do not specify a default character set at the time the database is created, the character set defaults to NONE. This means that there is no character set assumption for the columns; data is stored and retrieved just as it was originally entered. You can load any character set into a column defined with NONE, but you cannot load that same data into another column that has been defined with a different character set. No transliteration will be performed between the source and the destination character sets.

For a list of the international character sets and collation orders that InterBase supports, see Chapter 14, "Character Sets and Collation Orders."

Specifying domains

When several tables in the database contain columns with the same definitions and datatypes, you can create domain definitions and store them in the database. Users who create tables can then reference the domain definition to define column attributes locally.

For more information about creating and referencing domains, see Chapter 5, "Working with Domains."

Setting default values and NULL status

You can set an optional default value that is automatically entered into a column when you do not specify an explicit value. Defaults can save data entry time and prevent data entry errors. For example, a possible default for a DATE column could be today's date, or in a Y/N flag column for saving changes, "Y" could be the default. Column-level defaults override defaults set at the domain level.

Assign a NULL status to insert a NULL in the column if the user does not enter a value. Assign NOT NULL to force the user to enter a value, or to define a default value for the column. NOT NULL must be defined for PRIMARY KEY and UNIQUE key columns.

Defining integrity constraints

Integrity constraints are rules that govern column-to-table and table-to-table relationships, and validate data entries. They span all transactions that access the database and are maintained automatically by the system. Integrity constraints can be applied to an entire table or to an individual column. A PRIMARY KEY or UNIQUE constraint guarantees that no two values in a column or set of columns are the same.

Data values that uniquely identify rows (a primary key) in one table can also appear in other tables. A foreign key is a column or set of columns in one table that contain values that match a primary key in another table. The ON UPDATE and ON DELETE referential constraints allow you to specify what happens to the referencing foreign key when the primary key changes or is deleted.

For more information on using PRIMARY KEY and FOREIGN KEY constraints, see Chapter 6, "Working with Tables." For more information on the reasons for using foreign keys, see "Establishing relationships between objects" on page 18.

Defining CHECK constraints

Along with preventing the duplication of values using UNIQUE and PRIMARY KEY constraints, you can specify another type of data entry validation. A CHECK constraint places a condition or requirement on the data values in a column at the time the data is entered. The CHECK constraint enforces a search condition that must be true in order to insert into or update the table or column.

Establishing relationships between objects

The relationship between tables and columns in the database must be defined in the design. For example, how are employees and departments related? An employee can have only one department (a one-to-one relationship), but a department has many employees (a one-to-many relationship). How are projects and employees related? An employee can be working on more than one project, and a project can include several employees (a many-to-many relationship). Each of these different types of relationships has to be modeled in the database.

The relational model represents one-to-many relationships with primary key/foreign key pairings. Refer to the following two tables. A project can include many employees, so to avoid duplication of employee data, the PROJECT table can reference employee information with a foreign key. TEAM_LEADER is a foreign key referencing the primary key, EMP_NO, in the EMPLOYEE table.

PROJ_ID	TEAM_LEAD ER	PROJ_NAME	PROJ_DESC	PRODUCT
DGPII	44	Automap	blob data	hardware
VBASE	47	Video database	blob data	software
HWRII	24	Translator upgrade	blob data	software

TABLE 2.3   PROJECT table

EMP_NO	LAST_NA ME	FIRST_NA ME	DEPT_N O	JOB_CODE	PHONE_EXT	SALARY
24	Smith	John	100	Eng	4968	64000

TABLE 2.4   EMPLOYEE table

EMP_NO	LAST_NAME	FIRST_NAME	DEPT_NO	JOB_CODE	PHONE_EXT	SALARY
48	Carter	Catherine	900	Sales	4967	72500
36	Smith	Jane	600	Admin	4800	37500

TABLE 2.4   EMPLOYEE table (*continued*)

For more information on using PRIMARY KEY and FOREIGN KEY constraints, see Chapter 6, "Working with Tables."

Enforcing referential integrity

The primary reason for defining foreign keys is to ensure that the integrity of the data is maintained when more than one table references the same data—rows in one table must always have corresponding rows in the referencing table. InterBase enforces *referential integrity* in the following ways:

- Before a foreign key can be added, the unique or primary keys that the foreign key references must already be defined.

- If information is changed in one place, it must be changed in every other place that it appears. InterBase does this automatically when you use the ON UPDATE option to the REFERENCES clause when defining the constraints for a table or its columns. You can specify that the foreign key value be changed to match the new primary key value (CASCADE), or that it be set to the column default (SET DEFAULT), or to null (SET NULL). If you choose NO ACTION as the ON UPDATE action, you must manually ensure that the foreign key is updated when the primary key changes. For example, to change a value in the EMP_NO column of the EMPLOYEE table (the primary key), that value must also be updated in the TEAM_LEADER column of the PROJECT table (the foreign key).

- When a row containing a primary key in one table is deleted, the meaning of any rows in another table that contain that value as a foreign key is lost unless appropriate action is taken. InterBase provides the ON DELETE option to the REFERENCES clause of CREATE TABLE and ALTER TABLE so that you can specify whether the foreign key is deleted, set to the column default, or set to null when the primary key is deleted. If you choose NO ACTION as the ON DELETE action, you must manually delete the foreign key before deleting the referenced primary key.

- InterBase also prevents users from adding a value in a column defined as a foreign key that does not reference an existing primary key value. For example, to change a value in the TEAM_LEADER column of the PROJECT table, that value must first be updated in the EMP_NO column of the EMPLOYEE table.

For more information on using PRIMARY KEY and FOREIGN KEY constraints, see Chapter 6, "Working with Tables."

Normalizing the database

After your tables, columns, and keys are defined, look at the design as a whole and analyze it using normalization guidelines in order to find logical errors. As mentioned in the overview, normalization involves breaking down larger tables into smaller ones in order to group data together that is naturally related.

Note A detailed explanation of the normal forms are out of the scope of this document. There are many excellent books on the subject on the market.

When a database is designed using proper normalization methods, data related to other data does not need to be stored in more than one place—if the relationship is properly specified. The advantages of storing the data in one place are:

- The data is easier to update or delete.
- When each data item is stored in one location and accessed by reference, the possibility for error due to the existence of duplicates is reduced.
- Because the data is stored only once, the possibility for introducing inconsistent data is reduced.

In general, the normalization process includes:

- Eliminating repeating groups.
- Removing partially-dependent columns.
- Removing transitively-dependent columns.

An explanation of each step follows.

▶ *Eliminating repeating groups*

When a field in a given row contains more than one value for each occurrence of the primary key, then that group of data items is called a *repeating group*. This is a violation of the first normal form, which does not allow multi-valued attributes.

Refer to the DEPARTMENT table. For any occurrence of a given primary key, if a column can have more than one value, then this set of values is a repeating group. Therefore, the first row, where DEPT_NO = "100," contains a repeating group in the DEPT_LOCATIONS column.

DEPT_NO	DEPARTMENT	HEAD_DEPT	BUDGET	DEPT_LOCATIONS
100	Sales	000	1000000	Monterey, Santa Cruz, Salinas
600	Engineering	120	1100000	San Francisco
900	Finance	000	400000	Monterey

TABLE 2.5   DEPARTMENT table

In the next example, even if you change the attribute to represent only one location, for every occurrence of the primary key "100," all of the columns contain repeating information *except* for DEPT_LOCATION, so this is still a repeating group.

DEPT_NO	DEPARTMENT	HEAD_DEPT	BUDGET	DEPT_LOCATION
100	Sales	000	1000000	Monterey
100	Sales	000	1000000	Santa Cruz
600	Engineering	120	1100000	San Francisco
100	Sales	000	1000000	Salinas

TABLE 2.6   DEPARTMENT table

To normalize this table, we could eliminate the DEPT_LOCATION attribute from the DEPARTMENT table, and create another table called DEPT_LOCATIONS. We could then create a primary key that is a combination of DEPT_NO and DEPT_LOCATION. Now a distinct row exists for each location of the department, and we have eliminated the repeating groups.

DEPT_NO	DEPT_LOCATION
100	Monterey
100	Santa Cruz
600	San Francisco
100	Salinas

TABLE 2.7   DEPT_LOCATIONS table

▶ *Removing partially-dependent columns*

Another important step in the normalization process is to remove any non-key columns that are dependent on only part of a composite key. Such columns are said to have a *partial key dependency*. Non-key columns provide information about the subject, but do not uniquely define it.

For example, suppose you wanted to locate an employee by project, and you created the PROJECT table with a composite primary key of EMP_NO and PROJ_ID.

EMP_NO	PROJ_ID	LAST_NAME	PROJ_NAME	PROJ_DESC	PRODUCT
44	DGPII	Smith	Automap	blob data	hardware
47	VBASE	Jenner	Video database	blob data	software
24	HWRII	Stevens	Translator upgrade	blob data	software

TABLE 2.8   PROJECT table

The problem with this table is that PROJ_NAME, PROJ_DESC, and PRODUCT are attributes of PROJ_ID, but not EMP_NO, and are therefore only partially dependent on the EMP_NO/PROJ_ID primary key. This is also true for LAST_NAME because it is an attribute of EMP_NO, but does not relate to PROJ_ID. To normalize this table, we would remove the EMP_NO and LAST_NAME columns from the PROJECT table, and create another table called EMPLOYEE_PROJECT that has EMP_NO and PROJ_ID as a composite primary key. Now a unique row exists for every project that an employee is assigned to.

▶ *Removing transitively-dependent columns*

The third step in the normalization process is to remove any non-key columns that depend upon other non-key columns. Each non-key column must be a fact about the primary key column. For example, suppose we added TEAM_LEADER_ID and PHONE_EXT to the PROJECT table, and made PROJ_ID the primary key. PHONE_EXT is a fact about TEAM_LEADER_ID, a non-key column, not about PROJ_ID, the primary key column.

PROJ_ID	TEAM_LEADER_ID	PHONE_EXT	PROJ_NAME	PROJ_DESC	PRODUCT
DGPII	44	4929	Automap	blob data	hardware
VBASE	47	4967	Video database	blob data	software
HWRII	24	4668	Translator upgrade	blob data	software

TABLE 2.9   PROJECT table

To normalize this table, we would remove PHONE_EXT, change TEAM_LEADER_ID to TEAM_LEADER, and make TEAM_LEADER a foreign key referencing EMP_NO in the EMPLOYEE table.

PROJ_ID	TEAM_LEADER	PROJ_NAME	PROJ_DESC	PRODUCT
DGPII	44	Automap	blob data	hardware
VBASE	47	Video database	blob data	software
HWRII	24	Translator upgrade	blob data	software

TABLE 2.10   PROJECT table

EMP_NO	LAST_NAME	FIRST_NAME	DEPT_NO	JOB_CODE	PHONE_EXT	SALARY
24	Smith	John	100	Eng	4968	64000
48	Carter	Catherine	900	Sales	4967	72500
36	Smith	Jane	600	Admin	4800	37500

TABLE 2.11   EMPLOYEE table

▶ *When to break the rules*

You should try to correct any normalization violations, or else make a conscious decision to ignore them in the interest of ease of use or performance. Just be sure that you understand the design trade-offs that you are making, and document your reasons. It might take several iterations to reach a design that is a desirable compromise between purity and reality, but this is the heart of the design process.

For example, suppose you always want data about dependents every time you look up an employee, so you decide to include DEP1_NAME, DEP1_BIRTHDATE, and so on for DEP1 through DEP30, in the EMPLOYEE table. Generally speaking, that is terrible design, but the requirements of your application are more important than the abstract purity of your design. In this case, if you wanted to compute the average age of a given employee's dependents, you would have to explicitly add field values together, rather than asking for a simple average. If you wanted to find all employees with a dependent named "Jennifer," you would have to test 30 fields for each employee instead of one. If those are not operations that you intend to perform, then go ahead and break the rules. If the efficiency attracts you less than the simplicity, you might consider defining a view that combines records from employees with records from a separate DEPENDENTS table.

While you are normalizing your data, remember that InterBase offers direct support for array columns, so if your data includes, for example, hourly temperatures for twenty cities for a year, you could define a table with a character column that contains the city name, and a 24 by 366 matrix to hold all of the temperature data for one city for one year. This would result in a table containing 20 rows (one for each city) and two columns, one NAME column and one TEMP_ARRAY column. A normalized version of that record might have 366 rows per city, each of which would hold a city name, a Julian date, and 24 columns to hold the hourly temperatures.

Choosing indexes

Once you have your design, you need to consider what indexes are necessary. The basic trade-off with indexes is that more distinct indexes make retrieval by specific criteria faster, but updating and storage slower. One optimization is to avoid creating several indexes on the same column. For example, if you sometimes retrieve employees based on name, department, badge number, or department name, you should define one index for each of these columns. If a query includes more than one column value to retrieve, InterBase will use more than one index to qualify records. In contrast, defining indexes for every permutation of those three columns will actually slow both retrieval and update operations.

When you are testing your design to find the optimum combination of indexes, remember that the size of the tables affects the retrieval performance significantly. If you expect to have tables with 10,000 to 100,000 records each, do not run tests with only 10 to 100 records.

Another factor that affects index and data retrieval times is page size. By increasing the page size, you can store more records on each page, thus reducing the number of pages used by indexes. If any of your indexes are more than 4 levels deep, you should consider increasing the page size. If indexes on volatile data (data that is regularly deleted and restored, or data that has index key values that change frequently) are less than 3 levels deep, you should consider *reducing* your page size. In general, you should use a page size larger than your largest record, although InterBase's data compression will generally shrink records that contain lots of string data, or lots of numeric values that are 0 or NULL. If your records have those characteristics, you can probably store records on pages which are 20% smaller than the full record size. On the other hand, if your records are not compressible, you should add 5% to the actual record size when comparing it to the page size.

For more information on creating indexes, see Chapter 7, "Working with Indexes."

Increasing cache size

When InterBase reads a page from the database onto disk, it stores that page in its cache, which is a set of buffers that are reserved for holding database pages. Ordinarily, the default cache size of 256 buffers is adequate. If your application includes joins of 5 or more tables, InterBase automatically increases the size of the cache. If your application is well localized, that is, it uses the same small part of the database repeatedly, you might want to consider increasing the cache size so that you never have to release one page from cache to make room for another.

You can use the **gfix** utility to increase the default number of buffers for a specific database using the following command:

```
gfix -buffers n database_name
```

You can also change the default cache size for an entire server either by setting the value of DATABASE_CACHE_PAGES in the configuration file or by changing is on the IB Settings page of the InterBase Server Properties dialog on Windows platforms. This setting is not recommended because it affects all databases on the server and can easily result in overuse of memory or in unusably small caches. It's is better to tune your cache on a per-database basis using **gfix** *-buffers*.

For more information about cache size, see the *Programmer's Guide*. For more information about using **gfix** *-buffers*, see the *Operations Guide*.

Creating a multi-file, distributed database

If you feel that your application performance is limited by disk bandwidth, you might consider creating a multi-file database and distributing it across several disks. Multi-file databases were designed to avoid limiting databases to the size of a disk on systems that do not support multi-disk files.

Planning security

Planning security for a database is important. When implementing the database design, you should answer the following questions:

- Who will have authority to use InterBase?

- Who will have authority to open a particular database?

- Who will have authority to create and access a particular database object within a given database?

For more information about database security, see Chapter 13, "Planning Security."

3

Creating Databases

This chapter describes how to:

- Create a database with CREATE DATABASE.

- Enlarge the database with ALTER DATABASE.

- Delete a database with DROP DATABASE.

- Create an in-sync, online duplication of the database for recovery purposes with CREATE SHADOW.

- Stop database shadowing with DROP SHADOW.

- Increase the size of a shadow.

- Extract metadata from an existing database.

What you should know

Before creating the database, you should know:

- Where to create the database. Users who create databases need to know only the logical names of the available devices in order to allocate database storage. Only the system administrator needs to be concerned about physical storage (disks, disk partitions, operating system files).

- The tables that the database will contain.

- The *record size* of each table, which affects what database page size you choose. A record that is too large to fit on a single page requires more than one page fetch to read or write to it, so access could be faster if you increase the page size.

- How large you expect the database to grow. The *number of records* also affects the page size because the number of pages affects the depth of the index tree. Larger page size means fewer total pages. InterBase operates more efficiently with a shallow index tree.

- The number of users that will be accessing the database.

Creating a database

Create a database in **isql** with an interactive command or with the CREATE DATABASE statement in an **isql** script file. For a description of creating a database interactively with Windows ISQL, see the *Operations Guide*.

Although you *can* create, alter, and drop a database interactively, it is preferable to use a data definition file because it provides a record of the structure of the database. It is easier to modify a source file than it is to start over by retyping interactive SQL statements.

Using a data definition file

A data definition file contains SQL statements, including those for creating, altering, or dropping a database. To issue SQL statements through a data definition file, follow these steps:

1. Use a text editor to write the data definition file.

2. Save the file.

3. Process the file with **isql**.

Use *-input* in command-line **isql** or use **File | Run** in an ISQL Script in Windows ISQL. For more information about command-line **isql** and Windows ISQL, see the *Operations Guide*.

Using CREATE DATABASE

CREATE DATABASE establishes a new database and populates its system tables, or metadata, which are the tables that describe the internal structure of the database. CREATE DATABASE must occur before creating database tables, views, and indexes.

CREATE DATABASE optionally allows you to do the following:

- Specify a user name and a password
- Change the default page size of the new database
- Specify a default character set for the database
- Add secondary files to expand the database

CREATE DATABASE must be the first statement in the data definition file. You cannot create a database directly from the **isql** command line.

IMPORTANT   In DSQL, CREATE DATABASE can be executed only with EXECUTE IMMEDIATE. The database handle and transaction name, if present, must be initialized to zero prior to use.

The syntax for CREATE DATABASE is:

```
CREATE {DATABASE | SCHEMA} "<filespec>"
[USER "username" [PASSWORD "password"]]
[PAGE_SIZE [=] int]
[LENGTH [=] int [PAGE[S]]]
[DEFAULT CHARACTER SET charset]
[<secondary_file>];
<secondary_file> = FILE "<filespec>" [<fileinfo>] [<secondary_file>]
<fileinfo> = LENGTH [=] int [PAGE[S]] | STARTING [AT [PAGE]] int
[<fileinfo>]
```

▸ *Creating a single-file database*

Although there are many optional parameters, CREATE DATABASE requires only one parameter, *filespec*, which is the new database file specification. The file specification contains the device name, path name, and database name.

By default, a database is created as a single file, called the *primary file*. The following example creates a single-file database, named *employee.gdb*, in the current directory.

```
CREATE DATABASE "employee.gdb";
```

For more information about file naming conventions, see the *Operations Guide*.

SPECIFYING FILE SIZE FOR A SINGLE-FILE DATABASE

You can optionally specify a file length, in pages, for the primary file. For example, the following statement creates a database that is stored in one 10,000-page- long file:

```
CREATE DATABASE "employee.gdb" LENGTH 10000;
```

If the database grows larger than the specified file length, InterBase extends the primary file beyond the LENGTH limit until the disk space runs out. To avoid this, you can store a database in more than one file, called a *secondary file*.

Note Use LENGTH for the primary file only if defining a secondary file in the same statement.

▶ *Creating a multi-file database*

A multi-file database consists of a *primary file* and one or more *secondary files*. You can create one or more secondary files to be used for overflow purposes only; you cannot specify what information goes into each file because InterBase handles this automatically. Each secondary file is typically assigned to a different disk than that of the main database. When the primary file fills up, InterBase allocates one of the secondary files that was created. When that secondary file fills up, another secondary file is allocated, and so on, until all of the secondary file allocations run out.

IMPORTANT   Whenever possible, the database should be created locally; create the database on the same machine where you are running **isql**. If the database is created locally, secondary file names can include a full file specification, including both host or node names, and a directory path to the location of the database file. If the database is created on a remote server, secondary file specifications cannot include a node name, as all secondary files must reside on the same node.

SPECIFYING FILE SIZE OF A SECONDARY FILE

Unlike primary files, when you define a secondary file, you *must* declare either a file length in pages, or a starting page number. The LENGTH parameter specifies a database file size in pages.

If you choose to describe page ranges in terms of length, list the files in the order in which they should be filled. The following example creates a database that is stored in four 10,000-page files, Starting with page 10,001, the files are filled in the order *employee.gdb, employee.gd1, employee.gd2*, and *employee.gd3*.

```
CREATE DATABASE "employee.gdb"
   FILE "employee.gd1" STARTING AT PAGE 10001
      LENGTH 10000 PAGES
   FILE "employee.gd2"
      LENGTH 10000 PAGES
   FILE "employee.gd3";
      LENGTH 10000 PAGES
```

Note Because file-naming conventions are platform-specific, for the sake of simplicity, none of the examples provided include the device and path name portions of the file specification.

When the last secondary file fills up, InterBase automatically extends the file beyond the LENGTH limit until its disk space runs out. You can either specify secondary files when the database is defined, or add them later, as they become necessary, using ALTER DATABASE. Defining secondary files when a database is created immediately reserves disk space for the database.

SPECIFYING THE STARTING PAGE NUMBER OF A SECONDARY FILE

If you do not declare a length for a secondary file, then you must specify a starting page number. STARTING AT PAGE specifies the beginning page number for a secondary file.

The primary file specification in a multi-file database does not need to include a length, but secondary file specifications must then include a starting page number. You can specify a combination of length and starting page numbers for secondary files.

InterBase overrides a secondary file length that is inconsistent with the starting page number. In the next example, the primary file is 10,000 pages long, but the first secondary file starts at page 5,000:

```
CREATE DATABASE "employee.gdb" LENGTH 10000
   FILE "employee.gd1" STARTING AT PAGE 5000
      LENGTH 10000 PAGES
   FILE "employee.gd2"
      LENGTH 10000 PAGES
   FILE "employee.gd3";
```

InterBase generates a primary file that is 10,000 pages long, starting the first secondary file at page 10,001.

▶ *Specifying user name and password*

If provided, the user name and password are checked against valid user name and password combinations in the security database on the server where the database will reside. Passwords are restricted to 8 characters in length.

IMPORTANT   Windows client applications must create their databases on a remote server. For these remote connections, the user name and password are *not* optional. Windows clients *must* provide the USER and PASSWORD options with CREATE DATABASE before connecting to a remote server.

The following statement creates a database with a user name and password:

```
CREATE DATABASE "employee.gdb" USER "SALES" PASSWORD "mycode";
```

▶ *Specifying database page size*

You can optionally override the default page size of 1024 bytes for database pages by specifying a different PAGE_SIZE. PAGE_SIZE can be 1024, 2048, 4096, or 8192. The next statement creates a single-file database with a page size of 2048 bytes:

```
CREATE DATABASE "employee.gdb" PAGE_SIZE 2048;
```

WHEN TO INCREASE PAGE SIZE

Increasing page size can improve performance for several reasons:

▪ Indexes work faster because the depth of the index is kept to a minimum.

▪ Keeping large rows on a single page is more efficient. (A row that is too large to fit on a single page requires more than one page fetch to read or write to it.)

▪ BLOB data is stored and retrieved more efficiently when it fits on a single page. If an application typically stores large BLOB columns (between 1K and 2K), a page size of 2048 bytes is preferable to the default (1024).

If most transactions involve only a few rows of data, a smaller page size might be appropriate, since less data needs to be passed back and forth and less memory is used by the disk cache.

CHANGING PAGE SIZE FOR AN EXISTING DATABASE

To change a page size of an *existing* database, follow these steps:

1. Back up the database.

2. Restore the database using the PAGE_SIZE option to specify a new page size.

For more detailed information on backing up the database, see the *Operations Guide*.

▶ *Specifying the default character set*

DEFAULT CHARACTER SET allows you to optionally set the default character set for the database. The character set determines:

▪ What characters can be used in CHAR, VARCHAR, and BLOB text columns.

▪ The default collation order that is used in sorting a column.

Choosing a default character set is useful for all databases, even those where international use is not an issue. Choice of character set determines if transliteration among character sets is possible. For example, the following statement creates a database that uses the ISO8859_1 character set, typically used in Europe to support European languages:

```
CREATE DATABASE "employee.gdb"
DEFAULT CHARACTER SET "ISO8859_1";
```

For a list of the international character sets and collation orders that InterBase supports, see Chapter 14, "Character Sets and Collation Orders."

USING CHARACTER SET NONE

If you do not specify a default character set, the character set defaults to NONE. Using CHARACTER SET NONE means that there is no character set assumption for columns; data is stored and retrieved just as you originally entered it. You can load any character set into a column defined with NONE, but you cannot load that same data into another column that has been defined with a different character set. No transliteration will be performed between the source and destination character sets, so in most cases, errors will occur during the attempted assignment.

For example:

```
CREATE TABLE MYDATA (PART_NUMBER CHARACTER(30) CHARACTER SET NONE);
SET NAMES LATIN1;
INSERT INTO MYDATA (PART_NUMBER) VALUES ("à");
SET NAMES DOS437;
SELECT * FROM MYDATA;
```

The data ("à") is returned just as it was entered, without the à being transliterated from the input character (LATIN1) to the output character (DOS437). If the column had been set to anything other than NONE, the transliteration would have occurred.

Altering a database

Use ALTER DATABASE to add one or more secondary files to an existing database. Secondary files are useful for controlling the growth and location of a database. They permit database files to be spread across storage devices, but must remain on the same node as the primary database file. For more information on secondary files, see "Creating a multi-file database" on page 30.

A database can be altered by its creator, the SYSDBA user, and any users with operating system root privileges.

ALTER DATABASE requires exclusive access to the database. For more information about exclusive database access, see the *Operations Guide*.

The syntax for ALTER DATABASE is:

```
ALTER {DATABASE | SCHEMA}
ADD <add_clause>;
<add_clause> =
FILE "<filespec>" <fileinfo> [<add_clause>]
```

```
<fileinfo> = LENGTH [=] int [PAGE[S]] | STARTING [AT [PAGE]] int
[<fileinfo>]
```

You must specify a range of pages for each file either by providing the number of pages in each file, or by providing the starting page number for the file. The following statement adds two secondary files to the currently connected database:

```
ALTER DATABASE
    ADD FILE "employee.gd1"
    STARTING AT PAGE 10001
        LENGTH 10000
    ADD FILE "employee.gd2"
        LENGTH 10000;
```

Dropping a database

DROP DATABASE is the command that deletes the database currently connected to, including any associated shadow and log files. Dropping a database deletes any data it contains. A database can be dropped by its creator, the SYSDBA user, and any users with operating system root privileges.

The following statement deletes the current database:

```
DROP DATABASE;
```

Creating a database shadow

InterBase lets you recover a database in case of disk failure, network failure, or accidental deletion of the database. The recovery method is called *shadowing*. This section describes the various tasks involved in shadowing, as well as the advantages and limitations of shadowing. The main tasks in setting up and maintaining shadowing are as follows:

- **CREATING A SHADOW** Shadowing begins with the creation of a *shadow*. A shadow is an identical physical copy of a database. When a shadow is defined for a database, changes to the database are written simultaneously to its shadow. In this way, the shadow always reflects the current state of the database. For information about the different ways to define a shadow, see "Using CREATE SHADOW" on page 36.

- **DELETING A SHADOW** If shadowing is no longer desired, the shadow can be deleted. For more information about deleting a shadow, see "Dropping a shadow" on page 39.

- **ADDING FILES TO A SHADOW** A shadow can consist of more than one file. As shadows grow in size, files can be added to accommodate the increased space requirements.

Advantages of shadowing

Shadowing offers several advantages:

- Recovery is quick: Activating a shadow makes it available immediately.

- Creating a shadow does not require exclusive access to the database.

- You can control the allocation of disk space. A shadow can span multiple files on multiple disks.

- Shadowing does not use a separate process. The database process handles writing to the shadow.

- Shadowing runs behind the scenes and needs little or no maintenance.

Limitations of shadowing

Shadowing has the following limitations:

- Shadowing is useful only for recovery from hardware failures or accidental deletion of the database. User errors or software failures that corrupt the database are duplicated in the shadow.

- Recovery to a specific point in time is not possible. When a shadow is activated, it takes over as a duplicate of the database. Shadowing is an "all or nothing" recovery method.

- Shadowing can occur only to a local disk. Shadowing to a NFS filesystem, mapped drive, tape, or other media is not supported.

Before creating a shadow

Before creating a shadow, consider the following questions:

- Where will the shadow reside?

- A shadow should be created on a different disk from where the main database resides. Because shadowing is intended as a recovery mechanism in case of disk failure, maintaining a database and its shadow on the same disk defeats the purpose of shadowing.

- How will the shadow be distributed?

- A shadow can be created as a single disk file called a shadow file or as multiple files called a shadow set. To improve space allocation and disk I/O, each file in a shadow set can be placed on a different disk.

- If something happens that makes a shadow unavailable, should users be allowed to access the database?

- If a shadow becomes unavailable, InterBase can either deny user access until shadowing is resumed, or InterBase can allow access even though database changes are not being shadowed. Depending on which database behavior is desired, the database administrator (DBA) creates a shadow either in auto mode or in manual mode. For more information about these modes, see "Auto mode and manual mode" on page 38.

- If a shadow takes over for a database, should a new shadow be automatically created?

- To ensure that a new shadow is automatically created, create a conditional shadow. For more information, see "Conditional shadows" on page 39.

Using CREATE SHADOW

Use the CREATE SHADOW statement to create a database shadow. Because this does not require exclusive access, it can be done without affecting other users. A shadow can be created using a combination of the following options:

- Single-file or multi-file shadows

- Auto or manual shadows

- Conditional shadows

These options are not mutually exclusive. For example, you can create a single-file, manual, conditional shadow.

The syntax of CREATE SHADOW is:

```
CREATE SHADOW set_num [AUTO | MANUAL] [CONDITIONAL]
"<filespec>" [LENGTH [=] int [PAGE[S]]]
[<secondary_file>];
<secondary_file> = FILE "<filespec>" [<fileinfo>] [<secondary_file>]
<fileinfo> = LENGTH [=] int [PAGE[S]] | STARTING [AT [PAGE]] int
[<fileinfo>]
```

▶ *Creating a single-file shadow*

To create a single-file shadow for the database *employee.gdb,* enter:

```
CREATE SHADOW 1 "employee.shd";
```

The shadow is associated with the currently connected database, *employee.gdb.* The name of the shadow file is *employee.shd*, and it is identified by a shadow set number, 1, in this example. The shadow set number tells InterBase that all of the shadow files listed are grouped together under this identifier.

To verify that the shadow has been created, enter the **isql** command SHOW DATABASE:

```
SHOW DATABASE;
Database: employee.gdb
Shadow 1: "/usr/interbase/employee.shd" auto
PAGE_SIZE 1024
Number of DB pages allocated = 392
Sweep interval = 20000
```

The page size of the shadow is the same as that of the database.

▶ *Creating a multi-file shadow*

If the size of a database exceeds the space available on one disk, create a multi-file shadow and spread the files over several disks. To create a multi-file shadow, specify the name and size of each file in the shadow set. File specifications are platform-specific. The following examples illustrate the creation of a multi-file shadow on a Unix platform. The shadow files are created on the A, B, and C drives of the *IB_bckup* node:

```
CREATE SHADOW 1 "IB_bckup:/employee.shd" LENGTH 1000
FILE "IB_bckup:/emp1.shd" LENGTH 2000
FILE "IB_bckup:/emp2.shd" LENGTH 2000;
```

This example creates a shadow set consisting of three files. The primary file, *employee.shd*, is 1,000 database pages in length. The secondary files, each identified by the FILE keyword, are each 2,000 database pages long.

Instead of specifying the page length of secondary files, you can specify their starting pages. The previous example could be entered as follows:

```
CREATE SHADOW 1 "IB_bckup:/employee.shd" LENGTH 1000
FILE "IB_bckup:/emp1.shd" STARTING AT 1000
FILE "IB_bckup:/emp2.shd" STARTING AT 3000;
```

In either case, you can use SHOW DATABASE to verify the file names, page lengths, and starting pages for the shadow just created:

```
SHOW DATABASE;
Database: employee.gdb
Shadow 1: IB_bckup:/employee.shd auto length 1000
file IB_bckup:/emp1.shd length 2000 starting 1000
file IB_bckup:/emp2.shd length 2000 starting 3000
PAGE_SIZE 1024
Number of DB pages allocated = 392
Sweep interval = 20000
```

Note The page length allocated for secondary shadow files need not correspond to the page length of the database's secondary files. As the database grows and its first shadow file becomes full, updates to the database automatically overflow into the next shadow file.

▶ *Auto mode and manual mode*

A shadow can become unavailable for the same reasons a database becomes unavailable: disk failure, network failure, or accidental deletion. If a shadow becomes unavailable, and it was created in auto mode, database operations continue automatically without shadowing. If a shadow becomes unavailable, and it was created in manual mode, further access to the database is denied until the database administrator intervenes. The benefits of auto mode and manual mode are compared in the following table:

Mode	Advantage	Disadvantage
Auto	Database operation is uninterrupted.	Creates a temporary period when the database is not shadowed.
		The DBA might be unaware that the database is operating without a shadow.
Manual	Prevents the database from running unintentionally without a shadow.	Database operation is halted until the problem is fixed. Needs intervention of the DBA.

TABLE 3.1   Auto vs. manual shadows

AUTO MODE

The AUTO keyword directs the CREATE SHADOW statement to create a shadow in auto mode:

```
CREATE SHADOW 1 AUTO "employee.shd";
```

Auto mode is the default, so omitting the AUTO keyword achieves the same result.

In auto mode, database operation is uninterrupted even though there is no shadow. To resume shadowing, it might be necessary to create a new shadow. If the original shadow was created as a conditional shadow, a new shadow is automatically created. For more information about conditional shadows, see "Conditional shadows" on page 39.

MANUAL MODE

The MANUAL keyword directs the CREATE SHADOW statement to create a shadow in manual mode:

```
CREATE SHADOW 1 MANUAL "employee.shd";
```

Manual mode is useful when continuous shadowing is more important than continuous operation of the database. When a manual-mode shadow becomes unavailable, further connections to the database are prevented. To allow database connections again, the database administrator must remove the old shadow file. After deleting the references, a new shadow can be created if shadowing needs to resume.

▶ *Conditional shadows*

A shadow can be defined so that if it replaces a database, a new shadow will be automatically created, allowing shadowing to continue uninterrupted. A shadow defined with this behavior is called a *conditional shadow.*

To create a conditional shadow, specify the CONDITIONAL keyword with the CREATE SHADOW statement. For example:

```
CREATE SHADOW 3 CONDITIONAL "employee.shd";
```

Creating a conditional file directs InterBase to automatically create a new shadow. This happens in either of two cases:

- The database or one of its shadow files becomes unavailable.
- The shadow takes over for the database due to hardware failure.

Dropping a shadow

To stop shadowing, use the shadow number as an argument to the DROP SHADOW statement. DROP SHADOW deletes shadow references from a database's metadata, as well as the physical files on disk.

A shadow can be dropped by its creator, the SYSDBA user, or any user with operating system root privileges.

DROP SHADOW syntax

```
DROP SHADOW set_num;
```

The following example drops all of the files associated with the shadow set number 1:

```
DROP SHADOW 1;
```

If you need to look up the shadow number, use the **isql** command SHOW DATABASE.

```
SHOW DATABASE;
Database: employee.gdb
Shadow 1: "employee.shd" auto
PAGE_SIZE 1024
Number of DB pages allocated = 392
Sweep interval = 20000
```

Expanding the size of a shadow

If a database is expected to increase in size, or if the database is already larger than the space available for a shadow on one disk, you might need to expand the size of the shadow. To do this, drop the current shadow and create a new one containing additional files. To add a shadow file, first use DROP SHADOW to delete the existing shadow, then use CREATE SHADOW to recreate it with the desired number of secondary files.

The page length allocated for secondary shadow files need not correspond to the page length of the database's secondary files. As the database grows and its first shadow file becomes full, updates to the database automatically overflow into the next shadow file.

Using isql to extract data definitions

isql enables you to extract data definition statements from a database and store them in an output file. All keywords and objects are extracted into the file in uppercase.

The output file enables users to:

- Examine the current state of a database's system tables before planning alterations. This is especially useful when the database has changed significantly since its creation.
- Create a database with schema definitions that are identical to the extracted database.
- Make changes to the database, or create a new database source file with a text editor.

Extracting an InterBase 4.0 database

You can use Windows ISQL on a Windows Client PC to extract data definition statements. On some servers, you can also use command-line **isql** on the server platform to extract data definition statements. For more information on using Windows ISQL and command-line **isql**, see the *Operations Guide*.

Extracting a 3.x database

To extract metadata from a 3.*x* database, use command-line **isql** on the server. Use the **-a** switch instead of *-x*. The difference between the *-x* option and the *-a* option is that the **-x** option extracts metadata for SQL objects only, and the *-a* option extracts all DDL for the named database. The syntax can differ depending upon operating system requirements.

The following command extracts the metadata from the *employee.gdb* database into the file, *newdb.sql*:

```
isql -a employee.gdb -o newdb.sql
```

For more information on using command-line **isql**, see the *Operations Guide*.

4

Specifying Datatypes

This chapter describes the following:

- All of the datatypes that are supported by InterBase, and the allowable operations on each type
- Where to specify the datatype, and which data definition statements reference or define the datatype
- How to specify a default character set
- How to create each datatype, including BLOB data
- How to create arrays of datatypes
- How to perform datatype conversions

About InterBase datatypes

When creating a new column in an InterBase table, the primary attribute that you must define is the *datatype*, which establishes the set of valid data that the column can contain. Only values that can be represented by that datatype are allowed. Besides establishing the set of valid data that a column can contain, the datatype defines the kinds of operations that you can perform on the data. For example, numbers in INTEGER columns can be manipulated with arithmetic operations, while CHARACTER columns cannot.

The datatype also defines how much space each data item occupies on the disk. Choosing an optimum size for the data value is an important consideration when disk space is limited, especially if a table is very large.

InterBase supports the following datatypes:

- INTEGER and SMALLINT
- FLOAT and DOUBLE PRECISION
- NUMERIC and DECIMAL
- DATE
- CHARACTER and VARYING CHARACTER
- BLOB

InterBase provides the binary large object (BLOB) datatype to store data that cannot easily be stored in one of the standard SQL datatypes. A BLOB is used to store very large data objects of indeterminate and variable size, such as bitmapped graphics images, vector drawings, sound files, video segments, chapter or book-length documents, or any other kind of multimedia information.

InterBase also supports arrays of most datatypes. An *array* is a matrix of individual items composed of any single InterBase datatype (except BLOB). An array can have from 1 to 16 dimensions. An array can be handled as a single entity, or manipulated item-by-item.

A DATE datatype is supported that includes information about year, month, day of the month, and time. The DATE datatype is stored as two long integers, and requires conversion to and from InterBase when entered or manipulated in a host-language program.

The following table describes the datatypes supported by InterBase:

Name	Size	Range/Precision	Description
BLOB	Variable	None. BLOB segment size is limited to 64K.	Binary large object. Stores large data, such as graphics, text, and digitized voice. Basic structural unit: segment. BLOB subtype describes BLOB contents.
CHAR(*n*)	*n* characters	1 to 32767 bytes.	Fixed length CHAR or text string type.
		Character set character size determines the maximum number of characters that can fit in 32K.	Alternate keyword: CHARACTER.
DATE	64 bits	1 Jan 100 to 11 Dec 5941.	Also included time information.

TABLE 4.1 Datatypes supported by InterBase

Name	Size	Range/Precision	Description
DECIMAL (*precision, scale*)	variable	*precision* = 1 to 15. Specifies at least *precision* digits of precision to store.	Number with a decimal point *scale* digits from the right. For example, DECIMAL(10, 3) holds numbers accurately in the following format:
		scale = 1 to 15. Specifies number of decimal places for storage. Must be less than or equal to *precision*.	ppppppp.sss
DOUBLE PRECISION	64 bits[a]	1.7×10^{-308} to 1.7×10^{308}.	Scientific: 15 digits of precision.
FLOAT	32 bits	3.4×10^{-38} to 3.4×10^{38}	Single precision: 7 digits of precision.
INTEGER	32 bits	−2,147,483,648 to 2,147,483,647.	Signed long (longword).
NUMERIC (*precision, scale*)	variable	*precision* = 1 to 15. Specifies exactly *precision* digits of precision to store.	Number with a decimal point *scale* digits from the right. For example, NUMERIC(10,3) holds numbers accurately in the following format:
		scale = 1 to 15. Specifies number of decimal places for storage. Must be less than or equal to *precision*.	ppppppp.sss
SMALLINT	16 bits	−32768 to 32767.	Signed short (word).
VARCHAR(*n*)	*n* characters	1 to 32765 bytes.	Variable length CHAR or text string type.
		Character set character size determines the maximum number of characters that can fit in 32K.	Alternate keywords: CHAR VARYING, CHARACTER VARYING

TABLE 4.1   Datatypes supported by InterBase (*continued*)

a.   Actual size of DOUBLE is platform-dependent. Most platforms support the 64-bit size.

Where to specify datatypes

A datatype is assigned to a column in the following situations:

- Creating a table using CREATE TABLE.
- Creating a global column template using CREATE DOMAIN.

▪ Adding a new column to a table using ALTER TABLE.

The syntax for specifying the datatype with these statements is provided here for reference.

```
<datatype> = {
{SMALLINT | INTEGER | FLOAT | DOUBLE PRECISION} [<array_dim>]
| {DECIMAL | NUMERIC} [(precision [, scale])] [<array_dim>]
| DATE [<array_dim>]
| {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR}
      [(int)] [<array_dim>] [CHARACTER SET charname]
| {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
      [VARYING] [(int)] [<array_dim>]
| BLOB [SUB_TYPE {int | subtype_name}] [SEGMENT SIZE int]
      [CHARACTER SET charname]
| BLOB [(seglen [, subtype])]
}
```

For more information on how to create a datatype using CREATE TABLE and ALTER TABLE, see Chapter 6, "Working with Tables." For more information on using CREATE DOMAIN to define datatypes, see Chapter 5, "Working with Domains."

Defining numeric datatypes

The numeric datatypes that InterBase supports include integer numbers of various sizes (INTEGER and SMALLINT), floating-point numbers with variable precision (FLOAT, DOUBLE PRECISION), and formatted, fixed-decimal numbers (DECIMAL, NUMERIC).

Integer datatypes

Integers are whole numbers. InterBase supports two integer datatypes: SMALLINT and INTEGER. SMALLINT is a signed short integer with a range from –32,768 to 32,767. INTEGER is a signed long integer with a range from –2,147,483,648 to 2,147,483,647.

The next two statements create domains with the SMALLINT and INTEGER datatypes:

```
CREATE DOMAIN EMPNO
AS SMALLINT;
CREATE DOMAIN CUSTNO
AS INTEGER
CHECK (VALUE > 99999);
```

You can perform the following operations on the integer datatypes:

- Comparisons using the standard relational operators (=, <, >, >=, <=). Other operators such as CONTAINING, STARTING WITH, and LIKE perform string comparisons on numeric values.

- Arithmetic operations. The standard arithmetic operators determine the sum, difference, product, or dividend of two or more integers.

- Conversions. When performing arithmetic operations that involve mixed datatypes, InterBase automatically converts between INTEGER, FLOAT, and CHAR datatypes. For operations that involve comparisons of numeric data with other datatypes, InterBase first converts the data to a numeric type, then performs the arithmetic operation or comparison.

- Sorts. By default, a query retrieves rows in the exact order that it finds them in the table, which is likely to be unordered. You can sort rows using the ORDER BY clause of a SELECT statement in descending or ascending order.

Fixed-decimal datatypes

InterBase supports two SQL datatypes, NUMERIC, and DECIMAL, for handling numeric data with a fixed decimal point, such as monetary values. You can specify optional precision and scale factors for both datatypes. *Precision* is the maximum number of total digits, both significant and fractional, that can appear in a column of these datatypes. *Scale* is the number of digits to the right of the decimal point that comprise the fractional portion of the number. The allowable range for both precision and scale is from 1 to a maximum of 15, and scale must be less than or equal to precision.

The syntax for NUMERIC and DECIMAL is as follows:

```
NUMERIC[(precision [, scale])]
DECIMAL[(precision [, scale])]
```

You can specify NUMERIC and DECIMAL datatypes without precision or scale, with precision only, or with both precision and scale. When you specify a NUMERIC datatype with both precision and scale, the exact number of digits that you specified in precision and scale are stored. For example,

```
NUMERIC(4,2)
```

declares that a column of this type always holds numbers with up to two significant digits, with exactly two digits to the right of the decimal point: *pp.ss.*

When you specify a DECIMAL datatype with both precision and scale, the number of total digits stored is *at least* as many as you specified in precision, and the exact number of fractional digits that you specified in scale. For example,

```
DECIMAL(4,2)
```

declares that a column of this type must be capable of holding at least two, but possibly more significant digits, and exactly two digits to the right of the decimal point: *pp.ss*.

▶ *How InterBase stores fixed-decimal datatypes*

When you create a domain or column with a NUMERIC or DECIMAL datatype, InterBase determines which datatype to use for internal storage based on the precision and scale that you specify. NUMERIC and DECIMAL datatypes store numbers in three ways:

- Defined without precision or scale—always stored as INTEGER.

- Defined with precision, but not scale—depending upon the precision specified, stored as SMALLINT, INTEGER, or DOUBLE PRECISION.

- Defined with both precision and scale—depending upon the precision specified, stored as SMALLINT, INTEGER, or DOUBLE PRECISION.

The following table summarizes how InterBase stores NUMERIC and DECIMAL datatypes based on precision and scale:

Datatype specified as…	Datatype stored as…
NUMERIC	INTEGER
NUMERIC(4)	SMALLINT
NUMERIC(9)	INTEGER
NUMERIC(10)	DOUBLE PRECISION
NUMERIC(4,2)	SMALLINT
NUMERIC(9,3)	INTEGER
NUMERIC(10,4)	DOUBLE PRECISION
DECIMAL	INTEGER
DECIMAL(4)	INTEGER
DECIMAL(9)	INTEGER
DECIMAL(10)	DOUBLE PRECISION
DECIMAL(4,2)	INTEGER
DECIMAL(9,3)	INTEGER
DECIMAL(10,4)	DOUBLE PRECISION

▶ *Specifying* NUMERIC *and* DECIMAL *without scale*

IMPORTANT   For a NUMERIC datatype, if a precision of less than 5 is specified without scale, InterBase stores the datatype as a SMALLINT. If the precision is less than 10, InterBase stores the type as an INTEGER. For precisions of 10 or greater, the datatype is stored as DOUBLE PRECISION. See the previous table for the exact specifications.

Therefore, when you declare NUMERIC and DECIMAL datatypes with a precision of 10 or greater, *fractional numbers can be stored without specifying a scale.*

For example, in **isql**, if you specify "NUMERIC(10)", and insert a 13-digit number "2555555.256789," the number is stored exactly as specified, with 13 digits of precision and six digits to the right of the decimal. Conversely, if you format the column as NUMERIC(9), and insert the same 13-digit number "2555555.256789," InterBase truncates the fraction and stores the number as an INTEGER, "2555555."

Similarly, for a DECIMAL datatype, if a precision of less than 10 is specified without scale, InterBase stores the datatype as INTEGER; otherwise, it stores the datatype as DOUBLE PRECISION.

IMPORTANT   When you format the column as NUMERIC or DECIMAL with a precision of 10 or greater without scale, you *lose the ability to control both scale and precision.*

Using the same NUMERIC(10) example, when you insert the 13-digit number "2555555.256789," the number is stored exactly as specified, with *13* digits of precision and *6* digits to the right of the decimal. If you insert an 11-digit number "255555.25678," the number is also stored exactly as specified with *11* digits of precision, and *5* fractional digits. You might expect that the precision would always be 10 because you explicitly specified 10, but it also varies depending upon precision of the inserted data.

TIP   If you want to store fixed-decimal numbers such as monetary values, do not declare NUMERIC or DECIMAL with a precision of 10 or greater without specifying scale. In addition, if you need to control the precision for decimal data, you must specify scale.

▶ *Specifying* NUMERIC *and* DECIMAL *with scale and precision*

When a NUMERIC or DECIMAL datatype declaration includes both precision and scale, values containing a fractional portion can be stored, and you can control the number of fractional digits. InterBase stores such values internally as SMALLINT, INTEGER, or DOUBLE PRECISION data, depending on the precision specified. How can a number with a fractional portion be stored as an integer value? For all SMALLINT and INTEGER data entered, InterBase stores:

- A scale factor, a negative number indicating how many decimal places are contained in the number, based on the power of 10. A –1 scale factor indicates a fractional portion of tenths; a –2 scale factor indicates a fractional portion of hundredths. You do not need to include the sign; it is negative by default.

- For example, when you specify NUMERIC(4,2), InterBase stores the number internally as a SMALLINT. If you insert the number "25.253," it is stored as a decimal "25.25," with 4 digits of precision, and a scale of 2.

- The number is divided by 10 to the power of "scale" (number/10^{scale}) to produce a number without a fractional portion.

▸ *Specifying datatypes using embedded applications*

DSQL applications such as **isql** can correct for the scale factor for SMALLINT and INTEGER datatypes by examining the XSQLVAR *sqlscale* field and dividing to produce the correct value.

IMPORTANT Embedded applications cannot use or recognize small precision NUMERIC or DECIMAL datatypes with fractional portions when they are stored as SMALLINT or INTEGER types. To avoid this problem, create all NUMERIC and DECIMAL datatypes that are to be accessed from embedded applications with a precision of 10 or more, which forces them to be stored as DOUBLE PRECISION. Again, remember to specify a scale if you want to control the precision and scale.

Both SQL and DSQL applications handle NUMERIC and DECIMAL types stored as DOUBLE PRECISION without problem.

Floating-point datatypes

InterBase provides two floating-point datatypes, FLOAT and DOUBLE PRECISION; the only difference is their size. FLOAT specifies a single-precision, 32-bit datatype with a precision of approximately 7 decimal digits. DOUBLE PRECISION specifies a double-precision, 64-bit datatype with a precision of approximately 15 decimal digits.

The precision of FLOAT and DOUBLE PRECISION is fixed by their size, but the scale is not, and you cannot control the formatting of the scale. With floating numeric datatypes, the placement of the decimal point can vary; the position of the decimal is allowed to "float." For example, in the same column, one value could be stored as "25.33333," and another could be stored as "25.333."

Use floating-point numbers when you expect the placement of the decimal point to vary, and for applications where the data values have a very wide range, such as in scientific calculations.

If the value stored is outside of the range of the precision of the floating-point number, then it is stored only approximately, with its least-significant digits treated as zeros. For example, if the type is FLOAT, you are limited to 7 digits of precision. If you insert a 10-digit number "25.33333312" into the column, it is stored as "25.33333."

The next statement creates a column, PERCENT_CHANGE, using a DOUBLE PRECISION type:

```
CREATE TABLE SALARY_HISTORY
( . . .
PERCENT_CHANGE DOUBLE PRECISION
      DEFAULT 0
      NOT NULL
      CHECK (PERCENT_CHANGE BETWEEN -50 AND 50),
. . .);
```

You can perform the following operations on FLOAT and DOUBLE PRECISION datatypes:

- Comparisons using the standard relational operators (=, <, >, >=, <=). Other operators such as CONTAINING, STARTING WITH, and LIKE perform string comparisons on the integer portion of floating data.

- Arithmetic operations. The standard arithmetic operators determine the sum, difference, product, or dividend of two or more integers.

- Conversions. When performing arithmetic operations that involve mixed datatypes, InterBase automatically converts between INTEGER, FLOAT, and CHAR datatypes. For operations that involve comparisons of numeric data with other datatypes, such as CHARACTER and INTEGER, InterBase first converts the data to a numeric type, then compares them numerically.

- Sorts. By default, a query retrieves rows in the exact order that it finds them in the table, which is likely to be unordered. Sort rows using the ORDER BY clause of a SELECT statement in descending or ascending order.

The following CREATE TABLE statement provides an example of how the different numeric types can be used: an INTEGER for the total number of orders, a fixed DECIMAL for the dollar value of total sales, and a FLOAT for a discount rate applied to the sale.

```
CREATE TABLE SALES
( . . .
   QTY_ORDERED INTEGER
      DEFAULT 1
         CHECK (QTY_ORDERED >= 1),
TOTAL_VALUE DECIMAL (9,2)
```

```
           CHECK (TOTAL_VALUE >= 0),
DISCOUNT FLOAT
       DEFAULT 0
          CHECK (DISCOUNT >= 0 AND DISCOUNT <= 1));
```

The DATE datatype

InterBase supports a DATE datatype that stores dates as two 32-bit longwords. Valid dates are from 1-JAN-100 to 11-DEC-5941. The following statement creates DATE columns in the SALES table:

```
CREATE TABLE SALES
( . . .
ORDER_DATE DATE
       DEFAULT "now"
       NOT NULL,
   SHIP_DATE DATE
       CHECK (SHIP_DATE >= ORDER_DATE OR SHIP_DATE IS NULL),
. . .);
```

Note In the previous example, "now" returns the system date and time.

Most languages do not support the DATE datatype. Instead they express dates as strings or structures. The DATE datatype requires conversion to and from InterBase when entered or manipulated in a host-language program. There are two ways to use the DATE datatype:

1. Create a string in a format that InterBase understands (for example, "1-JAN-1994"). When you insert the date into a DATE column, InterBase automatically converts the text into the internal DATE format.

2. Use the call interface routines provided by InterBase to do the conversion. **isc_decode_date()** converts from the InterBase internal DATE format to the C time structure. **isc_encode_date()** converts from the C time structure to the internal InterBase DATE format.

Note The string conversion described in item #1 does not work in the other direction. To read a date in an InterBase format and convert it to a C date variable, you must call **isc_decode_date()**.

For more information about how to convert date datatypes in C, and how to use the **cast()** function for type conversion using SELECT statements, see the *Programmer's Guide*.

Character datatypes

InterBase supports four character string datatypes:

1. A fixed-length character datatype, called CHAR(*n*) or CHARACTER(*n*), where *n* is the *exact* number of characters stored.

2. A variable-length character type, called VARCHAR(*n*) or CHARACTER VARYING(*n*), where *n* is the *maximum* number of characters in the string.

3. An NCHAR(*n*) or NATIONAL CHARACTER(*n*) or NATIONAL CHAR(*n*) datatype, which is a fixed-length character string of *n* characters which uses the ISO8859_1 character set.

4. An NCHAR VARYING(*n*) or NATIONAL CHARACTER VARYING(*n*) or NATIONAL CHAR VARYING(*n*) datatype, which is a variable-length national character string up to a maximum of *n* characters.

Specifying a character set

When you define the datatype for a column, you can specify a character set for the column with the CHARACTER SET argument. This setting overrides the database default character set that is assigned when the database is created.

You can also change the default character set with SET NAMES in command-line **isql** or with the Session | Advanced Settings command in Windows ISQL. For details about using interactive SQL in either environment, see the *Operations Guide*.

The character set determines:

▪ What characters can be used in CHAR, VARCHAR, and BLOB text columns.

▪ The collation order to be used in sorting the column.

Note Collation order does not apply to BLOB data.

For example, the following statement creates a column that uses the ISO8859_1 character set, which is typically used in Europe to support European languages:

```
CREATE TABLE EMPLOYEE
    (FIRST_NAME VARCHAR(10) CHARACTER SET ISO8859_1,
    . . .);
```

For a list of the international character sets and collation orders that InterBase supports, see Chapter 14, "Character Sets and Collation Orders."

▶ *Characters vs. bytes*

The number of bytes that the system uses to store a single character can vary depending upon the character set. InterBase limits a character column to 32,767 bytes. Some character sets require two or three bytes per character, so the maximum number of characters allowed in *n* varies depending upon the character set used.

In the case of a single-byte character column, one character is stored in one byte, so the internal memory used to store the string is also 32,767 bytes. Therefore, you can define 32,767 characters per single-byte column without encountering an error.

In the case of multi-byte characters, one character does *not* equal one byte.
In the following example, the user specifies a CHAR datatype using the UNICODE_FSS character set:

```
CHAR (10922) CHARACTER SET UNICODE_FSS; /* succeeds */
CHAR (10923) CHARACTER SET UNICODE_FSS; /* fails */
```

This character set has a maximum size of 3 bytes for a single character. Because each character requires 3 bytes of internal storage, the maximum number of characters allowed without encountering an error is 10,922 (32,767 divided by 3 is approximately 10,922).

Note To determine the maximum number of characters allowed in the data definition statement of any multi-byte column, look up the number of bytes per character in Appendix A. Then divide 32,767 (the internal byte storage limit for any character datatype) by the number of bytes for each character. Two-byte character sets have a character limit of 16,383 per field, and a three-byte character set has a limit of 10,922 characters per field.

▶ *Using* CHARACTER SET NONE

If a default character set was not specified when the database was created, the character set defaults to NONE. Using CHARACTER SET NONE means that there is no character set assumption for columns; data is stored and retrieved just as you originally entered it. You can load any character set into a column defined with NONE, but you cannot load that same data into another column that has been defined with a different character set. No transliteration will be performed between the source and destination character sets, so in most cases, errors will occur during the attempted assignment.

For example:

```
CREATE TABLE MYDATA (PART_NUMBER CHARACTER(30) CHARACTER SET NONE);
SET NAMES LATIN1;
INSERT INTO MYDATA (PART_NUMBER) VALUES("à");
SET NAMES DOS437;
SELECT * FROM MYDATA;
```

The data ("à") is returned just as it was entered, without the à being transliterated from the input character (LATIN1) to the output character (DOS437). If the column had been set to anything other than NONE, the transliteration would have occurred.

▶ *About collation order*

Each character set has its own subset of possible collation orders. The character set that you choose when you define the datatype limits your choice of collation orders. The collation order for a column is specified when you create the table.

For a list of the international character sets and collation orders that InterBase supports, see Chapter 14, "Character Sets and Collation Orders."

Fixed-length character data

InterBase supports two fixed-length string datatypes: CHAR(*n*), or alternately CHARACTER (*n*), and NCHAR(*n*), or alternately NATIONAL CHAR(*n*).

▶ CHAR*(n) or* CHARACTER*(n)*

The CHAR(*n*) or CHARACTER(*n*) datatype contains character strings. The number of characters *n* is fixed. For the maximum number of characters allowed for the character set that you have specified, see Chapter 14, "Character Sets and Collation Orders."

When the string to be stored or read contains less than *n* characters, InterBase fills in the blanks to make up the difference. If a string is larger than *n*, then the value is truncated. If you do not supply *n*, it will default to 1, so CHAR is the same as CHAR(1). The next statement illustrates this:

```
CREATE TABLE SALES
( . . .
PAID CHAR
      DEFAULT 'n'
      CHECK (PAID IN ('y', 'n'),
. . .);
```

Trailing blanks   InterBase compresses trailing blanks when it stores fixed-length strings, so data with trailing blanks uses the same amount of space as an equivalent variable-length string. When the data is read, InterBase reinserts the blanks. This saves disk space when the length of the data items varies widely.

▶ NCHAR*(n) or* NATIONAL CHAR*(n)*

NCHAR(n) is exactly the same as CHARACTER(n), except that the ISO8859_1 character set is used by definition. Using NCHAR(n) is a shortcut for using the CHARACTER SET clause to specify the "ISO8859_1" character set for a column.

The next two CREATE TABLE examples are equivalent:

```
CREATE TABLE EMPLOYEE
    (. . .
    FIRST_NAME NCHAR(10),
    LAST_NAME NCHAR(15),
    . . .);
CREATE TABLE EMPLOYEE
    (. . .
    FIRST_NAME CHAR(10) CHARACTER SET "ISO8859_1",
    LAST_NAME CHAR(15) CHARACTER SET "ISO8859_1",
    . . .);
```

Variable-length character data

InterBase supports two variable-length string datatypes: VARCHAR(n), or alternately CHAR(n) VARYING, and NCHAR(n), or alternately NATIONAL CHAR(n) VARYING.

▶ VARCHAR*(n)*

VARCHAR(n)—also called CHAR VARYING(n), or CHARACTER VARYING(n)—allows you to store the exact number of characters that is contained in your data, up to a maximum of n. You must supply n; there is no default to 1.

If the length of the data within a column varies widely, and you do not want to pad your character strings with blanks, use the VARCHAR(n) or CHARACTER VARYING(n) datatype.

InterBase converts from variable-length character data to fixed-length character data by adding spaces to the value in the varying column until the column reaches its maximum length n. When the data is read, InterBase removes the blanks.

The main advantages of using the VARCHAR(n) datatype are that it saves disk space, and since more rows fit on a disk page, the database server can search the table with fewer disk I/O operations. The disadvantage is that table updates can be slower than using a fixed-length column in some cases.

The next statement illustrates the VARCHAR(*n*) datatype:

```
CREATE TABLE SALES
( . . .
ORDER_STATUS VARCHAR(7)
      DEFAULT "new"
      NOT NULL
      CHECK (ORDER_STATUS IN ("new", "open", "shipped", "waiting")),
. . .);
```

▶ NCHAR VARYING*(n)*

NCHAR VARYING(*n*)—also called NATIONAL CHARACTER VARYING (*n*) or NATIONAL CHAR VARYING(*n*)—is exactly the same as VARCHAR(*n*), except that the ISO8859_1 character set is used. Using NCHAR VARYING(*n*) is a shortcut for using the CHARACTER SET clause of CREATE TABLE, CREATE DOMAIN, or ALTER TABLE to specify the ISO8859_1 character set.

Defining BLOB datatypes

InterBase supports a dynamically sizable datatype called a BLOB to store data that cannot easily be stored in one of the standard SQL datatypes. A Blob is used to store very large data objects of indeterminate and variable size, such as bitmapped graphics images, vector drawings, sound files, video segments, chapter or book-length documents, or any other kind of multimedia information. Because a Blob can hold different kinds of information, it requires special processing for reading and writing. For more information about Blob handling, see the *Programmer's Guide*.

The BLOB datatype provides the advantages of a database management system, including transaction control, maintenance by database utilities, and access using SELECT, INSERT, UPDATE, and DELETE statements. Use the BLOB datatype to avoid storing pointers to non-database files.

BLOB columns

BLOB columns can be defined in database tables like non-BLOB columns. For example, the following statement creates a table with a BLOB column:

```
CREATE TABLE PROJECT
    (PROJ_ID PROJNO NOT NULL,
    PROJ_NAME VARCHAR(20) NOT NULL UNIQUE,
    PROJ_DESC BLOB,
```

```
TEAM_LEADER EMPNO,
PRODUCT PRODTYPE,
. . .);
```

Rather than storing BLOB data directly, a BLOB column stores a BLOB ID. A BLOB ID is a unique numeric value that references BLOB data. The BLOB data is stored elsewhere in the database, in a series of BLOB *segments*, units of BLOB data read and written in chunks. When a BLOB is created, data is written to it a segment at a time. Similarly, when a BLOB is read, it is read a segment at a time.

The following diagram shows the relationship between a BLOB column containing a BLOB ID and the BLOB data referenced by the BLOB ID:

FIGURE 4.1   BLOB relationships
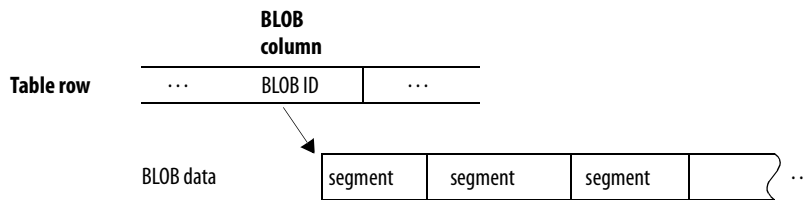
BLOB segment length

When a BLOB column is defined in a table, the BLOB definition can specify the expected size of BLOB segments that are written to the column. Actually, for SELECT, INSERT, and UPDATE operations, BLOB segments can be of varying length. For example, during insertion, a BLOB might be read in as three segments, the first segment having length 30, the second having length 300, and the third having length 3.

The length of an individual segment should be specified when it is written. For example, the following code fragment inserts a BLOB segment. The segment length is specified in the host variable, *segment_length*:

```
INSERT CURSOR BCINS VALUES (:write_segment_buffer:segment_length);
```

▶ *Defining segment length*

gpre, the InterBase precompiler, is used to process embedded SQL statements inside applications. The *segment length* setting, defined for a BLOB column when it is created, is used to determine the size of the internal buffer where the BLOB segment data will be written. This setting specifies (to **gpre**) the maximum number of bytes that an application is expected to write to any segment in the column. The default segment length is 80. Normally, an application should not attempt to write segments larger than the segment length defined in the table; doing so overflows the internal segment buffer, corrupting memory in the process.

The segment length setting does not affect InterBase system performance. Choose the segment length most convenient for the specific application. The largest possible segment length is 32 kilobytes (32,767 bytes).

▶ *Segment syntax*

The following statement creates two BLOB columns, BLOB1, with a default segment size of 80, and BLOB2, with a specified segment length of 512:

```
CREATE TABLE TABLE2
    (BLOB1 BLOB,
    BLOB2 BLOB SEGMENT SIZE 512);
```

BLOB subtypes

When a BLOB column is defined, its subtype can be specified. A BLOB *subtype* is a positive or negative integer that describes the nature of the BLOB data contained in the column. InterBase provides two predefined subtypes, 0, signifying that a BLOB contains binary data, the default, and 1, signifying that a BLOB contains ASCII text. User-defined subtypes must always be represented as negative integers. Positive integers are reserved for use by InterBase.

Blob subtype	Description
0	Unstructured, generally applied to binary data or data of an indeterminate type
1	Text
2	Binary language representation (BLR)
3	Access control list
4	(Reserved for future use)

Blob subtype	Description
5	Encoded description of a table's current metadata
6	Description of multi-database transaction that finished irregularly

For example, the following statement defines three BLOB columns: BLOB1 with subtype 0 (the default), BLOB2 with InterBase subtype 1 (TEXT), and BLOB3 with user-defined subtype –1:

```
CREATE TABLE TABLE2
    (BLOB1 BLOB,
    BLOB2 BLOB SUB_TYPE 1,
    BLOB3 BLOB SUB_TYPE -1);
```

The application is responsible for ensuring that data stored in a BLOB column agrees with its subtype. For example, if subtype –10 denotes a certain datatype in a particular application, then the application must ensure that only data of that datatype is written to a BLOB column of subtype –10. InterBase does not check the type or format of BLOB data.

To specify both a default segment length and a subtype when creating a BLOB column, use the SEGMENT SIZE option after the SUB_TYPE option, as in the following example:

```
CREATE TABLE TABLE2
    (BLOB1 BLOB SUB_TYPE 1 SEGMENT SIZE 100 CHARACTER SET DOS437;);
```

BLOB filters

BLOB subtypes are used in conjunction with BLOB filters. A BLOB *filter* is a routine that translates BLOB data from one subtype to another. InterBase includes a set of special internal BLOB filters that convert from subtype 0 to subtype 1 (TEXT), and from InterBase system subtypes to subtype 1 (TEXT). In addition to using the internal text filters, programmers can write their own external filters to provide special data translation. For example, an external filter might automatically translate from one bitmapped image format to another.

Note BLOB filters are not supported on NetWare servers.

Associated with every filter is an integer pair that specifies the input subtype and the output subtype. When declaring a cursor to read or write BLOB data, specify FROM and TO subtypes that correspond to a particular BLOB filter. InterBase invokes the filter based on the FROM and TO subtype specified by the read or write cursor declaration.

The display of BLOB subtypes in **isql** can be specified with SET BLOBDISPLAY in command-line **isql** or with the **Session | Advanced Settings** command in Windows ISQL.

For more information about Windows ISQL and command-line **isql**, see the *Operations Guide*. For more information about creating external BLOB filters, see the *Programmer's Guide*.

Defining arrays

InterBase allows you to create arrays of datatypes. Using an array enables multiple data items to be stored in a single column. InterBase can perform operations on an entire array, effectively treating it as a single element, or it can operate on an *array slice*, a subset of array elements. An array slice can consist of a single element, or a set of many contiguous elements.

Using an array is appropriate when:

- The data items naturally form a set of the same datatype.
- The entire set of data items in a single database column must be represented and controlled as a unit, as opposed to storing each item in a separate column.
- Each item must also be identified and accessed individually.

The data items in an array are called *array elements*. An array can contain elements of any InterBase datatype except BLOB, and cannot be an array of arrays. All of the elements of a particular array are of the same datatype.

Arrays are defined with the CREATE DOMAIN or CREATE TABLE statements. Defining an array column is just like defining any other column, except that the array dimensions must also be specified. For example, the following statement defines both a regular character column, and a single-dimension, character array column containing four elements:

```
EXEC SQL
    CREATE TABLE TABLE1
        (NAME CHAR(10),
        CHAR_ARR CHAR(10)[4]);
```

Array dimensions are always enclosed in square brackets following a column's datatype specification.

For a complete discussion of CREATE TABLE and array syntax, see the *Language Reference*. To learn more about the flexible data access provided by arrays, see the *Programmer's Guide*.

Multi-dimensional arrays

InterBase supports *multi-dimensional arrays*, arrays with 1 to 16 dimensions. For example, the following statement defines three INTEGER array columns with two, three, and six dimensions respectively:

```
EXEC SQL
CREATE TABLE TABLE1
        (INT_ARR2 INTEGER[4,5],
        INT_ARR3 INTEGER[4,5,6],
        INT_ARR6 INTEGER[4,5,6,7]);
```

In this example, INT_ARR2 allocates storage for 4 rows, 5 elements in width, for a total of 20 integer elements, INT_ARR3 allocates 120 elements, and INT_ARR6 allocates 840 elements.

IMPORTANT   InterBase stores multi-dimensional arrays in row-major order. Some host languages, such as FORTRAN, expect arrays to be in column-major order. In these cases, care must be taken to translate element ordering correctly between InterBase and the host language.

Specifying subscript ranges for array dimensions

In InterBase, array dimensions have a specific range of upper and lower boundaries, called *subscripts*. In many cases, the subscript range is implicit. The first element of the array is element 1, the second element 2, and the last is element *n*. For example, the following statement creates a table with a column that is an array of four integers:

```
EXEC SQL
CREATE TABLE TABLE1
        (INT_ARR INTEGER[4]);
```

The subscripts for this array are 1, 2, 3, and 4.

A different set of upper and lower boundaries for each array dimension can be explicitly defined when an array column is created. For example, C programmers, familiar with arrays that start with a lower subscript boundary of zero, might want to create array columns with a lower boundary of zero as well.

To specify array subscripts for an array dimension, both the lower and upper boundaries of the dimension must be specified using the following syntax:

lower:*upper*

For example, the following statement creates a table with a single-dimension array column of four elements where the lower boundary is 0 and the upper boundary is 3:

```
EXEC SQL
CREATE TABLE TABLE1
        (INT_ARR INTEGER[0:3]);
```

The subscripts for this array are 0, 1, 2, and 3.

When creating multi-dimensional arrays with explicit array boundaries, separate each dimension's set of subscripts from the next with commas. For example, the following statement creates a table with a two-dimensional array column where each dimension has four elements with boundaries of 0 and 3:

```
EXEC SQL
CREATE TABLE TABLE1
        (INT_ARR INTEGER[0:3, 0:3]);
```

Converting datatypes

Normally, you must use compatible datatypes to perform arithmetic operations, or to compare data in search conditions. If you need to perform operations on mixed datatypes, or if your programming language uses a datatype that is not supported by InterBase, then datatype conversions must be performed before the database operation can proceed. InterBase either automatically converts the data to an equivalent datatype (an implicit type conversion), or you can use the **cast()** function in search conditions to explicitly translate one datatype into another for comparison purposes.

Implicit type conversions

InterBase automatically converts columns of an unsupported datatype to an equivalent one, if required. This is an implicit datatype conversion. For example, in the following operation,

```
3 + '1' = 4
```

InterBase automatically converts the character "1" to an INTEGER for the addition operation.

The next example returns an error because InterBase cannot convert the "a" to an INTEGER:

```
3 + 'a' = 4
```

Explicit type conversions

When InterBase cannot do an implicit type conversion, you must perform an explicit type conversion using the **cast()** function. Use **cast()** to convert one datatype to another inside a SELECT statement. Typically, **cast()** is used in the WHERE clause to compare different datatypes. The syntax is:

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

Use **cast()** to translate a:

- DATE datatype into a CHARACTER or NUMERIC datatype.

- CHARACTER datatype into a NUMERIC or DATE datatype.

- NUMERIC datatype into a CHARACTER or DATE datatype.

For example, in the following WHERE clause, **cast()** is used to translate a CHAR datatype, INTERVIEW_DATE, to a DATE datatype in order to compare against a DATE datatype, HIRE_DATE:

```
… WHERE HIRE_DATE = (CAST(INTERVIEW_DATE AS DATE);
```

In the next example, **cast()** is used to translate a DATE datatype into a CHAR datatype:

```
… WHERE CAST(HIRE_DATE AS CHAR) = INTERVIEW_DATE;
```

You can use **cast()** to compare columns with different datatypes in the same table, or across tables.

5

Working with Domains

This chapter describes how to:

- Create a domain.
- Alter a domain.
- Drop a domain.

Creating domains

When you create a table, you can use a global column definition, called a *domain*, to define a column locally. Before defining a column that references a domain, you must first create the domain definition in the database with CREATE DOMAIN. CREATE DOMAIN acts as a template for defining columns in subsequent CREATE TABLE and ALTER TABLE statements. For more information on creating and modifying tables, see Chapter 6, "Working with Tables."

Domains are useful when many tables in a database contain identical column definitions. Columns based on a domain definition inherit all characteristics of the domain; some of these attributes can be overridden by local column definitions.

Note You cannot apply referential integrity constraints to a domain.

The syntax for CREATE DOMAIN is:

```
CREATE DOMAIN domain [AS] <datatype>
[DEFAULT {literal | NULL | USER}]
[NOT NULL] [CHECK (<dom_search_condition>)]
[COLLATE collation];
```

Using CREATE DOMAIN

When you create a domain in the database, you must specify a unique name for the domain, and define the various attributes and constraints of the column definition. These attributes include:

- datatype
- Default values and NULL status
- CHECK constraints
- Collation order

Specifying the domain datatype

The *datatype* is the only required attribute that must be set for the domain—all other attributes are optional. The datatype defines the set of valid data that the column can contain. The datatype also determines the set of allowable operations that can be performed on the data, and defines the disk space requirements for each data item.

The syntax for specifying the datatype is:

```
<datatype> = {
{SMALLINT | INTEGER | FLOAT | DOUBLE PRECISION} [<array_dim>]
| {DECIMAL | NUMERIC} [(precision [, scale])] [<array_dim>]
| DATE [<array_dim>]
| {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR}
    [(int)] [<array_dim>] [CHARACTER SET charname]
| {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
    [VARYING] [(int)] [<array_dim>]
| BLOB [SUB_TYPE {int | subtype_name}] [SEGMENT SIZE int]
    [CHARACTER SET charname]
| BLOB [(seglen [, subtype])]
}
<array_dim> = [x:y [, x1:y1 …]]
```
Note The outermost (boldface) brackets must be included when declaring arrays.

datatype is the SQL datatype for any column based on a domain. You cannot override the domain datatype with a local column definition.

The general categories of SQL datatypes include:

- Character datatypes.
- Integer datatypes.
- Decimal datatypes, both fixed and floating.
- A DATE datatype to represent date and time. InterBase does not directly support the SQL DATE, TIME, and TIMESTAMP datatypes.
- A BLOB datatype to represent unstructured binary data, such as graphics and digitized voice.
- Arrays of datatypes (except for BLOB data).

InterBase supports the following datatypes:

Name	Size	Range/Precision	Description
BLOB	Variable	None. BLOB segment size is limited to 64K.	Binary large object. Stores large data, such as graphics, text, and digitized voice. Basic structural unit: segment. BLOB subtype describes BLOB contents.
CHAR(*n*)	*n* characters	1 to 32767 bytes.	Fixed length CHAR or text string type.
		Character set character size determines the maximum number of characters that can fit in 32K.	Alternate keyword: CHARACTER.
DATE	64 bits	1 Jan 100 to 11 Dec 5941.	Also included time information.
DECIMAL (*precision*, *scale*)	variable	*precision* = 1 to 15. Specifies at least *precision* digits of precision to store.	Number with a decimal point *scale* digits from the right. For example, DECIMAL(10, 3) holds numbers accurately in the following format:
		scale = 1 to 15. Specifies number of decimal places for storage. Must be less than or equal to *precision*.	pppppppp.sss
DOUBLE PRECISION	64 bits[a]	1.7×10^{-308} to 1.7×10^{308}.	Scientific: 15 digits of precision.
FLOAT	32 bits	3.4×10^{-38} to 3.4×10^{38}	Single precision: 7 digits of precision.
INTEGER	32 bits	$-2,147,483,648$ to 2,147,483,648.	Signed long (longword).

TABLE 5.1   Datatypes supported by InterBase

Name	Size	Range/Precision	Description
NUMERIC (*precision, scale*)	variable	*precision* = 1 to 15. Specifies exactly *precision* digits of precision to store.	Number with a decimal point *scale* digits from the right. For example, NUMERIC(10,3) holds numbers accurately in the following format:
		scale = 1 to 15. Specifies number of decimal places for storage. Must be less than or equal to *precision*.	ppppppp.sss
SMALLINT	16 bits	−32768 to 32767.	Signed short (word).
VARCHAR(*n*)	*n* characters	1 to 32765 bytes.	Variable length CHAR or text string type.
		Character set character size determines the maximum number of characters that can fit in 32K.	Alternate keywords: CHAR VARYING, CHARACTER VARYING

TABLE 5.1   Datatypes supported by InterBase (*continued*)

a.   Actual size of DOUBLE is platform-dependent. Most platforms support the 64-bit size.

For more information about datatypes, see Chapter 4, "Specifying Datatypes."

The following statement creates a domain that defines an array of CHARACTER datatype:

```
CREATE DOMAIN DEPTARRAY AS CHAR(31) [4:5];
```

The next statement creates a BLOB domain with a text subtype that has an assigned character set:

```
CREATE DOMAIN DESCRIPT AS BLOB SUB_TYPE TEXT SEGMENT SIZE 80
CHARACTER SET SJIS;
```

Specifying domain collation order

The COLLATE clause of CREATE DOMAIN allows you to specify a particular collation order for columns defined as CHAR or VARCHAR text datatypes. You must choose a collation order that is supported for the column's given character set. The character set is either the default character set for the entire database, or you can specify a different set in the CHARACTER SET clause of the datatype definition. The collation order set at the column level overrides a collation order set at the domain level.

For a list of the collation orders available for each character set, see Chapter 14, "Character Sets and Collation Orders."

In the following statement, the domain, TITLE, overrides the database default character set, specifying a DOS437 character set with a PDOX_INTL collation order:

```
CREATE DOMAIN TITLE AS
CHAR(50) CHARACTER SET DOS437 COLLATE PDOX_INTL;
```

Specifying domain defaults

You can set an optional default value that is automatically entered into a column if you do not specify an explicit value. Defaults set at the column level with CREATE TABLE or ALTER TABLE override defaults set at the domain level. Defaults can save data entry time and prevent data entry errors. For example, a possible default for a DATE column could be today's date, or in a (Y/N) flag column for saving changes, "Y" could be the default.

Default values can be:

- *literal:* The default value is a user-specified string, numeric value, or date value.

- NULL: If the user does not enter a value, a NULL value is entered into the column.

- USER: The default is the name of the current user. If your operating system supports the use of 8 or 16-bit characters in user names, then the column into which USER will be stored must be defined using a compatible character set.

In the following example, the first statement creates a domain with USER named as the default. The next statement creates a table that includes a column, ENTERED_BY, based on the USERNAME domain.

```
CREATE DOMAIN USERNAME AS VARCHAR(20)
DEFAULT USER;
CREATE TABLE ORDERS (ORDER_DATE DATE, ENTERED_BY USERNAME, ORDER_AMT
DECIMAL(8,2));
INSERT INTO ORDERS (ORDER_DATE, ORDER_AMT)
VALUES ("1-MAY-93", 512.36);
```

The INSERT statement does not include a value for the ENTERED_BY column, so InterBase automatically inserts the user name of the current user, JSMITH:

```
SELECT * FROM ORDERS;
1-MAY-93 JSMITH 512.36
```

Specifying NOT NULL

You can optionally specify NOT NULL to force the user to enter a value. If you do not specify NOT NULL, then NULL values are allowed for any column that references this domain. NOT NULL specified on the domain level cannot be overridden by a local column definition.

Note If you have already specified NULL as a default value, be sure not to create contradictory constraints by also assigning NOT NULL to the domain, as in the following example:

```
CREATE DOMAIN DOM1 INTEGER DEFAULT NULL, NOT NULL;
```

Specifying domain CHECK constraints

You can specify a condition or requirement on a data value at the time the data is entered by applying a CHECK constraint to a column. The CHECK constraint in a domain definition sets a search condition (*dom_search_condition*) that must be true before data can be entered into columns based on the domain.

The syntax of the search condition is:

```
<dom_search_condition> = {
VALUE <operator> <val>
| VALUE [NOT] BETWEEN <val> AND <val>
| VALUE [NOT] LIKE <val> [ESCAPE <val>]
| VALUE [NOT] IN (<val> [, <val> …])
| VALUE IS [NOT] NULL
| VALUE [NOT] CONTAINING <val>
| VALUE [NOT] STARTING [WITH] <val>
| (<dom_search_condition>)
| NOT <dom_search_condition>
| <dom_search_condition> OR <dom_search_condition>
| <dom_search_condition> AND <dom_search_condition>
}
<operator> = {= | < | > | <= | >= | !< | !> | <> | !=}
```

The following restrictions apply to CHECK constraints:

- A CHECK constraint cannot reference any other domain or column name.

- A domain can have only one CHECK constraint.

- You cannot override the domain's CHECK constraint with a local CHECK constraint. A column based on a domain can add *additional* CHECK constraints to the local column definition.

Using the VALUE keyword

VALUE defines the set of values that is valid for the domain. VALUE is a placeholder for the name of a column that will eventually be based on the domain. The search condition can verify whether the value entered falls within a certain range, or match it to any one value in a list of values.

Note If NULL values are allowed, they must be included in the CHECK constraint, as in the following example:

```
CHECK ((VALUE IS NULL) OR (VALUE > 1000));
```

The next statement creates a domain where value must be > 1,000:

```
CREATE DOMAIN CUSTNO
AS INTEGER
CHECK (VALUE > 1000);
```

The following statement creates a domain that must have a positive value greater than 1,000, with a default value of 9,999.

```
CREATE DOMAIN CUSTNO
    AS INTEGER
    DEFAULT 9999
    CHECK (VALUE > 1000);
```

The next statement limits the values entered in the domain to four specific values:

```
CREATE DOMAIN PRODTYPE
    AS VARCHAR(12)
    CHECK (VALUE IN ("software", "hardware", "other", "N/A"));
```

When a problem cannot be solved using comparisons, you can instruct the system to search for a specific pattern in a character column. For example, the next search condition allows only cities in California to be entered into columns that are based on the CALIFORNIA domain:

```
CREATE DOMAIN CALIFORNIA
AS VARCHAR(25)
CHECK (VALUE LIKE "%, CA");
```

Altering domains with ALTER DOMAIN

ALTER DOMAIN changes any aspect of an existing domain except its datatype and NOT NULL setting. Changes that you make to a domain definition affect all column definitions based on the domain that have not been overridden at the table level.

Note To change a datatype or NOT NULL setting of a domain, drop the domain and recreate it with the desired combination of features.

A domain can be altered by its creator, the SYSDBA user, and any users with operating system root privileges.

ALTER DOMAIN allows you to:

- Drop an existing default value.
- Set a new default value.
- Drop an existing CHECK constraint.
- Add a new CHECK constraint.

The syntax for ALTER DOMAIN is:

```
ALTER DOMAIN name {
[SET DEFAULT {literal | NULL | USER}]
| [DROP DEFAULT]
| [ADD [CONSTRAINT] CHECK (<dom_search_condition>)]
| [DROP CONSTRAINT]
};
```

The following statement sets a new default value for the CUSTNO domain:

```
ALTER DOMAIN CUSTNO SET DEFAULT 9999;
```

Dropping a domain

DROP DOMAIN removes an existing domain definition from a database.

If a domain is currently used in any column definition in the database, the DROP operation fails. To prevent failure, delete the columns based on the domain with ALTER TABLE before executing DROP DOMAIN.

A domain can be dropped by its creator, the SYSDBA, and any users with operating system root privileges.

The syntax of DROP DOMAIN is:

```
DROP DOMAIN name;
```

The following statement deletes a domain:

```
DROP DOMAIN COUNTRYNAME;
```

6

Working with Tables

This chapter describes:

- What to do before creating a table.
- How to create database tables.
- How to alter tables.
- How to drop tables.

Before creating a table

Before creating a table, you should have:

- Designed, normalized, created, and connected to the database.
- Determined which tables, columns, and column definitions to create.
- Created the domain definitions in the database. For information on how to create, drop, and modify domains, see Chapter 5, "Working with Domains."
- Declared the table if an embedded SQL application both creates a table and populates the table with data in the same program. The DECLARE TABLE statement must precede CREATE TABLE. For the syntax of DECLARE TABLE, see the *Language Reference*.

Creating tables

You can create tables in the database with the CREATE TABLE statement. The syntax for CREATE TABLE is:

```
CREATE TABLE table [EXTERNAL [FILE] "<filespec>"]
(<col_def> [, <col_def> | <tconstraint> ...]);
```

The first argument that you supply to CREATE TABLE is the table name, which is required, and must be unique among all table and procedure names in the database. You must also supply at least one column definition.

InterBase automatically imposes the default SQL security scheme on the table. The person who creates the table (the owner), is assigned all privileges for it, including the right to grant privileges to other users, triggers, and stored procedures. For more information on security, see Chapter 13, "Planning Security."

For a detailed specification of CREATE TABLE syntax, see the *Language Reference*.

Using the EXTERNAL FILE option

The EXTERNAL FILE option creates a table for which the data resides in an external table or file, rather than in the InterBase database. External files are ASCII text that can also be read and manipulated by non-InterBase applications. See the syntax for CREATE TABLE. *filespec* is the fully qualified file specification for the external data file. You can modify the external file outside InterBase; InterBase accesses it only when needed.

Use the EXTERNAL FILE option to:

- Import data from a flat external file in a known fixed-length format into a new or existing InterBase table. This allows you to populate an InterBase table with data from an external source. Many applications allow you to create an external file with fixed-length records.

- SELECT from the external file as if it were a standard InterBase table.

- Export data from an existing InterBase table to an external file. You can format the data from the InterBase table into a fixed-length file that another application can use.

Restrictions

The following restrictions apply to using the EXTERNAL FILE option:

- You must create the external file before you try to access the external table inside of the database.

- Each record in the external file must be of fixed length. You cannot put BLOB or array data into an external file.

- When you create the table that will be used to import the external data, you must define a column to contain the end-of-line (EOL) or new-line character. The size of this column must be exactly large enough to contain a particular system's EOL symbol (usually one or two bytes). For most versions of Unix, it is 1 byte. For Windows, NT, and NetWare, it is 2 bytes.

- While it is possible to read in numeric data directly from an external table, it is much easier to read it in as character data, and convert using the **cast()** function.

- Data to be treated as VARCHAR in InterBase must be stored in an external file in the following format:

  ```
  <2-byte unsigned short><string of character bytes>
  ```

- where the 2-byte unsigned short indicates the number of bytes in the actual string, and the string immediately follows. Because of its lack of easy portability, use of VARCHAR data in an external file is not recommended.

- You can only INSERT into and SELECT from the rows of an external table. You cannot UPDATE or DELETE from an external table; if you try to do so, InterBase returns an error message.

- Inserting into and selecting from an external table are not under standard transaction control because the external file is outside of the database. Therefore, changes are immediate and permanent—you cannot roll back your changes. If you want your table to be under transaction control, create another internal InterBase table, and insert the data from the external table into the internal one.

- If you use DROP DATABASE to delete the database, you must also remove the external file— it will not be automatically deleted as a result of DROP DATABASE.

Importing external files to InterBase tables

The following steps describe how to import an external file into an InterBase table:

1. Create an InterBase table that allows you to view the external data. Declare all columns as CHAR. The text file containing the data must be on the server. In the following example, the external file exists on a Unix system, so the EOL character is 1 byte.

   ```
   CREATE TABLE EXT_TBL EXTERNAL FILE "file.txt"
     (
   ```

```
    FNAME CHAR(10),
    LNAME CHAR(20),
    HDATE CHAR(8),
    NEWLINE CHAR(1)
);
COMMIT;
```

2. Create another InterBase table that will eventually be your working table. If you expect to export data from the internal table back to an external file at a later time, be sure to create a column to hold the newline. Otherwise, you do not need to leave room for the newline character. In the following example, a column for the newline is provided:

```
CREATE TABLE PEOPLE
(
    FIRST_NAME CHAR(10),
    LAST_NAME CHAR(20),
    HIRE_DATE CHAR(8),
    NEW_LINE CHAR(1)
);
COMMIT;
```

3. Create and populate the external file. You can create the file with a text editor, or you can create an appropriate file with an application like Paradox for Windows or dBASE for Windows. If you create the file yourself with a text editor, make each record the same length, pad the unused characters with blanks, and insert the EOL character(s) at the end of each record.

Note The number of characters in the EOL character is platform-specific. You need to know how many characters are contained in your platform's EOL (typically one or two) in order to correctly format the columns of the tables and the corresponding records in the external file. In the following example, the record length is 36 characters. "b" represents a blank space, and "n" represents the EOL:

```
123456789012345678901234567890123456
fname.....lname.............hdate..n
-----------------------------------
Robertbbbbbrickmanbbbbbbbbbb6/12/92n
Sambbbbbbbbbjonesbbbbbbbbbbbb12/13/93n
```

4. At this point, when you do a SELECT statement from table EXT_TBL, you will see the records from the external file:

```
SELECT FNAME, LNAME, HDATE FROM EXT_TBL;
FNAME      LNAME             HDATE
========   ================= ===========
Robert     Brickman          12-JUN-1992
Sam        Jones             13-DEC-1993
```

5. Insert the data into the destination table.

```
INSERT INTO PEOPLE SELECT FNAME, LNAME, CAST(HDATE AS DATE),
    NEWLINE FROM EXT_TBL;
```

Now if you SELECT from PEOPLE, the data from your external table will be there.

```
SELECT FIRST_NAME, LAST_NAME, HIRE_DATE FROM PEOPLE;
FIRST_NAME LAST_NAME           HIRE_DATE
========== =================== ===========
Robert     Brickman            12-JUN-1992
Sam        Jones               13-DEC-1993
```

InterBase allows you to store the date as an integer by converting from a CHAR(8) to DATE using the CAST() function.

Exporting InterBase tables to an external file

If you add, update, or delete a record from an internal table, the changes will not be reflected in the external file. So in the previous example, if you delete the "Sam Jones" record from the PEOPLE table, and do a subsequent SELECT from EXT_TBL, you would still see the "Sam Jones" record.

This section explains how to export InterBase data to an external file. Using the example developed in the previous section, follow these steps:

1. Open the external file in a text editor and remove everything from the file. If you then do a SELECT on EXT_TBL, it should be empty.

2. Use an INSERT statement to copy the InterBase records from PEOPLE into the external file, *file.txt*.

```
INSERT INTO EXT_TBL SELECT FIRST_NAME, LAST_NAME, HIRE_DATE, NEW_LINE
FROM PEOPLE WHERE FIRST_NAME LIKE "Rob%";
```

3. Now if you do a SELECT from the external table, EXT_TBL, only the records you inserted should be there. In this example, only a single record should be displayed:

```
SELECT FNAME, LNAME, HDATE FROM EXT_TBL;
FNAME      LNAME             HDATE
========   ================= ===========
Robert     Brickman          12-JUN-1992
```

IMPORTANT   Make sure that all records that you intend to export from the internal table to the external file have the correct EOL character(s) in the newline column.

Defining columns

When you create a table in the database, your main task is to define the various attributes and constraints for each of the columns in the table. The syntax for defining a column is:

```
<col_def> = col {datatype | COMPUTED [BY] (<expr>) | domain}
[DEFAULT {literal | NULL | USER}]
[NOT NULL] [<col_constraint>]
[COLLATE collation]
```

The next sections list the required and optional attributes that you can define for a column.

▸ *Required attributes*

You are required to specify:

▪ A column name, which must be unique among the columns in the table.

▪ *One* of the following:

· An SQL datatype (*datatype*).

· An expression (*expr*) for a computed column.

· A domain definition (*domain*) for a domain-based column.

▸ *Optional attributes*

You have the option to specify:

▪ A default value for the column.

▪ Integrity constraints. Constraints can be applied to a set of columns (a table-level constraint), or to a single column (a column-level constraint). Integrity constraints include:

· The PRIMARY KEY column constraint, if the column is a PRIMARY KEY, and the PRIMARY KEY constraint is not defined at the table level. Creating a PRIMARY KEY requires exclusive database access.

· The UNIQUE constraint, if the column is not a PRIMARY KEY, but should still disallow duplicate and NULL values.

· The FOREIGN KEY constraint, if the column references a PRIMARY KEY in another table. Creating a FOREIGN KEY requires exclusive database access. The foreign key constraint includes the ON UPDATE and ON DELETE mechanisms for specifying what happens to the foreign key when the primary key is updated (cascading referential integrity).

▪ A NOT NULL attribute does not allow NULL values. This attribute is required if the column is a PRIMARY KEY or UNIQUE key.

- A CHECK constraint for the column. A CHECK constraint enforces a condition that must be true before an insert or an update to a column or group of columns is allowed.

- A CHARACTER SET can be specified for a single column when you define the datatype. If you do not specify a character set, the column assumes the database character set as a default.

▶ *Specifying the datatype*

When creating a table, you must specify the datatype for each column. The *datatype* defines the set of valid data that the column can contain. The datatype also determines the set of allowable operations that can be performed on the data, and defines the disk space requirements for each data item.

The syntax for specifying the datatype is:

```
<datatype> = {
{SMALLINT | INTEGER | FLOAT | DOUBLE PRECISION} [<array_dim>]
| {DECIMAL | NUMERIC} [(precision [, scale])] [<array_dim>]
| DATE [<array_dim>]
| {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR}
    [(int)] [<array_dim>] [CHARACTER SET charname]
| {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
    [VARYING] [(int)] [<array_dim>]
| BLOB [SUB_TYPE {int | subtype_name}] [SEGMENT SIZE int]
    [CHARACTER SET charname]
| BLOB [(seglen [, subtype])]
}
<array_dim> = [x:y [, x1:y1 ...]]
```

Note Outermost (boldface) brackets must be included when declaring arrays.

The general categories of datatypes that are supported include:

- Character datatypes.

- Integer datatypes.

- Decimal datatypes, both fixed and floating.

- A DATE datatype to represent date and time. InterBase does not directly support the SQL DATE, TIME, and TIMESTAMP datatypes.

- A BLOB datatype to represent unstructured binary data, such as graphics and digitized voice.

- Arrays of datatypes (except for BLOB data).

Note If your application programming language does not support a particular datatype, InterBase either automatically converts the data to an equivalent datatype (an implicit type conversion), or you can use the **cast()** function in search conditions to explicitly translate one datatype into another for comparison purposes. For more information about specifying datatypes and using the **cast()** function, see Chapter 4, "Specifying Datatypes."

InterBase supports the following datatypes:

Name	Size	Range/Precision	Description
BLOB	Variable	None. BLOB segment size is limited to 64K.	Binary large object. Stores large data, such as graphics, text, and digitized voice. Basic structural unit: segment. BLOB subtype describes BLOB contents.
CHAR(*n*)	*n* characters	1 to 32767 bytes.	Fixed length CHAR or text string type.
		Character set character size determines the maximum number of characters that can fit in 32K.	Alternate keyword: CHARACTER.
DATE	64 bits	1 Jan 100 to 11 Dec 5941.	Also included time information.
DECIMAL (*precision*, *scale*)	variable	*precision* = 1 to 15. Specifies at least *precision* digits of precision to store.	Number with a decimal point *scale* digits from the right. For example, DECIMAL(10, 3) holds numbers accurately in the following format:
		scale = 1 to 15. Specifies number of decimal places for storage. Must be less than or equal to *precision*.	ppppppp.sss
DOUBLE PRECISION	64 bits[a]	1.7×10^{-308} to 1.7×10^{308}.	Scientific: 15 digits of precision.
FLOAT	32 bits	3.4×10^{-38} to 3.4×10^{38}	Single precision: 7 digits of precision.
INTEGER	32 bits	$-2,147,483,648$ to $2,147,483,648$.	Signed long (longword).

TABLE 6.1   Datatypes supported by InterBase

Name	Size	Range/Precision	Description
NUMERIC (*precision*, *scale*)	variable	*precision* = 1 to 15. Specifies exactly *precision* digits of precision to store.	Number with a decimal point *scale* digits from the right. For example, NUMERIC(10,3) holds numbers accurately in the following format:
		scale = 1 to 15. Specifies number of decimal places for storage. Must be less than or equal to *precision*.	ppppppp.sss
SMALLINT	16 bits	−32768 to 32767.	Signed short (word).
VARCHAR(*n*)	*n* characters	1 to 32765 bytes.	Variable length CHAR or text string type.
		Character set character size determines the maximum number of characters that can fit in 32K.	Alternate keywords: CHAR VARYING, CHARACTER VARYING

TABLE 6.1   Datatypes supported by InterBase (*continued*)

a.   Actual size of DOUBLE is platform-dependent. Most platforms support the 64-bit size.

▶ *Defining a* CHARACTER SET

The datatype specification for a CHAR, VARCHAR, or BLOB text column definition can include a CHARACTER SET clause to specify a particular character set for a column. If you do not specify a character set, the column assumes the default database character set. If the database default character set is subsequently changed, all columns defined after the change have the new character set, but existing columns are not affected. For a list of available character sets recognized by InterBase, see Chapter 14, "Character Sets and Collation Orders."

THE COLLATE CLAUSE

The collation order determines the order in which values are sorted. The COLLATE clause of CREATE TABLE allows you to specify a particular collation order for columns defined as CHAR and VARCHAR text datatypes. You must choose a collation order that is supported for the column's given character set. The character set is either the default character set for the entire database, or you can specify a different set in the CHARACTER SET clause of the datatype definition. The collation order set at the column level overrides a collation order set at the domain level.

In the following statement, BOOKNO keeps the default collating order for the database's default character set. The second (TITLE) and third (EUROPUB) columns specify different character sets and collating orders.

```
CREATE TABLE BOOKADVANCE (BOOKNO CHAR(6),
TITLE CHAR(50) CHARACTER SET DOS437 COLLATE PDOX_INTL,
EUROPUB CHAR(50) CHARACTER SET ISO8859_1 COLLATE FR_FR);
```

For a list of the available characters sets and collation orders that InterBase recognizes, see Chapter 14, "Character Sets and Collation Orders."

▶ *Defining domain-based columns*

When you create a table, you can set column attributes by using an existing domain definition that has been previously stored in the database. A *domain* is a global column definition. Domains must be created with the CREATE DOMAIN statement before you can reference them to define columns locally. For information on how to create a domain, see Chapter 5, "Working with Domains."

Domain-based columns inherit all the characteristics of a domain, but the column definition can include a new default value, additional CHECK constraints, or a collation clause that overrides the domain definition. It can also include additional column constraints. You can specify a NOT NULL setting if the domain does not already define one.

Note You cannot override the domain's NOT NULL setting with a local column definition.

For example, the following statement creates a table, COUNTRY, referencing the domain, COUNTRYNAME, which was previously defined with a datatype of VARCHAR(15):

```
CREATE TABLE COUNTRY
    (COUNTRY COUNTRYNAME NOT NULL PRIMARY KEY,
CURRENCY VARCHAR(10) NOT NULL);
```

▶ *Defining expression-based columns*

A computed column is one whose value is calculated each time the column is accessed at run time. The syntax is:

```
<col_name> COMPUTED [BY] (<expr>);
```

If you do not specify the datatype, InterBase calculates an appropriate one. *expr* is any arithmetic expression that is valid for the datatypes in the columns; it must return a single value, and cannot be an array or return an array. Columns referenced in the expression must exist before the COMPUTED [BY] clause can be defined.

For example, the following statement creates a computed column, FULL_NAME, by concatenating the LAST_NAME and FIRST_NAME columns.

```
CREATE TABLE EMPLOYEE
    (FIRST_NAME VARCHAR(10) NOT NULL,
LAST_NAME VARCHAR(15) NOT NULL,
FULL_NAME COMPUTED BY (LAST_NAME || ", " || FIRST_NAME));
```

The next example creates a table with a calculated column (NEW_SALARY) using the previously created EMPNO and SALARY domains.

```
CREATE TABLE SALARY_HISTORY
    (EMP_NO EMPNO NOT NULL,
    CHANGE_DATE DATE DEFAULT "NOW" NOT NULL,
    UPDATER_ID VARCHAR(20) NOT NULL,
    OLD_SALARY SALARY NOT NULL,
    PERCENT_CHANGE DOUBLE PRECISION
    DEFAULT 0
    NOT NULL
    CHECK (PERCENT_CHANGE BETWEEN -50 AND 50),
    NEW_SALARY COMPUTED BY
    (OLD_SALARY + OLD_SALARY * PERCENT_CHANGE / 100),
    PRIMARY KEY (EMP_NO, CHANGE_DATE, UPDATER_ID),
    FOREIGN KEY (EMP_NO) REFERENCES EMPLOYEE (EMP_NO)
        ON UPDATE CASCADE
        ON DELETE CASCADE);
```

Note Constraints on computed columns are not enforced, but InterBase does not return an error if you do define such a constraint.

▶ *Specifying column default values*

You can set an optional default value that is automatically entered into a column if you do not specify an explicit value. Defaults set at the column level with CREATE TABLE or ALTER TABLE override defaults set at the domain level. Defaults can save data entry time and prevent data entry errors. For example, a possible default for a DATE column could be today's date, or in a (Y/N) flag column for saving changes, "Y" could be the default.

Default values can be:

▪ *literal*—The default value is a user-specified string, numeric value, or date value.

▪ NULL—If the user does not enter a value, a NULL value is entered into the column.

▪ USER—The default is the name of the current user. If your operating system supports the use of 8 or 16-bit characters in user names, then the column into which USER will be stored must be defined using a compatible character set.

In the following example, the first statement creates a domain with USER named as the default. The next statement creates a table that includes a column, ENTERED_BY, based on the USERNAME domain.

```
CREATE DOMAIN USERNAME AS VARCHAR(20)
DEFAULT USER;
CREATE TABLE ORDERS (ORDER_DATE DATE, ENTERED_BY USERNAME, ORDER_AMT
DECIMAL(8,2));
INSERT INTO ORDERS (ORDER_DATE, ORDER_AMT)
VALUES ("1-MAY-93", 512.36);
```

The INSERT statement does not include a value for the ENTERED_BY column, so InterBase automatically inserts the user name of the current user, JSMITH:

```
SELECT * FROM ORDERS;
```

▶ *Specifying* NOT NULL

You can optionally specify NOT NULL to force the user to enter a value. If you do *not* specify NOT NULL, then NULL values are allowed in the column. You cannot override a NOT NULL setting that has been set at a domain level with a local column definition.

Note If you have already specified NULL as a default value, be sure not to create contradictory constraints by also specifying the NOT NULL attribute, as in the following example:

```
CREATE TABLE MY_TABLE (COUNT INTEGER DEFAULT NULL NOT NULL);
```

▶ *Defining integrity constraints*

InterBase allows you to optionally apply certain constraints to a column, called *integrity constraints*, which are the rules that govern column-to-table and table-to-table relationships, and validate data entries. They span all transactions that access the database and are automatically maintained by the system. Integrity constraints can be applied to an entire table or to an individual column.

PRIMARY KEY AND UNIQUE CONSTRAINTS

The PRIMARY KEY and UNIQUE integrity constraints ensure that the values entered into a column or set of columns are unique in each row. If you try to insert a duplicate value in a PRIMARY KEY or UNIQUE column, the database will prevent the operation and return an error. When you define a UNIQUE or PRIMARY KEY column, determine whether the data stored in the column is inherently unique. For example, no two social security numbers or driver's license numbers are ever the same. If no single column has this property, then define the primary key as a composite of two or more columns which, when taken together, are unique.

EMP_NO	LAST_NAME	FIRST_NAME	JOB_TITLE	PHONE_EXT
10335	Smith	John	Engineer	4968
21347	Carter	Catherine	Product Manager	4967
13314	Jones	Sarah	Senior Writer	4800

TABLE 6.2   The EMPLOYEE table

In the EMPLOYEE table, EMP_NO is the primary key that uniquely identifies each employee. EMP_NO is the primary key because no two values in the column are alike. If the EMP_NO column did not exist, then no other column is a candidate for primary key due to the high probability for duplication of values. LAST_NAME, FIRST_NAME, and JOB_TITLE fail because more than one employee can have the same first name, last name, and job title. In a large database, a *combination* of LAST_NAME and FIRST_NAME could still result in duplicate values. A primary key that combines LAST_NAME and PHONE_EXT might work, but there could be two people with identical last names at the same extension. In this table, the EMP_NO column is actually the only acceptable candidate for the primary key because it *guarantees* a unique number for each employee in the table.

A table can have only one primary key. If you define a PRIMARY KEY constraint at the table level, you cannot do it again at the column level. The reverse is also true; if you define a PRIMARY KEY constraint at the column level, you cannot define a primary key at the table level. You must define the NOT NULL attribute for a PRIMARY KEY column in order to preserve the uniqueness of the data values in that column.

Like primary keys, a unique key ensures that no two rows have the same value for a specified column or ordered set of columns. You must define the NOT NULL attribute for a UNIQUE column. A unique key is different from a primary key in that the UNIQUE constraint specifies *alternate* keys that you can use to uniquely identify a row. You can have more than one unique key defined for a table, but the same set of columns cannot make up more than one PRIMARY KEY or UNIQUE constraint for a table. Like a primary key, a unique key can be referenced by a foreign key in another table.

ENFORCING REFERENTIAL INTEGRITY WITH THE FOREIGN KEY

A foreign key is a column or set of columns in one table that correspond in exact order to a column or set of columns defined as a primary key in another table. For example, in the PROJECT table, TEAM_LEADER is a foreign key referencing the primary key, EMP_NO in the EMPLOYEE table.

PROJ_ID	TEAM_LEADER	PROJ_NAME	PROJ_DESC	PRODUCT
DGPII	44	Automap	blob data	hardware
VBASE	47	Video database	blob data	software
HWRII	24	Translator upgrade	blob data	software

TABLE 6.3   The PROJECT table

EMP_NO	LAST_NAME	FIRST_NAME	DEPT_NO	JOB_CODE	PHONE_EXT	SALARY
24	Smith	John	100	Eng	4968	64000
48	Carter	Catherine	900	Sales	4967	72500
36	Smith	Jane	600	Admin	4800	37500

TABLE 6.4   The EMPLOYEE table

The primary reason for defining foreign keys is to ensure that data integrity is maintained when more than one table uses the same data: rows in the referencing table must always have corresponding rows in the referenced table.

InterBase enforces *referential integrity* in the following ways:

- The unique or primary key columns must already be defined before you can create the foreign key that references them.

- Referential integrity checks are available in the form of the ON UPDATE and ON DELETE options to the REFERENCES statement. When you create a foreign key by defining a column or table REFERENCES constraint, you can specify what should happen to the foreign key when the referenced primary key changes. The options are:

Action specified	Effect on foreign key
NO ACTION	[Default] The foreign key does not change (can cause the primary key update or delete to fail due to referential integrity checks)
CASCADE	The corresponding foreign key is updated or deleted as appropriate to the new value of the primary key
SET DEFAULT	Every column of the corresponding foreign key is set to its default value; fails if the default value of the foreign key is not found in the primary key
SET NULL	Every column of the corresponding foreign key is set to NULL

TABLE 6.5   Referential integrity check options

- If you do not use the ON UPDATE and ON DELETE options when defining foreign keys, you must make sure that when information changes in one place, it changes in all referencing columns as well. Typically, you write triggers to do this. For example, to change a value in the EMP_NO column of the EMPLOYEE table (the primary key), that value must also be updated in the TEAM_LEADER column of the PROJECT table (the foreign key).

- If you delete a row from a table that is a primary key, you must first delete all foreign keys that reference that row. If you use the ON DELETE CASCADE option when defining the foreign keys, InterBase does this for you.

    Note When you specify SET DEFAULT as the action, the default value used is the one in effect when the referential integrity constraint was defined. When the default for a foreign key column is changed after the referential integrity constraint is set up, the change does not have an effect on the default value used in the referential integrity constraint.

- You cannot add a value to a column defined as a foreign key unless that value exists in the referenced primary key. For example, to enter a value in the TEAM_LEADER column of the PROJECT table, that value must first exist in the EMP_NO column of the EMPLOYEE table.

    The following example specifies that when a value is deleted from a primary key, the corresponding values in the foreign key are set to NULL. When the primary key is updated, the changes are cascaded so that the corresponding foreign key values match the new primary key values.

```
CREATE TABLE PROJECT {
. . .
TEAM LEADER INTEGER REFERENCES EMPLOYEE (EMP_NO)
```

```
    ON DELETE SET NULL
    ON UPDATE CASCADE
. . .};
```

REFERENCING TABLES OWNED BY OTHERS

If you want to create a foreign key that references a table owned by someone else, that owner must first use the GRANT command to grant you REFERENCES privileges on that table. Alternately, the owner can grant REFERENCES privileges to a role and then grant that role to you. See Chapter 13, "Planning Security" and the *Language Reference* for more information on granting privileges to users and roles. See the *Language Reference* for more on creating and dropping roles.

CIRCULAR REFERENCES

When two tables reference each other's foreign keys *and* primary keys, a circular reference exists between the two tables. In the following illustration, the foreign key in the EMPLOYEE table, DEPT_NO, references the primary key, DEPT_NO, in the DEPARTMENT table. Therefore, the primary key, DEPT_NO must be defined in the DEPARTMENT table before it can be referenced by a foreign key in the EMPLOYEE table. In the same manner, EMP_NO, which is the EMPLOYEE table's primary key, must be created before the DEPARTMENT table can define EMP_NO as its foreign key.

FIGURE 6.1   Circular references
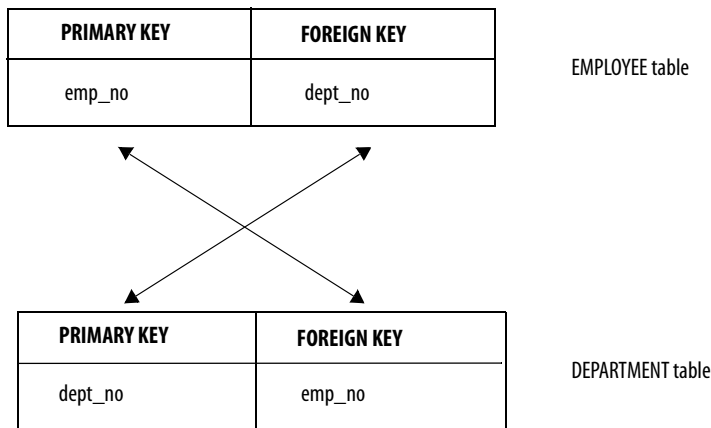
The problem with circular referencing occurs when you try to insert a new row into either table. Inserting a new row into the EMPLOYEE table causes a new value to be inserted into the DEPT_NO (foreign key) column, but you cannot insert a value into the foreign key column unless that value already exists in the DEPT_NO (primary key) column of the DEPARTMENT table. It is also true that you cannot add a new row to the DEPARTMENT table unless the values placed in the EMP_NO (foreign key) column already exist in the EMP_NO (primary key) column of the EMPLOYEE table. Therefore, you are in a deadlock situation because you cannot add a new row to either table!

InterBase gets around the problem of circular referencing by allowing you to insert a NULL value into a foreign key column before the corresponding primary key value exists. The following example illustrates the sequence for inserting a new row into each table:

- Insert a new row into the EMPLOYEE table by placing "1" in the EMP_NO primary key column, and a NULL in the DEPT_NO foreign key column.

- Insert a new row into the DEPARTMENT table, placing "2" in the DEPT_NO primary key column, and "1" in the foreign key column.

- Use ALTER TABLE to modify the EMPLOYEE table. Change the DEPT_NO column from NULL to "2."

HOW TO DECLARE CONSTRAINTS

When declaring a table-level or a column-level constraint, you can optionally name the constraint using the CONSTRAINT clause. If you omit the CONSTRAINT clause, InterBase generates a unique system constraint name which is stored in the system table, RDB$RELATION_CONSTRAINTS.

TIP To ensure that the constraint names are visible in RDB$RELATION_CONSTRAINTS, commit your transaction before trying to view the constraint in the RDB$RELATION_CONSTRAINTS system table.

The syntax for a column-level constraint is:

```
<col_constraint> = [CONSTRAINT constraint] <constraint_def>
    [<col_constraint> ...]
<constraint_def> = {UNIQUE | PRIMARY KEY
    | CHECK (<search_condition>)
    | REFERENCES other_table [(other_col [, other_col …])]
        [ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
        [ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
}
```

The syntax for a table-level constraint is:

```
<tconstraint> = [CONSTRAINT constraint] <tconstraint_def>
    [<tconstraint> ...]
```

```
<tconstraint_def> = {{PRIMARY KEY | UNIQUE} (col [, col …])
    | FOREIGN KEY (col [, col …]) REFERENCES other_table
        [ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
        [ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
    | CHECK (<search_condition>)}}
```

TIP   Although naming a constraint is optional, assigning a descriptive name with the CONSTRAINT clause can make the constraint easier to find for changing or dropping, and easier to find when its name appears in a constraint violation error message.

The following statement illustrates how to create a simple, column-level PRIMARY KEY constraint:

```
CREATE TABLE COUNTRY
    (COUNTRY COUNTRYNAME NOT NULL PRIMARY KEY,
    CURRENCY VARCHAR(10) NOT NULL);
```

The next example illustrates how to create a UNIQUE constraint at both the column level and the table level:

```
CREATE TABLE STOCK
(MODEL SMALLINT NOT NULL UNIQUE,
MODELNAME CHAR(10) NOT NULL,
ITEMID INTEGER NOT NULL, CONSTRAINT MOD_UNIQUE UNIQUE (MODELNAME,
ITEMID));
```

▶ *Defining a* CHECK *constraint*

You can specify a condition or requirement on a data value at the time the data is entered by applying a CHECK constraint to a column. Use CHECK constraints to enforce a condition that must be true before an insert or an update to a column or group of columns is allowed. The search condition verifies whether the value entered falls within a certain permissible range, or matches it to one value in a list of values. The search condition can also compare the value entered with data values in other columns.

Note A CHECK constraint guarantees data integrity only when the values being verified are *in the same row* that is being inserted and deleted. If you try to compare values in different rows of the same table or in different tables, another user could later modify those values, thus invalidating the original CHECK constraint that was applied at insertion time.

In the following example, the CHECK constraint is guaranteed to be satisfied:

```
CHECK (VALUE (COL_1 > COL_2));
INSERT INTO TABLE_1 (COL_1, COL_2) VALUES (5,6);
```

The syntax for creating a CHECK constraint is:

```
CHECK (<search condition>);
<search_condition> = {<val> <operator>
{<val> | (<select_one>)}
| <val> [NOT] BETWEEN <val> AND <val>
| <val> [NOT] LIKE <val> [ESCAPE <val>]
| <val> [NOT] IN (<val> [, <val> ...] | <select_list>)
| <val> IS [NOT] NULL
| <val> {[NOT] {= | < | >} | >= | <=}
        {ALL | SOME | ANY} (<select_list>)
| EXISTS (<select_expr>)
| SINGULAR (<select_expr>)
| <val> [NOT] CONTAINING <val>
| <val> [NOT] STARTING [WITH] <val>
| (<search_condition>)
| NOT <search_condition>
| <search_condition> OR <search_condition>
| <search_condition> AND <search_condition>}
```

When creating CHECK constraints, the following restrictions apply:

- A CHECK constraint cannot reference a domain.

- A column can have only one CHECK constraint.

- On a domain-based column, you cannot override a CHECK constraint imposed by the domain with a local CHECK constraint. A column based on a domain can add *additional* CHECK constraints to the local column definition.

In the next example, a CHECK constraint is placed on the SALARY domain. VALUE is a placeholder for the name of a column that will eventually be based on the domain.

```
CREATE DOMAIN BUDGET
AS NUMERIC(12,2)
DEFAULT 0
CHECK (VALUE > 0);
```

The next statement illustrates PRIMARY KEY, FOREIGN KEY, CHECK, and the referential integrity constraints ON UPDATE and ON DELETE. The PRIMARY KEY constraint is based on three columns, so it is a table-level constraint. The FOREIGN KEY column (JOB_COUNTRY) references the PRIMARY KEY column (COUNTRY) in the table, COUNTRY. When the primary key changes, the ON UPDATE and ON DELETE clauses guarantee that the foreign key column will reflect the changes. This example also illustrates using domains (JOBCODE, JOBGRADE, COUNTRYNAME, SALARY) and a CHECK constraint to define columns:

```
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(400,1),
LANGUAGE_REQ VARCHAR(15) [5],
PRIMARY KEY (JOB_CODE, JOB_GRADE, JOB_COUNTRY),
FOREIGN KEY (JOB_COUNTRY) REFERENCES COUNTRY (COUNTRY)
    ON UPDATE CASCADE
    ON DELETE CASCADE,
CHECK (MIN_SALARY < MAX_SALARY));
```

Altering tables

Use ALTER TABLE to modify the structure of an existing table. ALTER TABLE allows you to:

- Add a new column to a table.
- Drop a column from a table.
- Drop integrity constraints from a table or column.

You can perform any number of the above operations with a single ALTER TABLE statement. A table can be altered by its creator, the SYSDBA user, and any users with operating system root privileges.

Before using ALTER TABLE

Before modifying or dropping columns in a table, you need to do three things:

1. Make sure you have the proper database privileges.

2. Save the existing data.

3. Drop any constraints on the column.

▶ *Saving existing data*

Before modifying an existing column definition using ALTER TABLE, you must preserve existing data, or it will be lost.

Preserving data in a column and modifying the definition for a column, is a six-step process:

1. Add a temporary column to the table whose definition mirrors the current column to be changed.

2. Copy the data from the column to be changed to the temporary column.

3. Drop the column to be changed.

4. Add a new column definition, giving it the same name as the dropped column.

5. Copy the data from the temporary column to the new column.

6. Drop the temporary column.

For example, suppose the EMPLOYEE table contains a column, OFFICE_NO, defined to hold a datatype of CHAR(3), and suppose that the size of the column needs to be increased by one. The following numbered sequence describes each step and provides sample code:

1. First, create a temporary column to hold the data in OFFICE_NO during the modification process:

   ```
   ALTER TABLE EMPLOYEE ADD TEMP_NO CHAR(3);
   ```

2. Move existing data from OFFICE_NO to TEMP_NO to preserve it:

   ```
   UPDATE EMPLOYEE
   SET TEMP_NO = OFFICE_NO;
   ```

3. After the data is moved, drop the OFFICE_NO column:

   ```
   ALTER TABLE DROP OFFICE_NO;
   ```

4. Add a new column definition for OFFICE_NO, specifying the datatype and new size:

   ```
   ALTER TABLE ADD OFFICE_NO CHAR(4);
   ```

5. Move the data from TEMP_NO to OFFICE_NO:

   ```
   UPDATE EMPLOYEE
   SET OFFICE_NO = TEMP_NO;
   ```

6. Finally, drop the TEMP_NO column:

   ```
   ALTER TABLE DROP TEMP_NO;
   ```

▸ *Dropping columns*

Before attempting to drop or modify a column, you should be aware of the different ways that ALTER TABLE can fail:

▪ The person attempting to alter data does not have the required privileges.

- Current data in a table violates a PRIMARY KEY or UNIQUE constraint definition added to the table; there is duplicate data in columns that you are trying to define as PRIMARY KEY or UNIQUE.

- The column to be dropped is part of a UNIQUE, PRIMARY, or FOREIGN KEY constraint.

- The column is used in a CHECK constraint. When altering a column based on a domain, you can supply an additional CHECK constraint for the column. Changes to tables that contain CHECK constraints with subqueries can cause constraint violations.

- The column is used in another view, trigger, or in the value expression of a computed column.

IMPORTANT   You must drop the constraint or computed column before dropping the table column. You cannot drop PRIMARY KEY and UNIQUE constraints if they are referenced by FOREIGN KEY constraints. In this case, drop the FOREIGN KEY constraint before dropping the PRIMARY KEY or UNIQUE key it references. Finally, you can drop the column.

IMPORTANT   When you alter or drop a column, all data stored in it is lost.

Using ALTER TABLE

ALTER TABLE allows you to make the following changes to an existing table:

- Add new column definitions. To create a column using an existing name, you must drop existing column definitions before adding new ones.

- Add new table constraints. To create a constraint using an existing name, you must drop existing constraints with that name before adding a new one.

- Drop existing column definitions without adding new ones.

- Drop existing table constraints without adding new ones.

For a detailed specification of ALTER TABLE syntax, see the *Language Reference*.

▶ *Adding a new column to a table*

The syntax for adding a column with ALTER TABLE is:

```
ALTER TABLE table ADD <col_def>

<col_def> = col {<datatype> | [COMPUTED [BY] (<expr>) | domain}
    [DEFAULT {literal | NULL | USER}]
    [NOT NULL] [<col_constraint>]
    [COLLATE collation]
```

```
<col_constraint> = [CONSTRAINT constraint] <constraint_def>
    [<col_constraint>]


<constraint_def> = {PRIMARY KEY | UNIQUE
    | CHECK (<search_condition>)
    | REFERENCES other_table [(other_col [, other_col …])]
        [ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
        [ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]}
```

For the complete syntax of ALTER TABLE, see the *Language Reference*.

For example, the following statement adds a column, EMP_NO, to the EMPLOYEE table using the EMPNO domain:

```
ALTER TABLE EMPLOYEE ADD EMP_NO EMPNO NOT NULL;
```

You can add multiple columns to a table at the same time. Separate column definitions with commas. For example, the following statement adds two columns, EMP_NO, and FULL_NAME, to the EMPLOYEE table. FULL_NAME is a computed column, a column that derives it values from calculations based on two other columns already defined for the EMPLOYEE table:

```
ALTER TABLE EMPLOYEE
ADD EMP_NO EMPNO NOT NULL,
ADD FULL_NAME COMPUTED BY (LAST_NAME || ', ' || FIRST_NAME);
```

You can also define integrity constraints for columns that you add to the table. For example, the next statement adds two columns, CAPITAL and LARGEST_CITY, to the COUNTRY table, and defines a UNIQUE constraint on CAPITAL:

```
ALTER TABLE COUNTRY
ADD CAPITAL VARCHAR(25) UNIQUE,
ADD LARGEST_CITY VARCHAR(25) NOT NULL;
```

▶ *Adding new table constraints*

You can use ALTER TABLE to add a new table-level constraint. The syntax is:

```
ALTER TABLE name ADD [CONSTRAINT constraint] <tconstraint_opt>;
```

where *tconstraint_opt* is a PRIMARY KEY, FOREIGN KEY, UNIQUE, or CHECK constraint. For example:

```
ALTER TABLE EMPLOYEE
ADD CONSTRAINT DEPT_NO UNIQUE(PHONE_EXT);
```

▶ *Dropping an existing column from a table*

You can use ALTER TABLE to delete a column definition and its data from a table. A column can be dropped only by the owner of the table. If another user is accessing a table when you attempt to drop a column, the other user's transaction will continue to have access to the table until that transaction completes. InterBase postpones the drop until the table is no longer in use.

The syntax for dropping a column with ALTER TABLE is:

```
ALTER TABLE name DROP colname [, colname ...];
```

For example, the following statement drops the EMP_NO column from the EMPLOYEE table:

```
ALTER TABLE EMPLOYEE DROP EMP_NO;
```

Multiple columns can be dropped with a single ALTER TABLE statement.

```
ALTER TABLE EMPLOYEE
DROP EMP_NO,
DROP FULL_NAME;
```

IMPORTANT   You cannot delete a column that is part of a UNIQUE, PRIMARY KEY, or FOREIGN KEY constraint. In the previous example, EMP_NO is the PRIMARY KEY for the EMPLOYEE table, so you cannot drop this column unless you first drop the PRIMARY KEY constraint.

▶ *Dropping existing constraints from a column*

You must drop constraints from a column in the correct sequence. See the following CREATE TABLE example. Because there is a foreign key in the PROJECT table that references the primary key (EMP_NO) of the EMPLOYEE table, you must first drop the foreign key reference before you can drop the PRIMARY KEY constraint in the EMPLOYEE table.

```
CREATE TABLE PROJECT
    (PROJ_ID PROJNO NOT NULL,
    PROJ_NAME VARCHAR(20) NOT NULL UNIQUE,
    PROJ_DESC BLOB(800,1),
    TEAM_LEADER EMPNO,
    PRODUCT PRODTYPE,

    PRIMARY KEY (PROJ_ID),
    CONSTRAINT TEAM_CONSTRT FOREIGN KEY (TEAM_LEADER) REFERENCES
    EMPLOYEE (EMP_NO));
```

The proper sequence is:

```
ALTER TABLE PROJECT
DROP CONSTRAINT TEAM_CONSTRT;
```

```
ALTER TABLE EMPLOYEE
DROP CONSTRAINT EMP_NO_CONSTRT;
ALTER TABLE EMPLOYEE
DROP EMP_NO;
```

Note Constraint names are in the system table, RDB$RELATION_CONSTRAINTS.

In addition, you cannot delete a column if it is referenced by another column's CHECK constraint. To drop the column, first drop the CHECK constraint, then drop the column.

▶ *Summary of* ALTER TABLE *arguments*

When you use ALTER TABLE to add column definitions and constraints, you can specify all of the same arguments that you use in CREATE TABLE; all column definitions, constraints, and datatype arguments are the same, with the exception of the *operation* argument. The following operations are available for ALTER TABLE.

▪ Add a new column definition with ADD *col_def.*

▪ Add a new table constraint with ADD *table_constraint.*

▪ Drop an existing column with DROP *col.*

▪ Drop an existing constraint with DROP CONSTRAINT *constraint.*

Dropping tables

Use DROP TABLE to delete an entire table from the database.

Note If you want to drop columns from a table, use ALTER TABLE.

Dropping a table

Use DROP TABLE to remove a table's data, metadata, and indexes from a database. It also drops any triggers that are based on the table. A table can be dropped by its creator, the SYSDBA user, or any user with operating system root privileges.

You cannot drop a table that is referenced in a computed column, a view, integrity constraint, or stored procedure. You cannot drop a table that is being used by an active transaction until the table is no longer in use.

DROP TABLE fails and returns an error if:

▪ The person who attempts to drop the table is not the owner of the table.

- The table is in use when the drop is attempted. The drop is postponed until the table is no longer in use.
- The table has a UNIQUE or PRIMARY KEY defined for it, and the PRIMARY KEY is referenced by a FOREIGN KEY in another table. First drop the FOREIGN KEY constraints in the other table, then drop the table.
- The table is used in a view, trigger, stored procedure, or computed column. Remove the other elements before dropping the table.
- The table is referenced in another table's CHECK constraint.

  Note DROP TABLE does not delete external tables; it removes the table definition from the database. You must explicitly delete the external file.

DROP TABLE syntax

```
DROP TABLE name;
```

The following statement drops the table, COUNTRY:

```
DROP TABLE COUNTRY;
```

7

Working with Indexes

This chapter explains the following:

- Index basics
- When and how to create indexes
- How to improve index performance

Index basics

An *index* is a mechanism that is used to speed the retrieval of records in response to certain search conditions, and to enforce uniqueness constraints on columns. Just as you search an index in a book for a list of page numbers to quickly find the pages that you want to read, a database index serves as a logical pointer to the physical location (address) of a row in a table. An index stores each value of the indexed column or columns along with pointers to all of the disk blocks that contain rows with that column value.

When executing a query, the InterBase engine first checks to see if any indexes exist for the named tables. It then determines whether it is more efficient to scan the entire table, or to use an existing index to process the query. If the engine decides to use an index, it searches the index to find the key values requested, and follows the pointers to locate the rows in the table containing the values.

Data retrieval is fast because the values in the index are ordered, and the index is relatively small. This allows the system to quickly locate the key value. Once the key value is found, the system follows the pointer to the physical location of the associated data. Using an index typically requires fewer page fetches than a sequential read of every row in the table.

An index can be defined on a single column or on multiple columns of a table. Multi-column indexes can be used for single-column lookups, as long as the column that is being retrieved is the first in the index.

When to index

An index on a column can mean the difference between an immediate response to a query and a long wait, as the length of time it takes to search the whole table is directly proportional to the number of rows in the table. So why not index every column? The main drawbacks are that indexes consume additional disk space, and inserting, deleting, and updating data takes longer on indexed columns than on non-indexed columns. The reason is that the index must be updated each time the data in the indexed column changes, and each time a row is added to or deleted from the table.

Nevertheless, the overhead of indexes is usually outweighed by the boost in performance for data retrieval queries. You *should* create an index on a column when:

- Search conditions frequently reference the column.
- Join conditions frequently reference the column.
- ORDER BY statements frequently use the column to sort data.

You do *not* need to create an index for:

- Columns that are seldom referenced in search conditions.
- Frequently updated non-key columns.
- Columns that have a small number of possible values.

Creating indexes

Indexes are either created by the user with the CREATE INDEX statement, or they are created automatically by the system as part of the CREATE TABLE statement. InterBase allows users to create as many as 64 indexes on a given table. To create indexes you must have authority to connect to the database.

Note To see all indexes defined for the current database, use the **isql** command SHOW INDEX. To see all indexes defined for a specific table, use the command, SHOW INDEX *tablename*. To view information about a specific index, use SHOW INDEX *indexname*.

InterBase automatically generates system-level indexes on a column or set of columns when tables are defined using PRIMARY KEY, FOREIGN KEY, and UNIQUE constraints. Indexes on PRIMARY KEY and FOREIGN KEY constraints preserve referential integrity.

Using CREATE INDEX

The CREATE INDEX statement creates an index on one or more columns of a table. A single-column index searches only one column in response to a query, while a multi-column index searches one or more columns. Options specify:

- The sort order for the index.
- Whether duplicate values are allowed in the indexed column.

Use CREATE INDEX to improve speed of data access. For faster response to queries that require sorted values, use the index order that matches the query's ORDER BY clause. Use an index for columns that appear in a WHERE clause to speed searching.

To improve index performance, use SET STATISTICS to recompute index selectivity, or rebuild the index by making it inactive, then active with sequential calls to ALTER INDEX. For more information about improving performance, see "Using SET STATISTICS" on page 105.

The syntax for CREATE INDEX is:

```
CREATE [UNIQUE] [ASC[ENDING] | DESC[ENDING]]
INDEX index ON table (col [, col ...]);
```

▸ *Preventing duplicate entries*

No two rows can be alike when a UNIQUE index is specified for a column or set of columns. The system checks for duplicate values when the index is created, and each time a row is inserted or updated. InterBase automatically creates a UNIQUE index on a PRIMARY KEY column, forcing the values in that column to be unique identifiers for the row. Unique indexes only make sense when uniqueness is a characteristic of the data itself. For example, you would not define a unique index on a LAST_NAME column because there is a high probability for duplication. Conversely, a unique index is a good idea on a column containing a social security number.

To define an index that disallows duplicate entries, include the UNIQUE keyword in CREATE INDEX. The following statement creates a unique ascending index (PRODTYPEX) on the PRODUCT and PROJ_NAME columns of the PROJECT table:

```
CREATE UNIQUE INDEX PRODTYPEX ON PROJECT (PRODUCT, PROJ_NAME);
```

TIP   InterBase does not allow you to create a unique index on a column that already contains
duplicate values. Before defining a UNIQUE index, use a SELECT statement to ensure there
are no duplicate keys in the table. For example:

```
SELECT PRODUCT, PROJ_NAME FROM PROJECT
GROUP BY PRODUCT, PROJ_NAME
HAVING COUNT(*) > 1;
```

▶ *Specifying index sort order*

Specify a direction (low to high or high to low) by using the ASCENDING or DESCENDING
keyword. By default, InterBase creates indexes in ascending order. To make a descending
index on a column or group of columns, use the DESCENDING keyword to define the index.
The following statement creates a descending index (DESC_X) on the CHANGE_DATE
column of the SALARY_HISTORY table:

```
CREATE DESCENDING INDEX DESC_X ON SALARY_HISTORY (CHANGE_DATE);
```

Note To retrieve indexed data from this table in descending order, use ORDER BY
CHANGE_DATE DESCENDING in the SELECT statement.

If you intend to use both ascending and descending sort orders on a particular column, define
both an ascending and a descending index for the same column. The following example
illustrates this:

```
CREATE ASCENDING INDEX ASCEND_X ON SALARY_HISTORY (CHANGE_DATE);
CREATE DESCENDING INDEX DESC_X ON SALARY_HISTORY (CHANGE_DATE);
```

When to use a multi-column index

The main reason to use a multi-column index is to speed up queries that often access the
same set of columns. You do not have to create the query with the exact column list that is
defined in the index. InterBase will use a subset of the components of a multi-column index
to optimize a query if the:

- Subset of columns used in the ORDER BY clause begins with the first column in the
multi-column index. Unless the query uses all prior columns in the list, InterBase cannot use
that index to optimize the search. For example, if the index column list is A1, A2, and A3, a
query using A1 and A2 would be optimized using the index, but a query using A2 and A3
would not.

- Order in which the query accesses the columns in an ORDER BY clause matches the order of
the column list defined in the index. (The query would not be optimized if its column list
were A2, A1.)

TIP   If you expect to issue frequent queries against a table where the queries use the OR operator, it is better to create a single-column index for each condition. Since multi-column indices are sorted hierarchically, a query that is looking for any one of two or more conditions would, of course, have to search the whole table, losing the advantage of an index.

Examples using multi-column indexes

The first example creates a multi-column index, NAMEX, on the EMPLOYEE table:

```
CREATE INDEX NAMEX ON EMPLOYEE (LAST_NAME, FIRST_NAME);
```

The following query will be optimized against the index because the ORDER BY clause references all of the indexed columns in the correct order:

```
SELECT LAST_NAME, SALARY FROM EMPLOYEE
    WHERE SALARY > 40000
    ORDER BY LAST_NAME, FIRST_NAME;
```

The next query will also process the following query with an index search (using LAST_NAME from NAMEX) because although the ORDER BY clause only references one of the indexed columns (LAST_NAME), it does so in the correct order.

```
SELECT LAST_NAME, SALARY FROM EMPLOYEE
    WHERE SALARY > 40000
    ORDER BY LAST_NAME;
```

Conversely, the following query will *not* be optimized against the index because the ORDER BY clause uses FIRST_NAME, which is not the first indexed column in the NAMEX column list.

```
SELECT LASTNAME, SALARY FROM EMP
    WHERE SALARY > 40000
    ORDER BY FIRST_NAME;
```

The same rules that apply to the ORDER BY clause also apply to queries containing a WHERE clause. The next example creates a multi-column index for the PROJECT table:

```
CREATE UNIQUE INDEX PRODTYPEX ON PROJECT (PRODUCT, PROJ_NAME);
```

The following query will be optimized against the PRODTYPEX index because the WHERE clause references the first indexed column (PRODUCT) of the index:

```
SELECT * FROM PROJECT
    WHERE PRODUCT ="software";
```

Conversely, the next query will not be optimized against the index because PROJ_NAME is not the first indexed column in the column list of the PRODTYPEX index:

```
SELECT * FROM PROJECT
```

```
WHERE PROJ_NAME ="InterBase 4.0";
```

Improving index performance

Indexes can become unbalanced after many changes to the database. When this happens, performance can be improved using one of the following methods:

- Rebuild the index with ALTER INDEX.
- Recompute index selectivity with SET STATISTICS.
- Delete and recreate the index with DROP INDEX and CREATE INDEX.
- Back up and restore the database with **gbak**.

Using ALTER INDEX

The ALTER INDEX statement deactivates and reactivates an index. Deactivating and reactivating an index is useful when changes in the distribution of indexed data cause the index to become unbalanced.

To rebuild the index, first use ALTER INDEX INACTIVE to deactivate the index, then ALTER INDEX ACTIVE to reactivate it again. This method recreates and balances the index.

Note You can also rebuild an index by backing up and restoring the database with the **gbak** utility. **gbak** stores only the definition of the index, not the data structure, so when you restore the database, **gbak** rebuilds the indexes.

TIP Before inserting a large number of rows, deactivate a table's indexes during the insert, then reactivate the index to rebuild it. Otherwise, InterBase incrementally updates the index each time a single row is inserted.

The syntax for ALTER INDEX is:

```
ALTER INDEX name {ACTIVE | INACTIVE};
```

The following statements deactivate and reactivate an index to rebuild it:

```
ALTER INDEX BUDGETX INACTIVE;
ALTER INDEX BUDGETX ACTIVE;
```

Note The following restrictions apply to altering an index:

- In order to alter an index, you must be the creator of the index, a SYSDBA user, or a user with operating system root privileges.

- You cannot alter an index if it is in use in an active database. An index is in use if it is currently being used by a compiled request to process a query. All requests using an index must be released to make it available.

- You cannot alter an index that has been defined with a UNIQUE, PRIMARY KEY, or FOREIGN KEY constraint. If you want to modify the constraints, you must use ALTER TABLE. For more information about ALTER TABLE, see the *Language Reference*.

- You cannot use ALTER INDEX to add or drop index columns or keys. Use DROP INDEX to delete the index and then redefine it with CREATE INDEX.

Using SET STATISTICS

For tables where the number of duplicate values in indexed columns radically increases or decreases, periodically recomputing index selectivity can improve performance. SET STATISTICS recomputes the selectivity of an index.

Index selectivity is a calculation that is made by the InterBase optimizer when a table is accessed, and is based on the number of distinct rows in a table. It is cached in memory, where the optimizer can access it to calculate the optimal retrieval plan for a given query.

The syntax for SET STATISTICS is:

```
SET STATISTICS INDEX name;
```

The following statement recomputes the selectivity for an index:

```
SET STATISTICS INDEX MINSALX;
```

Note The following restrictions apply to the SET STATISTICS statement:

- In order to use SET STATISTICS, you must be the creator of the index, a SYSDBA user, or a user with operating system root privileges.

- SET STATISTICS does not rebuild an index. To rebuild an index, use ALTER INDEX.

Using DROP INDEX

DROP INDEX removes a user-defined index from the database. System-defined indexes, such as those created on columns defined with UNIQUE, PRIMARY KEY, and FOREIGN KEY constraints cannot be dropped.

To alter an index, first use the DROP INDEX statement to delete the index, then use the CREATE INDEX statement to recreate the index (using the same name) with the desired characteristics.

The syntax for DROP INDEX is:

```
DROP INDEX name;
```

The following statement deletes an index:

```
DROP INDEX MINSALX;
```

Note The following restrictions apply to dropping an index:

- To drop an index, you must be the creator of the index, a SYSDBA user, or a user with operating system root privileges.

- An index in use cannot be dropped until it is no longer in use. If you try to alter or drop an index while transactions are being processed, the results depend on the type of transaction in operation. In a WAIT transaction, the ALTER INDEX or DROP INDEX operation waits until the index is not in use. In a NOWAIT transaction, InterBase returns an error.

- If an index was automatically created by the system on a column having a UNIQUE, PRIMARY KEY, or FOREIGN KEY constraint, you cannot drop the index. To drop an index on a column defined with those constraints, drop the constraint, the constrained column, or the table. To modify the constraints, use ALTER TABLE. For more information about ALTER TABLE, see the *Language Reference*.

8

Working with Views

This chapter describes:

- What views are and the reasons for using them.
- How to create and drop views.
- How to modify data through a view.

Introduction

Database users typically need to access a particular subset of the data that is stored in the database. Further, the data requirements within an individual user or group are often quite consistent. Views provide a way to create a customized version of the underlying tables that display only the clusters of data that a given user or group of users is interested in.

Once a view is defined, you can display and operate on it as if it were an ordinary table. A view can be derived from one or more tables, or from another view. Views look just like ordinary database tables, but they are not physically stored in the database. The database stores only the view definition, and uses this definition to filter the data when a query referencing the view occurs.

IMPORTANT It is important to understand that creating a view does not generate a *copy* of the data stored in another table; when you change the data through a view, you are changing the data in the actual underlying tables. Conversely, when the data in the base tables is changed directly, the views that were derived from the base tables are automatically updated to reflect the changes. Think of a view as a movable "window" or frame through which you can see the actual data. The data definition is the "frame." For restrictions on operations using views, see "Types of views: read-only and updatable" on page 111.

A view can be created from:

- A vertical subset of columns from a single table. For example, the table, JOB, in the *employee.gdb* database has 8 columns: JOB_CODE, JOB_GRADE, JOB_COUNTRY, JOB_TITLE, MIN_SALARY, MAX_SALARY, JOB_REQUIREMENT, and LANGUAGE_REQ. The following view displays a list of salary ranges (subset of columns) for all jobs (all rows) in the JOB table:

```
CREATE VIEW JOB_SALARY_RANGES AS
SELECT JOB_CODE, MIN_SALARY, MAX_SALARY
FROM JOB;
```

- A horizontal subset of rows from a single table. The next view displays all of the columns in the JOB table, but only the subset of rows where the MAX_SALARY is less than $15,000:

```
CREATE VIEW LOW_PAY AS
SELECT *
FROM JOB
WHERE MAX_SALARY < 15000;
```

- A combined vertical and horizontal subset of columns and rows from a single table. The next view displays only the JOB_CODE and JOB_TITLE columns and only those jobs where MAX_SALARY is less than $15,000:

```
CREATE VIEW ENTRY_LEVEL_JOBS AS
SELECT JOB_CODE, JOB_TITLE,
FROM JOB
WHERE MAX_SALARY < 15000;
```

- A subset of rows and columns from multiple tables (joins). The next example shows a view created from both the JOB and EMPLOYEE tables. The EMPLOYEE table contains 11 columns: EMP_NO, FIRST_NAME, LAST_NAME, PHONE_EXT, HIRE_DATE, DEPT_NO, JOB_CODE, JOB_GRADE, JOB_COUNTRY, SALARY, FULL_NAME. It displays two columns from the JOB table, and two columns from the EMPLOYEE table, and returns only the rows where SALARY is less than $15,000:

```
CREATE VIEW ENTRY_LEVEL_WORKERS AS
SELECT JOB_CODE, JOB_TITLE, FIRST_NAME, LAST_NAME
FROM JOB, EMPLOYEE
WHERE JOB.JOB_CODE = EMPLOYEE.JOB_CODE AND SALARY < 15000;
```

Advantages of views

The main advantages of views are:

- Simplified access to the data. Views enable you to encapsulate a subset of data from one or more tables to use as a foundation for future queries without requiring you to repeat the same set of SQL statements to retrieve the same subset of data.

- Customized access to the data. Views provide a way to tailor the database to suit a variety of users with dissimilar skills and interests. You can focus on the information that specifically concerns you without having to process extraneous data.

- Data independence. Views protect users from the effects of changes to the underlying database structure. For example, if the database administrator decides to split one table into two, a view can be created that is a join of the two new tables, thus shielding the users from the change.

- Data security. Views provide security by restricting access to sensitive or irrelevant portions of the database. For example, you might be able to look up job information, but not be able to see associated salary information.

Creating views

The CREATE VIEW statement creates a virtual table based on one or more underlying tables in the database. You can perform select, project, join, and union operations on views just as if they were tables.

The user who creates a view is its owner and has all privileges for it, including the ability to GRANT privileges to other users, triggers, and stored procedures. A user can be granted privileges to a view without having access to its base tables.

The syntax for CREATE VIEW is:

```
CREATE VIEW name [(view_col [, view_col …])]
AS <select> [WITH CHECK OPTION];
```

Note You cannot define a view that is based on the result set of a stored procedure.

Specifying view column names

▪ *view_col* names one or more columns for the view. Column names are optional unless the view includes columns based on expressions. When specified, view column names correspond in order and number to the columns listed in *select*, so you must specify view column names for every column selected, or do not specify names at all.

▪ Column names must be unique among all column names in the view. If column names are not specified, the view takes the column names from the underlying table by default.

▪ If the view definition includes an expression, *view_col* names are required. A *view_col* definition can contain one or more columns based on an expression.

Note isql does not support view definitions containing UNION clauses. You must write an embedded application to create this type of view.

Using the SELECT statement

The SELECT statement specifies the selection criteria for the rows to be included in the view. SELECT does the following:

▪ Lists the columns to be included from the base table. When SELECT * is used rather than a column list, the view contains all of the column names from the base table, and displays them in the order in which they appear in the base table. The following example creates a view, MY_VIEW, that contains all of the columns in the EMPLOYEE table:

```
CREATE VIEW MY_VIEW AS
SELECT * FROM EMPLOYEE;
```

▪ Identifies the source tables in the FROM clause. In the MY_VIEW example, EMPLOYEE is the source table.

▪ Specifies, if needed, row selection conditions in a WHERE clause. In the next example, only the employees that work in the USA are included in the view:

```
CREATE VIEW USA_EMPLOYEES AS
SELECT * FROM EMPLOYEE
WHERE JOB_COUNTRY = "USA";
```

▪ If WITH CHECK OPTION is specified, it prevents INSERT or UPDATE operations on an updatable view if the operation violates the search condition specified in the WHERE clause. For more information about using this option, see "Using WITH CHECK OPTION" on page 113. For an explanation of updatable views, see "Types of views: read-only and updatable" on page 111.

IMPORTANT   When creating views, the SELECT statement cannot include an ORDER BY clause.

Using expressions to define columns

An expression can be any SQL statement that performs a comparison or computation, and returns a single value. Examples of expressions are concatenating character strings, performing computations on numeric data, doing comparisons using comparison operators (<, >, <=, and so on) or Boolean operators (AND, OR, NOT). The expression must return a single value, and cannot be an array or return an array. Any columns used in the value expression must exist before the expression can be defined.

For example, suppose you want to create a view that displays the salary ranges for all jobs that pay at least $60,000. The view, GOOD_JOB, based on the JOB table, selects the pertinent jobs and their salary ranges:

```
CREATE VIEW GOOD_JOB (JOB_TITLE, STRT_SALARY, TOP_SALARY) AS
SELECT JOB_TITLE, MIN_SALARY, MAX_SALARY FROM JOB
WHERE MIN_SALARY > 60000;
```

Suppose you want to create a view that assigns a hypothetical 10% salary increase to all employees in the company. The next example creates a view that displays all of the employees and their new salaries:

```
CREATE VIEW 10%_RAISE (EMPLOYEE, NEW_SALARY) AS
SELECT EMP_NO, SALARY *1.1 FROM EMPLOYEE;
```

Note Remember, unless the creator of the view assigns INSERT or UPDATE privileges, the users of the view cannot affect the actual data in the underlying table.

Types of views: read-only and updatable

When you update a view, the changes are passed through to the underlying tables from which the view was created only if certain conditions are met. If a view meets these conditions, it is *updatable*. If it does not meet these conditions, it is *read-only*, meaning that writes to the view are not passed through to the underlying tables.

Note The terms updatable and read-only refer to how you access the data in the underlying tables, not to whether the view definition can be modified. To modify the view definition, you must drop the view and then recreate it.

A view is updatable if all of the following conditions are met:

- It is a subset of a single table or another updatable view.

- All base table columns excluded from the view definition allow NULL values.

- The view's SELECT statement does not contain subqueries, a DISTINCT predicate, a HAVING clause, aggregate functions, joined tables, user-defined functions, or stored procedures.

If the view definition does not meet all of these conditions, it is considered read-only.

Note Read-only views can be updated by using a combination of user-defined referential constraints, triggers, and unique indexes. For information on how to update read-only views using triggers, see Chapter 10, "Creating Triggers."

▸ *View privileges*

The creator of the view must have the following privileges:

- To create a read-only view, the creator needs SELECT privileges for any underlying tables.
- To create an updatable view, the creator needs ALL privileges to the underlying tables.

For more information on SQL privileges, see Chapter 13, "Planning Security."

▸ *Examples of views*

The following statement creates an updatable view:

```
CREATE VIEW EMP_MNGRS (FIRST, LAST, SALARY) AS
SELECT FIRST_NAME, LAST_NAME, SALARY
FROM EMPLOYEE
WHERE JOB_CODE = "Mngr";
```

The next statement uses a nested query to create a view, so the view is read-only:

```
CREATE VIEW ALL_MNGRS AS
SELECT FIRST_NAME, LAST_NAME, JOB_COUNTRY FROM EMPLOYEE WHERE JOB_COUNTRY
IN (SELECT JOB_COUNTRY FROM JOB
WHERE JOB_TITLE = "manager");
```

The next statement creates a view that joins two tables, and so it is also read-only:

```
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.
```

Inserting data through a view

Rows can be inserted and updated through a view if the following conditions are met:

- The view is updatable
- A user or stored procedure has INSERT privilege for the view
- The view is created using WITH CHECK OPTION

TIP   You can simulate updating a read-only view by writing triggers that perform the appropriate writes to the underlying tables. For an example of this, see "Updating views with triggers" on page 165.

▶ *Using* WITH CHECK OPTION

WITH CHECK OPTION specifies rules for modifying data through views. This option can be included only if the views are updatable. Views that are created using WITH CHECK OPTION enable InterBase to verify that a row inserted or updated through a view can be seen through the view before allowing the operation to succeed. Values can only be inserted through a view for those columns named in the view. InterBase stores NULL values for unreferenced columns.

WITH CHECK OPTION prevents you from inserting or updating values that do not satisfy the search condition specified in the WHERE clause of the CREATE VIEW *select* statement.

▶ *Examples*

Suppose you want to create a view that allows access to information about all departments with budgets between $10,000 and $500,000. The view, SUB_DEPT, is defined as follows:

```
CREATE VIEW SUB_DEPT
(DEPT_NAME, DEPT_NO, SUB_DEPT_NO, LOW_BUDGET)
AS SELECT DEPARTMENT, DEPT_NO, HEAD_DEPT, BUDGET
FROM DEPARTMENT WHERE BUDGET BETWEEN 10000 AND 500000
WITH CHECK OPTION;
```

The SUB_DEPT view references a single table, DEPARTMENT. If you are the creator of the view or have INSERT privileges, you can insert new data into the DEPARTMENT, DEPT_NO, HEAD_DEPT, and BUDGET columns of the base table, DEPARTMENT. WITH CHECK OPTION assures that all values entered through the view fall within the range prescribed for each column in the WHERE clause of the SUB_DEPT view.

The following statement inserts a new row for the Publications Department through the SUB_DEPT view:

```
INSERT INTO SUB_DEPT (DEPT_NAME, DEPT_NO, SUB_DEPT_NO, LOW_BUDGET)
VALUES ("Publications", "7735", "670", 250000);
```

InterBase inserts NULL values for all other columns in the DEPARTMENT base table that are not available directly through the view.

Dropping views

The DROP VIEW statement enables a view's creator to remove a view definition from the database. It does not affect the base tables associated with the view. You can drop a view only if:

- You created the view.

- The view is not used in another view, a stored procedure, or CHECK constraint definition. You must delete the associated database objects before dropping the view.

The syntax for DROP VIEW is:

```
DROP VIEW name;
```

The following statement removes a view definition:

```
DROP VIEW SUB_DEPT;
```

Note You cannot alter a view directly. To change a view, drop it and use the CREATE VIEW statement to create a view with the same name and the features you want.

Working with Stored Procedures

This chapter describes the following:

- How to create, alter, and drop procedures.
- The InterBase procedure and trigger language.
- How to use stored procedures.
- How to create, alter, drop, and raise exceptions.
- How to handle errors.

Overview of stored procedures

A stored procedure is a self-contained program written in InterBase procedure and trigger language, and stored as part of a the database metadata.

Once you have created a stored procedure, you can invoke it directly from an application, or substitute the procedure for a table or view in a SELECT statement. Stored procedures can receive input parameters from and return values to applications.

InterBase 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.

The advantages of using stored procedures include:

- Modular design

- Applications that access the same database can share stored procedures, eliminating duplicate code and reducing the size of the applications

- Streamlined maintenance

- When a procedure is updated, the changes are automatically reflected in all applications that use it without the need to recompile and relink them; applications are compiled and optimized only once for each client

- Improved performance

- Stored procedures are executed by the server, not the client, which reduces network traffic, and improves performance—especially for remote client access

Working with procedures

With **isql**, you can create, alter, and drop procedures and exceptions. Each of these operations is explained in the corresponding sections in this chapter.

There are two ways to create, alter, and drop procedures with **isql**:

- Interactively

- With an input file containing data definition statements

It is usually preferable to use data definition files, because they are easier to modify and provide separate documentation of the procedure. For simple changes to existing procedures or exceptions, the interactive interface can be convenient.

The user who creates a procedure is the owner of the procedure, and can grant the privilege to execute the procedure to other users, triggers, and stored procedures.

Using a data definition file

To create or alter a procedure through a data definition file, follow these steps:

1. Use a text editor to write the data definition file.

2. Save the file.

3. Process the file with **isql**. Use this command:

   ```
   isql -input filename database_name
   ```

   where *filename* is the name of the data definition file and *database_name* is the name of the database to use. Alternatively, from within **isql**, you can process the file using the command:

   ```
   SQL> input filename;
   ```

If you do not specify the database on the command line or interactively, the data definition file must include a statement to create or open a database.

The data definition file can include:

- Statements to create, alter, or drop procedures. The file can also include statements to create, alter, or drop exceptions. Exceptions must be created before they can be referenced in procedures.
- Any other **isql** statements.

Calling stored procedures

Applications can call stored procedures from SQL and DSQL. You can also use stored procedures in **isql**. For more information on calling stored procedures from applications, see the *Programmer's Guide*.

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.
- *Executable procedures* that an application can call directly with the EXECUTE PROCEDURE statement. An executable procedure can optionally 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. Select procedures can return more than one row, so that to the calling program they appear as a table or view. Executable procedures are routines invoked by the calling program, which can optionally return values.

In fact, a single procedure conceivably can be used as a select procedure or as an executable procedure, but in general a procedure is written specifically to be used in a SELECT statement (a select procedure) or to be used in an EXECUTE PROCEDURE statement (an executable procedure).

Privileges for stored procedures

To use a stored procedure, a user must be the creator of the procedure or must be given EXECUTE privilege for it. An extension to the GRANT statement assigns the EXECUTE privilege, and an extension to the REVOKE statement eliminates the privilege.

Stored procedures themselves sometimes need access to tables or views for which a user does not—or should not—have privileges. For more information about granting privileges to users and procedures, see Chapter 13, "Planning Security."

Creating procedures

You can define a stored procedure with the CREATE PROCEDURE statement in **isql**. You cannot create stored procedures in embedded SQL. A stored procedure 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, RETURNS 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 can be many levels of nesting.

IMPORTANT   Because each statement in a stored procedure body must be terminated by a semicolon, you must define a different symbol to terminate the CREATE PROCEDURE statement in **isql**. Use SET TERM before CREATE PROCEDURE to specify a terminator other than a semicolon. After the CREATE PROCEDURE statement, include another SET TERM to change the terminator back to a semicolon.

CREATE PROCEDURE syntax

```
CREATE PROCEDURE name
    [(param datatype [, param datatype …])]
    [RETURNS (param datatype [, param datatype …])]
AS
    <procedure_body>;
<procedure_body> =
    [<variable_declaration_list>]
    <block>
<variable_declaration_list> =
DECLARE VARIABLE var datatype;
[DECLARE VARIABLE var datatype; …]
<block> =
BEGIN
    <compound_statement>
    [<compound_statement> …]
END
```

```
<compound_statement> =
    {<block> | statement;}
```

Argument	Description
name	Name of the procedure. Must be unique among procedure, table, and view names in the database.
param <datatype>	Input parameters that the calling program uses to pass values to the procedure:
	param—Name of the input parameter, unique for variables in the procedure.
	<datatype>—An InterBase datatype.
RETURNS *param <datatype>*	Output parameters that the procedure uses to return values to the calling program:
	param—Name of the output parameter, unique for variables within the procedure.
	<datatype>—An InterBase datatype.
	The procedure returns the values of output parameters when it reaches a SUSPEND statement in the procedure body.
AS	Keyword that separates the procedure header and the procedure body.
DECLARE VARIABLE *var <datatype>*	Declares local variables used only in the procedure. Each declaration must be preceded by DECLARE VARIABLE and followed by a semicolon (;). *var* is the name of the local variable, unique for variables in the procedure.
statement	Any single statement in InterBase procedure and trigger language. Each statement (except BEGIN and END) must be followed by a semicolon (;).

TABLE 9.1   Arguments of the CREATE PROCEDURE statement

Procedure and trigger language

The InterBase procedure and trigger language is a complete programming language for stored procedures and triggers. It includes:

- SQL data manipulation statements: INSERT, UPDATE, DELETE, and singleton SELECT. Cursors are allowed.

- SQL operators and expressions, including UDFs linked with the database server and generators.

- Powerful extensions to SQL, including assignment statements, control-flow statements, context variables, event-posting statements, exceptions, and error-handling statements.

Although stored procedures and triggers are used in different ways and for different purposes, they both use the procedure and trigger language. Both triggers and stored procedures can use any statements in the procedure and trigger language, with some exceptions:

- Context variables are unique to triggers.

- Input and output parameters, and the SUSPEND and EXIT statements, which return values and are unique to stored procedures.

The following table summarizes the language extensions for stored procedures.

Statement	Description
BEGIN … END	Defines a block of statements that executes as one. The BEGIN keyword starts the block; the END keyword terminates it. Neither should be followed by a semicolon.
variable = expression	Assignment statement which assigns the value of *expression* to *variable*, a local variable, input parameter, or output parameter.
/* *comment_text* */	Programmer's comment, where *comment_text* can be any number of lines of text.
EXCEPTION *exception_name*	Raises the named exception. An exception is a user-defined error, which can be handled with WHEN.
EXECUTE PROCEDURE *proc_name* [*var* [, *var* …]] [RETURNING_VALUES *var* [, *var* …]]	Executes stored procedure, *proc_name*, with the input arguments listed following the procedure name, returning values in the output arguments listed following RETURNING_VALUES. Input and output parameters must be variables defined within the procedure. Enables nested procedures and recursion.
EXIT	Jumps to the final END statement in the procedure.
FOR *<select_statement>* DO *<compound_statement>*	Repeats the statement or block following DO for every qualifying row retrieved by *<select_statement>*.
	<select_statement>: a normal SELECT statement, except the INTO clause is required and must come last.

TABLE 9.2   Procedure and trigger language extensions

Statement	Description
<compound_statement>	Either a single statement in procedure and trigger language or a block of statements bracketed by BEGIN and END.
IF (<condition>) THEN <compound_statement> [ELSE <compound_statement>]	Tests <condition> and if it is TRUE, performs the statement or block following THEN. Otherwise, performs the statement or block following ELSE, if present.
	<condition>: a Boolean expression (TRUE, FALSE, or UNKNOWN), generally two expressions as operands of a comparison operator.
POST_EVENT event_name	Posts the event, event_name.
SUSPEND	In a SELECT procedure:
	Suspends execution of procedure until next FETCH is issued by the calling application
	Returns output values, if any, to the calling application.
	Not recommended for executable procedures.
WHILE (<condition>) DO <compound_statement>	While <condition> is TRUE, keep performing <compound_statement>. First <condition> is tested, and if it is TRUE, then <compound_statement> is performed. This sequence is repeated until <condition> is no longer TRUE.
WHEN {<error> [, <error> …] | ANY} DO <compound_statement>	Error-handling statement. When one of the specified errors occurs, performs <compound_statement>. WHEN statements, if present, must come at the end of a block, just before END.
	<error>—EXCEPTION exception_name, SQLCODE errcode or GDSCODE number.
	ANY—Handles any errors.

TABLE 9.2   Procedure and trigger language extensions (*continued*)

▶ *Using* SET TERM *in isql*

Because each statement in a procedure body must be terminated by a semicolon, you must define a different symbol to terminate the procedure body itself. In **isql** include a SET TERM statement before CREATE PROCEDURE to specify a terminator other than a semicolon. After the body of the procedure, include another SET TERM to change the terminator back to a semicolon.

The following example illustrates the use of SET TERM for a procedure. The terminator is temporarily set to a double exclamation point (!!).

```
SET TERM !!;
CREATE PROCEDURE SIMPLE
    AS BEGIN
    . . .
    END!!
SET TERM ;!!
```

There must be a space after SET TERM. Each SET TERM is itself terminated with the current terminator.

▶ *Syntax errors in stored procedures*

InterBase generates errors during parsing if there is incorrect syntax in a CREATE PROCEDURE statement. Error messages look similar to this:

```
Statement failed, SQLCODE = -104
Dynamic SQL Error
-SQL error code = -104
-Token unknown - line 4, char 9
-tmp
```

The line numbers are counted from the beginning of the CREATE PROCEDURE statement, not from the beginning of the data definition file. Characters are counted from the left, and the unknown token indicated is either the source of the error, or immediately to the right of the source of the error. When in doubt, examine the entire line to determine the source of the syntax error.

The procedure header

Everything before AS in the CREATE PROCEDURE statement forms the procedure header. The header contains:

- The name of the stored procedure, which must be unique among procedure and table names in the database.

- An optional list of input parameters and their datatypes. The procedure receives the values of the input parameters from the calling program.

- Optionally, the RETURNS keyword followed by a list of output parameters and their datatypes. The procedure returns the values of the output parameters to the calling program.

▶ *Declaring input parameters*

Use input parameters to pass values from an application to a procedure. Any input parameters are given in a comma-delimited list enclosed in parentheses immediately after the procedure name, as follows:

```
CREATE PROCEDURE name
    (var datatype [, var datatype …])
    . . .
```

Each input parameter declaration has two parts: a name and a datatype. The name of the parameter must be unique within the procedure, and the datatype can be any standard SQL datatype except BLOB and arrays of datatypes. The name of an input parameter need not match the name of any host parameter in the calling program.

Note No more than 1,400 input parameters can be passed to a stored procedure.

▶ *Declaring output parameters*

Use output parameters to return values from a procedure to an application. The RETURNS clause in the procedure header specifies a list of output parameters. The syntax of the RETURNS clause is:

```
. . .
    [RETURNS (var datatype [, var datatype …])]
AS
. . .
```

Each output parameter declaration has two parts: a name and a datatype. The name of the parameter must be unique within the procedure, and the datatype can be any standard SQL datatype except BLOB and arrays.

The procedure body

Everything following the AS keyword in the CREATE PROCEDURE statement forms the procedure body. The body consists of an optional list of local variable declarations followed by a block of statements.

A block is composed of statements in the InterBase procedure and trigger language, bracketed by BEGIN and END. A block can itself include other blocks, so that there can be many levels of nesting.

InterBase procedure and trigger language includes all standard InterBase SQL statements except data definition and transaction statements, plus statements unique to procedure and trigger language.

Features of InterBase procedure and trigger language include:

- Assignment statements, to set values of local variables and input/output parameters.

- SELECT statements, to retrieve column values. SELECT statements must have an INTO clause as the last clause.

- Control-flow statements, such as FOR SELECT … DO, IF … THEN, and WHILE … DO, to perform conditional or looping tasks.

- EXECUTE PROCEDURE statements, to invoke other procedures. Recursion is allowed.

- Comments to annotate procedure code.

- Exception statements, to return error messages to applications, and WHEN statements to handle specific error conditions.

- SUSPEND and EXIT statements, that return control—and return values of output parameters—to the calling application.

▶ BEGIN … END *statements*

Each block of statements in the procedure body starts with a BEGIN statement and ends with an END statement. BEGIN and END are not followed by a semicolon. In **isql**, the final END in the procedure body is followed by the terminator that you specified in the SET TERM statement.

▶ *Using variables*

There are three types of variables that can be used in the body of a procedure:

- Input parameters, used to pass values from an application to a stored procedure.

- Output parameters, used to pass values from a stored procedure back to the calling application.

- Local variables, used to hold values used only within a procedure.

Any of these types of variables can be used in the body of a stored procedure where an expression can appear. They can be assigned a literal value, or assigned a value derived from queries or expression evaluations.

Note In SQL statements, precede variables with a colon (:) to signify that they are variables rather than column names. In procedure and trigger language extension statements, you need not precede variables with a colon.

LOCAL VARIABLES

Local variables are declared and used within a stored procedure. They have no effect outside the procedure.

Local variables must be declared at the beginning of a procedure body before they can be used. Declare a local variable as follows:

```
DECLARE VARIABLE var datatype;
```

where *var* is the name of the local variable, unique within the procedure, and *datatype* is the datatype, which can be any SQL datatype except BLOB or an array. Each local variable requires a separate DECLARE VARIABLE statement, followed by a semicolon (;).

The following header declares the local variable, *any_sales*:

```
CREATE PROCEDURE DELETE_EMPLOYEE (EMP_NUM INTEGER)
AS
    DECLARE VARIABLE ANY_SALES INTEGER;
BEGIN
. . .
```

INPUT PARAMETERS

Input parameters are used to pass values from an application to a procedure. They are declared in a comma-delimited list in parentheses following the procedure name. Once declared, they can be used in the procedure body anywhere an expression can appear.

Input parameters are passed *by value* from the calling program to a stored procedure. This means that if the procedure changes the value of an input parameter, the change has effect only within the procedure. When control returns to the calling program, the input parameter still has its original value.

The following procedure header declares two input parameters, *emp_no* and *proj_id*:

```
CREATE PROCEDURE ADD_EMP_PROJ (EMP_NO SMALLINT, PROJ_ID CHAR(5))
    AS
    . . .
```

For more information on declaring input parameters in stored procedures, see "Declaring input parameters" on page 124.

OUTPUT PARAMETERS

Output parameters are used to return values from a procedure to the calling application. Declare them in a comma-delimited list in parentheses following the RETURNS keyword in the procedure header. Once declared, they can be used in the procedure body anywhere an expression can appear. For example, the following procedure header declares five output parameters, *head_dept*, *department*, *mngr_name*, *title*, and *emp_cnt*:

```
CREATE PROCEDURE ORG_CHART
    RETURNS (HEAD_DEPT CHAR(25), DEPARTMENT CHAR(25),
    MNGR_NAME CHAR(20), TITLE CHAR(5), EMP_CNT INTEGER)
```

If you declare output parameters in the procedure header, the procedure must assign them values to return to the calling application. Values can be derived from any valid expression in the procedure.

For more information on declaring output parameters in stored procedures, see "Declaring output parameters" on page 124.

A procedure returns output parameter values to the calling application with a SUSPEND statement. For more information about SUSPEND, see "Using SUSPEND, EXIT, and END" on page 132.

In a SELECT statement that retrieves values from a procedure, the column names must match the names and datatypes of the procedure's output parameters. In an EXECUTE PROCEDURE statement, the output parameters need not match the names of the procedure's output parameters, but the datatypes must match.

▶ *Using assignment statements*

A procedure can assign values to variables with the syntax:

```
variable = expression;
```

where *expression* is any valid combination of variables, operators, and expressions, and can include user-defined functions (UDFs) and generators.

A colon need not precede the variable name in an assignment statement. For example, the following statement assigns a value of zero to the local variable, *any_sales*:

```
any_sales = 0;
```

Variables should be assigned values of the datatype that they are declared to be. Numeric variables should be assigned numeric values, and character variables assigned character values. InterBase provides automatic type conversion. For example, a character variable can be assigned a numeric value, and the numeric value is automatically converted to a string. For more information on type conversion, see the *Programmer's Guide*.

▶ *Using* SELECT *statements*

In a stored procedure, use the SELECT statement with an INTO clause to retrieve a single row value from the database and assign it to a host variable. The SELECT statement must return at most one row from the database, like a standard singleton SELECT. The INTO clause is required and must be the last clause in the statement.

For example, the following statement is a standard singleton SELECT statement in an application:

```
EXEC SQL
    SELECT SUM(BUDGET), AVG(BUDGET)
        INTO :tot_budget, :avg_budget
```

```
      FROM DEPARTMENT
          WHERE HEAD_DEPT = :head_dept;
```

To use this SELECT statement in a procedure, move the INTO clause to the end as follows:

```
SELECT SUM(BUDGET), AVG(BUDGET)
    FROM DEPARTMENT
        WHERE HEAD_DEPT = :head_dept
        INTO :tot_budget, :avg_budget;
```

For a complete discussion of SELECT statement syntax, see the *Language Reference*.

▶ *Using* FOR SELECT … DO *statements*

To retrieve multiple rows in a procedure, use the FOR SELECT … DO statement. The syntax of FOR SELECT is:

```
FOR
    <select_expr>
DO
    <compound_statement>;
```

FOR SELECT differs from a standard SELECT as follows:

- It is a loop statement that retrieves the row specified in the *select_expr* and performs the statement or block following DO for each row retrieved.

- The INTO clause in the *select_expr* is required and must come last. This syntax allows FOR … SELECT to use the SQL UNION clause, if needed.

For example, the following statement from a procedure selects department numbers into the local variable, *rdno*, which is then used as an input parameter to the DEPT_BUDGET procedure:

```
FOR SELECT DEPT_NO
    FROM DEPARTMENT
        WHERE HEAD_DEPT = :dno
        INTO :rdno
DO
    BEGIN
        EXECUTE PROCEDURE DEPT_BUDGET :rdno RETURNS :sumb;
        tot = tot + sumb;
    END
… ;
```

▶ *Using* WHILE … DO *statements*

WHILE … DO is a looping statement that repeats a statement or block of statements as long as a condition is true. The condition is tested at the start of each loop. WHILE … DO uses the following syntax:

```
WHILE (<condition>) DO
    <compound_statement>
<compound_statement> =
    {<block> | statement;}
```

The *compound_statement* is executed as long as *condition* remains TRUE.
A *block* is one or more compound statements enclosed by BEGIN and END.

For example, the following procedure uses a WHILE … DO loop to compute the sum of all integers from one up to the input parameter, *i*:

```
SET TERM !!;
CREATE PROCEDURE SUM_INT (i INTEGER) RETURNS (s INTEGER)
AS
BEGIN
    s = 0;
    WHILE (i > 0) DO
    BEGIN
        s = s + i;
        i = i - 1;
    END
END!!
SET TERM ; !!
```

If this procedure is called from **isql** with the command:

```
EXECUTE PROCEDURE SUM_INT 4;
```

then the results will be:

```
S
==========
10
```

▶ *Using* IF … THEN … ELSE *statements*

The IF … THEN … ELSE statement selects alternative courses of action by testing a specified condition. The syntax of IF … THEN … ELSE is as follows:

```
IF (<condition>) THEN
    <compound_statement>
[ELSE
    <compound_statement>]
```

```
<compound_statement> =
    {<block> | statement;}
```

The *condition* clause is an expression that must evaluate to TRUE to execute the statement or block following THEN. The optional ELSE clause specifies an alternative statement or block to be executed if *condition* is FALSE.

The following lines of code illustrate the use of IF … THEN, assuming the variables *line2*, *first*, and *last* have been previously declared:

```
. . .
IF (first IS NOT NULL) THEN
    line2 = first || " " || last;
ELSE
    line2 = last;
. . .
```

▶ *Using event alerters*

To use an event alerter in a stored procedure, use the following syntax:

```
POST_EVENT <event_name>;
```

The parameter, *event_name*, can be either a quoted literal or string variable.

Note Variable names do not need to be—and must not be—preceded by a colon in stored procedures *except* in SELECT, INSERT, UPDATE, and DELETE clauses where they would be interpreted as column names without the colon.

When the procedure is executed, this statement notifies the event manager, which alerts applications waiting for the named event. For example, the following statement posts an event named "new_order":

```
POST_EVENT "new_order";
```

Alternatively, a variable can be used for the event name:

```
POST_EVENT event_name;
```

So, the statement can post different events, depending on the value of the string variable, *event_name*.

For more information on events and event alerters, see the *Programmer's Guide*.

▶ *Adding comments*

Stored procedure code should be commented to aid debugging and application development. Comments are especially important in stored procedures since they are global to the database and can be used by many different application developers.

Comments in stored procedure definitions are exactly like comments in standard C code, and use the following syntax:

```
/* comment_text */
```

comment_text can be any number of lines of text. A comment can appear on the same line as code. For example:

```
x = 42; /* Initialize value of x. */
```

▶ *Creating nested and recursive procedures*

A stored procedure can itself execute a stored procedure. Each time a stored procedure calls another procedure, the call is said to be *nested* because it occurs in the context of a previous and still active call to the first procedure. A stored procedure called by another stored procedure is known as a *nested procedure.*

If a procedure calls itself, it is *recursive.* Recursive procedures are useful for tasks that involve repetitive steps. Each invocation of a procedure is referred to as an *instance*, since each procedure call is a separate entity that performs as if called from an application, reserving memory and stack space as required to perform its tasks.

Note Stored procedures can be nested up to 1,000 levels deep. This limitation helps to prevent infinite loops that can occur when a recursive procedure provides no absolute terminating condition. Nested procedure calls can be restricted to fewer than 1,000 levels by memory and stack limitations of the server.

The following example illustrates a recursive procedure, FACTORIAL, which calculates factorials. The procedure calls itself recursively to calculate the factorial of *num*, the input parameter.

```
SET TERM !!;
CREATE PROCEDURE FACTORIAL (num INT)
    RETURNS (n_factorial DOUBLE PRECISION)
AS
DECLARE VARIABLE num_less_one INT;
BEGIN
    IF (num = 1) THEN
    BEGIN /**** Base case: 1 factorial is 1 ****/
        n_factorial = 1;
        SUSPEND;
    END
    ELSE
    BEGIN /**** Recursion: num factorial = num * (num-1) factorial ****/
        num_less_one = num - 1;
        EXECUTE PROCEDURE FACTORIAL num_less_one
            RETURNING_VALUES n_factorial;
```

```
        n_factorial = n_factorial * num;
        SUSPEND;
    END
END!!
SET TERM ;!!
```

The following C code demonstrates how a host-language program would call FACTORIAL:

```
. . .
printf("\nCalculate factorial for what value? ");
scanf("%d", &pnum);
EXEC SQL
    EXECUTE PROCEDURE FACTORIAL :pnum RETURNING_VALUES :pfact;
printf("%d factorial is %d.\n", pnum, pfact);
. . .
```

Recursion nesting restrictions would not allow this procedure to calculate factorials for numbers greater than 1,001. Arithmetic overflow, however, occurs for much smaller numbers.

▶ *Using* SUSPEND, EXIT, *and* END

The SUSPEND statement suspends execution of a select procedure, passes control back to the program, and resumes execution from the next statement when the next FETCH is executed. SUSPEND also returns values in the output parameters of a stored procedure.

SUSPEND should not be used in executable procedures, since the statements that follow it will never execute. Use EXIT instead to indicate to the reader explicitly that the statement terminates the procedure.

In a select procedure, the SUSPEND statement returns current values of output parameters to the calling program and continues execution. If an output parameter has not been assigned a value, its value is unpredictable, which can lead to errors. A procedure should ensure that all output parameters are assigned values before a SUSPEND.

In both select and executable procedures, EXIT jumps program control to the final END statement in the procedure.

What happens when a procedure reaches the final END statement depends on the type of procedure:

- In a select procedure, the final END statement returns control to the application and sets SQLCODE to 100, which indicates there are no more rows to retrieve.

- In an executable procedure, the final END statement returns control and values of output parameters, if any, to the calling application.

The behavior of these statements is summarized in the following table:

Procedure type	SUSPEND	EXIT	END
Select procedure	Suspends execution of procedure until next FETCH Returns values	Jumps to final END	Returns control to application Sets SQLCODE to 100
Executable procedure	Jumps to final END Not Recommended	Jumps to final END	Returns values Returns control to application

TABLE 9.3   SUSPEND, EXIT, and END

Consider the following procedure:

```
SET TERM !!;
CREATE PROCEDURE P RETURNS (r INTEGER)
AS
BEGIN
    r = 0;
    WHILE (r < 5) DO
    BEGIN
        r = r + 1;
        SUSPEND;
        IF (r = 3) THEN
            EXIT;
    END
END;
SET TERM ;!!
```

If this procedure is used as a select procedure, for example:

```
SELECT * FROM P;
```

then it returns values 1, 2, and 3 to the calling application, since the SUSPEND statement returns the current value of *r* to the calling application. The procedure terminates when it encounters EXIT.

If the procedure is used as an executable procedure, for example:

```
EXECUTE PROCEDURE P;
```

then it returns 1, since the SUSPEND statement terminates the procedure and returns the current value of *r* to the calling application. This is not recommended, but is included here for comparison.

Note If a select procedure has executable statements following the last SUSPEND in the procedure, all of those statements are executed, even though no more rows are returned to the calling program. The procedure terminates with the final END statement.

ERROR BEHAVIOR

When a procedure encounters an error—either an SQLCODE error, GDSCODE error, or user-defined exception—all statements since the last SUSPEND are undone.

Since select procedures can have multiple SUSPENDs, possibly inside a loop statement, only the actions since the last SUSPEND are undone. Since executable procedures should not use SUSPEND, when an error occurs the entire executable procedure is undone (if EXIT is used, as recommended).

Altering stored procedures

To change a stored procedure, use ALTER PROCEDURE. This statement changes the definition of an existing stored procedure without affecting its dependencies. If a procedure has dependencies which prevent you from dropping, changing, and recreating it, use ALTER PROCEDURE.

Changes made with ALTER PROCEDURE are automatically reflected in all applications that use the procedure without recompiling and relinking them. Only the creator of a procedure can alter it.

IMPORTANT Be careful about changing the type, number, and order of input and output parameters to a procedure, since existing code might assume that the procedure has its original format.

When a procedure is altered, the new procedure definition replaces the old one. Therefore, to alter a procedure, follow these steps:

1. Copy the original data definition file used to create the procedure. Alternatively, use **isql -extract** to extract a procedure from the database to a file.

2. Edit the file, changing CREATE to ALTER, and changing the procedure definition as desired. Retain whatever is still useful.

The syntax for ALTER PROCEDURE is similar to CREATE PROCEDURE as shown in the following syntax:

```
ALTER PROCEDURE name
    [(var datatype [, var datatype …])]
    [RETURNS (var datatype [, var datatype …])]
AS
    <procedure_body>;
```

The procedure *name* must be the name of an existing procedure. The arguments of the ALTER PROCEDURE statement are the same as CREATE PROCEDURE.

Dropping procedures

The DROP PROCEDURE statement deletes an existing stored procedure from the database. DROP PROCEDURE can be used interactively with **isql** or in a data definition file.

Note Procedures cannot be dropped with embedded SQL.

The following restrictions apply to dropping procedures:

- Only the creator of a procedure can drop it.

- Procedures used by other procedures, triggers, or views cannot be dropped.

- Procedures currently in use cannot be dropped.

If you attempt to drop a procedure and receive an error, make sure you have entered the procedure name correctly.

TIP   To see a list of database procedures and their dependencies, use the **isql** command, SHOW PROCEDURES.

The syntax for dropping a procedure is:

```
DROP PROCEDURE name;
```

The procedure *name* must be the name of an existing procedure. The following statement deletes the ACCOUNTS_BY_CLASS procedure:

```
DROP PROCEDURE ACCOUNTS_BY_CLASS;
```

Using stored procedures

Stored procedures can be used in applications in a variety of ways. Select procedures are used in place of a table or view in a SELECT statement. Executable procedures are used with an EXECUTE PROCEDURE statement.

Both kinds of procedures are defined with CREATE PROCEDURE and have the same syntax. The difference is in how the procedure is written and how it is intended to be used. Select procedures always return one or more rows, so that to the calling program they appear as a table or view. Executable procedures are simply routines invoked by the calling program and only optionally return values.

In fact, a single procedure can be used as a select procedure or an executable procedure, but this is not recommended. A procedure should be written specifically to be used in a SELECT statement (a select procedure) or to be used in an EXECUTE PROCEDURE statement (an executable procedure).

During application development, create and test stored procedures in **isql**. Once a stored procedure has been created, tested, and refined, it can be used in applications. For more information on using stored procedures in applications, see the *Programmer's Guide*.

Using executable procedures in isql

An executable procedure is invoked with EXECUTE PROCEDURE. It can return at most one row. To execute a stored procedure in **isql**, use the following syntax:

```
EXECUTE PROCEDURE name [(] [param [, param …]] [)];
```

The procedure *name* must be specified, and each *param* is an input parameter value (a constant). All input parameters required by the procedure must be supplied.

IMPORTANT   In **isql**, do not supply output parameters or use RETURNING_VALUES in the EXECUTE PROCEDURE statement, even if the procedure returns values. **isql** automatically displays output parameters.

To execute the procedure, DEPT_BUDGET, from **isql**, use:

```
EXECUTE PROCEDURE DEPT_BUDGET 110;
```

isql displays this output:

```
            TOT
====================
      1700000.00
```

Using select procedures in isql

A select procedure is used in place of a table or view in a SELECT statement and can return a single row or multiple rows.

The advantages of select procedures over tables or views are:

- They can take input parameters that can affect the output.
- They can contain logic not available in normal queries or views.
- They can return rows from multiple tables using UNION.

The syntax of SELECT from a procedure is:

```
SELECT <col_list> from name ([param [, param …]])
    WHERE <search_condition>
    ORDER BY <order_list>;
```

The procedure *name* must be specified, and in **isql** each *param* is a constant passed to the corresponding input parameter. All input parameters required by the procedure must be supplied. The *col_list* is a comma-delimited list of output parameters returned by the procedure, or * to select all rows.

The WHERE clause specifies a *search_condition* that selects a subset of rows to return. The ORDER BY clause specifies how to order the rows returned. For more information on SELECT, see the *Language Reference*.

The following code defines the procedure, GET_EMP_PROJ, which returns *emp_proj*, the project numbers assigned to an employee, when it is passed the employee number, *emp_no*, as the input parameter.

```
SET TERM !! ;
CREATE PROCEDURE GET_EMP_PROJ (emp_no SMALLINT)
RETURNS (emp_proj SMALLINT) AS
BEGIN
    FOR SELECT PROJ_ID
        FROM EMPLOYEE_PROJECT
        WHERE EMP_NO = :emp_no
        INTO :emp_proj
    DO
        SUSPEND;
END !!
```

The following statement selects from GET_EMP_PROJ in **isql**:

```
SELECT * FROM GET_EMP_PROJ(24);
```

The output is:

```
PROJ_ID
=======
DGPII
GUIDE
```

The following select procedure, ORG_CHART, displays an organizational chart:

```
CREATE PROCEDURE ORG_CHART
RETURNS (HEAD_DEPT CHAR(25), DEPARTMENT CHAR(25),
        MNGR_NAME CHAR(20), TITLE CHAR(5), EMP_CNT INTEGER)
AS
    DECLARE VARIABLE mngr_no INTEGER;
```

```
        DECLARE VARIABLE dno CHAR(3);
    BEGIN
        FOR SELECT H.DEPARTMENT, D.DEPARTMENT, D.MNGR_NO, D.DEPT_NO
            FROM DEPARTMENT D
            LEFT OUTER JOIN DEPARTMENT H ON D.HEAD_DEPT = H.DEPT_NO
            ORDER BY D.DEPT_NO
            INTO :head_dept, :department, :mngr_no, :dno
        DO
        BEGIN
            IF (:mngr_no IS NULL) THEN
            BEGIN
                mngr_name = "--TBH--";
                title = "";
            END
            ELSE
                SELECT FULL_NAME, JOB_CODE
                FROM EMPLOYEE
                WHERE EMP_NO = :mngr_no
                INTO :mngr_name, :title;
            SELECT COUNT(EMP_NO)
            FROM EMPLOYEE
            WHERE DEPT_NO = :dno
            INTO :emp_cnt;
            SUSPEND;
        END
    END !!
```

ORG_CHART is invoked from **isql** as follows:

```
SELECT * FROM ORG_CHART;
```

For each department, the procedure displays the department name, the department's "head department" (managing department), the department manager's name and title, and the number of employees in the department.

HEAD_DEPT	DEPARTMENT	MNGR_NAME	TITLE	EMP_CNT
================	===============	=============	=====	======
	Corporate Headquarters	Bender, Oliver H.	CEO	2
Corporate Headquarters	Sales and Marketing	MacDonald, Mary S.	VP	2

HEAD_DEPT	DEPARTMENT	MNGR_NAME	TITLE	EMP_CNT
Sales and Marketing	Pacific Rim Headquarters	Baldwin, Janet	Sales	2
Pacific Rim Headquarters	Field Office: Japan	Yamamoto, Takashi	SRep	2
Pacific Rim Headquarters	Field Office: Singapore	—TBH—		0

ORG_CHART must be used as a select procedure to display the full organization. If called with EXECUTE PROCEDURE, then the first time it encounters the SUSPEND statement, the procedure terminates, returning the information for Corporate Headquarters only.

SELECT can specify columns to retrieve from a procedure. For example, if ORG_CHART is invoked as follows:

```
SELECT DEPARTMENT FROM ORG_CHART;
```

then only the second column, DEPARTMENT, is displayed.

▶ *Using* WHERE *and* ORDER BY *clauses*

A SELECT from a stored procedure can contain WHERE and ORDER BY clauses, just as in a SELECT from a table or view.

The WHERE clause limits the results returned by the procedure to rows matching the search condition. For example, the following statement returns only those rows where the HEAD_DEPT is Sales and Marketing:

```
 SELECT * FROM ORG_CHART WHERE HEAD_DEPT = "Sales and Marketing";
```

The stored procedure then returns only the matching rows, for example:

HEAD_DEPT	DEPARTMENT	MNGR_NAME	TITLE	EMP_CNT
Sales and Marketing	Pacific Rim Headquarters	Baldwin, Janet	Sales	2
Sales and Marketing	European Headquarters	Reeves, Roger	Sales	3
Sales and Marketing	Field Office: East Cost	Weston, K. J.	SRep	2

The ORDER BY clause can be used to order the results returned by the procedure. For example, the following statement orders the results by EMP_CNT, the number of employees in each department, in ascending order (the default):

```
SELECT * FROM ORG_CHART ORDER BY EMP_CNT;
```

▶ *Selecting aggregates from procedures*

In addition to selecting values from a procedure, you can use aggregate functions. For example, to use ORG_CHART to display a count of the number of departments, use the following statement:

```
SELECT COUNT(DEPARTMENT) FROM ORG_CHART;
```

The results are:

```
      COUNT
============
         24
```

Similarly, to use ORG_CHART to display the maximum and average number of employees in each department, use the following statement:

```
SELECT MAX(EMP_CNT), AVG(EMP_CNT) FROM ORG_CHART;
```

The results are:

```
     MAX       AVG
========= =========
       5         2
```

If a procedure encounters an error or exception, the aggregate functions do not return the correct values, since the procedure terminates before all rows are processed.

Viewing arrays with stored procedures

If a table contains columns defined as arrays, you cannot view the data in the column with a simple SELECT statement, since only the array ID is stored in the table. Arrays can be used to display array values, as long as the dimensions and datatype of the array column are known in advance.

For example, in the *employee* database, the JOB table has a column named LANGUAGE_REQ containing the languages required for the position. The column is defined as an array of five VARCHAR(15).

In **isql**, if you perform a simple SELECT statement, such as:

```
SELECT JOB_CODE, JOB_GRADE, JOB_COUNTRY, LANGUAGE_REQ FROM JOB;
```

part of the results look like this:

```
JOB_CODE        JOB_GRADE       JOB_COUNTRY        LANGUAGE_REQ
==========      ===========     ==============     ==============
. . .
Sales           3               USA                <null>
```

Sales	3	England	20:af
SRep	4	USA	20:b0
SRep	4	England	20:b2
SRep	4	Canada	20:b4

To view the contents of the LANGUAGE_REQ column, use a stored procedure, such as the following:

```
SET TERM !! ;
CREATE PROCEDURE VIEW_LANGS
    RETURNS (code VARCHAR(5), grade SMALLINT, cty VARCHAR(15),
            lang VARCHAR(15))
AS
    DECLARE VARIABLE i INTEGER;
BEGIN
    FOR SELECT JOB_CODE, JOB_GRADE, JOB_COUNTRY
        FROM JOB
        WHERE LANGUAGE_REQ IS NOT NULL
        INTO :code, :grade, :cty
    DO
    BEGIN
        i = 1;
        WHILE (i <= 5) DO
        BEGIN
            SELECT LANGUAGE_REQ[:i] FROM JOB
            WHERE ((JOB_CODE = :code) AND (JOB_GRADE = :grade)
                AND (JOB_COUNTRY = :cty)) INTO :lang;
            i = i + 1;
            SUSPEND;
        END
    END
END!!
SET TERM ; !!
```

This procedure, VIEW_LANGS, uses a FOR … SELECT loop to retrieve each row from JOB for which LANGUAGE_REQ is not NULL. Then a WHILE loop retrieves each element of the LANGUAGE_REQ array and returns the value to the calling application (in this case, **isql**).

For example, if this procedure is invoked with:

```
SELECT * FROM VIEW_LANGS;
```

the output is:

```
CODE   GRADE CTY   LANG
===== ===== ============ =========
Eng  3   Japan    Japanese
Eng  3   Japan    Mandarin
Eng  3   Japan    English
Eng  3   Japan
Eng  3   Japan
Eng  4   England  English
Eng  4   England  German
Eng  4   England  French
. . .
```

This procedure can easily be modified to return only the language requirements for a particular job, when passed JOB_CODE, JOB_GRADE, and JOB_COUNTRY as input parameters.

Exceptions

An *exception* is a named error message that can be raised from a stored procedure. Exceptions are created with CREATE EXCEPTION, modified with ALTER EXCEPTION, and dropped with DROP EXCEPTION. A stored procedure *raises* an exception with EXCEPTION *name*.

When raised, an exception returns an error message to the calling program and terminates execution of the procedure that raised it, unless the exception is handled by a WHEN statement.

IMPORTANT Like procedures, exceptions are created and stored in a database, where they can be used by any procedure that needs them. Exceptions must be created and committed before they can be raised.

For more information on raising and handling exceptions, see "Raising an exception in a stored procedure" on page 144.

Creating exceptions

To create an exception, use the following CREATE EXCEPTION syntax:

```
CREATE EXCEPTION name "<message>";
```

For example, the following statement creates an exception named REASSIGN_SALES:

```
CREATE EXCEPTION REASSIGN_SALES "Reassign the sales records before
deleting this employee.";
```

Altering exceptions

To change the message returned by an exception, use the following syntax:

```
ALTER EXCEPTION name "<message>";
```

Only the creator of an exception can alter it. For example, the following statement changes the text of the exception created in the previous section:

```
ALTER EXCEPTION REASSIGN_SALES "Can't delete employee--Reassign Sales";
```

You can alter an exception even though a database object depends on it. If the exception is raised by a trigger, you cannot drop the exception unless you first drop the trigger or stored procedure. Use ALTER EXCEPTION instead.

Dropping exceptions

To delete an exception, use the following syntax:

```
DROP EXCEPTION name;
```

For example, the following statement drops the exception, REASSIGN_SALES:

```
DROP EXCEPTION REASSIGN_SALES;
```

The following restrictions apply to dropping exceptions:

- Only the creator of an exception can drop it.
- Exceptions used in existing procedures and triggers cannot be dropped.
- Exceptions currently in use cannot be dropped.

TIP In **isql**, SHOW PROCEDURES displays a list of *dependencies*, the procedures, exceptions, and tables which the stored procedure uses. SHOW PROCEDURE *name* displays the body and header information for the named procedure. SHOW TRIGGERS *table* displays the triggers defined for *table*. SHOW TRIGGER *name* displays the body and header information for the named trigger.

Raising an exception in a stored procedure

To raise an exception in a stored procedure, use the following syntax:

```
EXCEPTION name;
```

where *name* is the name of an exception that already exists in the database.

When an exception is raised, it does the following:

- Terminates the procedure in which it was raised and undoes any actions performed (directly or indirectly) by the procedure.

- Returns an error message to the calling application. In **isql**, the error message is displayed on the screen.

  Note If an exception is handled with a WHEN statement, it behaves differently. For more information on exception handling, see "Handling exceptions" on page 145.

  The following statements raise the exception, REASSIGN_SALES:

```
IF (any_sales > 0) THEN
    EXCEPTION REASSIGN_SALES;
```

Handling errors

Procedures can handle three kinds of errors with a WHEN … DO statement:

- Exceptions raised by EXCEPTION statements in the current procedure, in a nested procedure, or in a trigger fired as a result of actions by such a procedure.

- SQL errors reported in SQLCODE.

- InterBase errors reported in GDSCODE.

The WHEN ANY statement handles any of the three types of errors.

For more information about InterBase error codes and SQLCODE values, see the *Language Reference*.

The syntax of the WHEN … DO statement is:

```
WHEN {<error> [, <error> …] | ANY}
    DO <compound_statement>
<error> =
    {EXCEPTION exception_name | SQLCODE number | GDSCODE errcode}
```

IMPORTANT   If used, WHEN must be the last statement in a BEGIN … END block. It should come after SUSPEND, if present.

Handling exceptions

Instead of terminating when an exception occurs, a procedure can respond to and perhaps correct the error condition by handling the exception. When an exception is raised, it does the following:

- Seeks a WHEN statement that handles the exception. If one is not found, it terminates execution of the BEGIN … END block containing the exception and undoes any actions performed in the block.

- Backs out one level to the surrounding BEGIN … END block and seeks a WHEN statement that handles the exception, and continues backing out levels until one is found. If no WHEN statement is found, the procedure is terminated and all its actions are undone.

- Performs the ensuing statement or block of statements specified by the WHEN statement that handles the exception.

- Returns program control to the block in the procedure following the WHEN statement.

    Note An exception that is handled does *not* return an error message.

Handling SQL errors

Procedures can also handle error numbers returned in SQLCODE. After each SQL statement executes, SQLCODE contains a status code indicating the success or failure of the statement. SQLCODE can also contain a warning status, such as when there are no more rows to retrieve in a FOR SELECT loop.

For example, if a procedure attempts to insert a duplicate value into a column defined as a PRIMARY KEY, InterBase returns SQLCODE -803. This error can be handled in a procedure with the following statement:

```
WHEN SQLCODE -803
DO
    BEGIN
    . . .
```

The following procedure includes a WHEN statement to handle SQLCODE -803 (attempt to insert a duplicate value in a UNIQUE key column). If the first column in TABLE1 is a UNIQUE key, and the value of parameter *a* is the same as one already in the table, then SQLCODE -803 is generated, and the WHEN statement sets an error message returned by the procedure.

```
SET TERM !!;
CREATE PROCEDURE NUMBERPROC (a INTEGER, b INTEGER)
    RETURNS (e CHAR(60)) AS
BEGIN
    BEGIN
        INSERT INTO TABLE1 VALUES (:a, :b);
        WHEN SQLCODE -803 DO
            e = "Error Attempting to Insert in TABLE1 - Duplicate Value.";
    END;
END!!
SET TERM; !!
```

For more information about SQLCODE, see the *Language Reference*.

Handling InterBase errors

Procedures can also handle InterBase errors. For example, suppose a statement in a procedure attempts to update a row already updated by another transaction, but not yet committed. In this case, the procedure might receive an InterBase error **lock_conflict**. If the procedure retries its update, the other transaction might have rolled back its changes and released its locks. By using a WHEN GDSCODE statement, the procedure can handle lock conflict errors and retry its operation.

To handle InterBase error codes, use the following syntax:

```
WHEN GDSCODE errcode DO <compound_statement>;
```

For more information about InterBase error codes, see the *Language Reference*.

Examples of error behavior and handling

When a procedure encounters an error—either an SQLCODE error, GDSCODE error, or user-defined exception—the statements since the last SUSPEND are undone.

SUSPEND should not be used in executable procedures. EXIT should be used to terminate the procedure. If this recommendation is followed, then when an executable procedure encounters an error, the entire procedure is undone. Since select procedures can have multiple SUSPENDs, possibly inside a loop statement, only the actions since the last SUSPEND are undone.

For example, here is a simple executable procedure that attempts to insert the same values twice into the PROJECT table.

```
SET TERM !! ;
CREATE PROCEDURE NEW_PROJECT
    (id CHAR(5), name VARCHAR(20), product VARCHAR(12))
    RETURNS (result VARCHAR(80))
AS
BEGIN
INSERT INTO PROJECT (PROJ_ID, PROJ_NAME, PRODUCT)
    VALUES (:id, :name, :product);
result = "Values inserted OK.";
INSERT INTO PROJECT (PROJ_ID, PROJ_NAME, PRODUCT)
    VALUES (:id, :name, :product);
result = "Values Inserted Again.";
EXIT;
WHEN SQLCODE -803 DO
BEGIN
    result = "Could Not Insert Into Table - Duplicate Value";
EXIT;
END
END!!
SET TERM ; !!
```

This procedure can be invoked with a statement such as:

```
EXECUTE PROCEDURE NEW_PROJECT "XXX", "Project X", "N/A";
```

The second INSERT generates an error (SQLCODE -803, "invalid insert—no two rows can have duplicate values."). The procedure returns the string, "Could Not Insert Into Table - Duplicate Value," as specified in the WHEN clause, and the entire procedure is undone.

The next example is written as a select procedure, and invoked with the SELECT statement that follows it:

```
. . .
INSERT INTO PROJECT (PROJ_ID, PROJ_NAME, PRODUCT)
    VALUES (:id, :name, :product);
result = "Values inserted OK.";
SUSPEND;
INSERT INTO PROJECT (PROJ_ID, PROJ_NAME, PRODUCT)
```

```
    VALUES (:id, :name, :product);
result = "Values Inserted Again.";
SUSPEND;
WHEN SQLCODE -803 DO
BEGIN
    result = "Could Not Insert Into Table - Duplicate Value";
EXIT;
END
END!!
SET TERM ; !!
SELECT * FROM SIMPLE("XXX", "Project X", "N/A");
```

The first INSERT is performed, and SUSPEND returns the result string, "Values Inserted OK." The second INSERT generates the error because there have been no statements performed since the last SUSPEND, and no statements are undone. The WHEN statement returns the string, "Could Not Insert Into Table - Duplicate Value", in addition to the previous result string.

The select procedure successfully performs the insert, while the executable procedure does not.

The next example is a more complex stored procedure that demonstrates SQLCODE error handling and exception handling. It is based on the previous example of a select procedure, and does the following:

- Accepts a project ID, name, and product type, and ensures that the ID is in all capitals, and the product type is acceptable.

- Inserts the new project data into the PROJECT table, and returns a string confirming the operation, or an error message saying the project is a duplicate.

- Uses a FOR … SELECT loop with a correlated subquery to get the first three employees not assigned to any project and assign them to the new project using the ADD_EMP_PROJ procedure.

- If the CEO's employee number is selected, raises the exception, CEO, which is handled with a WHEN statement that assigns the CEO's administrative assistant (employee number 28) instead to the new project.

Note that the exception, CEO, is handled within the FOR … SELECT loop, so that only the block containing the exception is undone, and the loop and procedure continue after the exception is raised.

```
CREATE EXCEPTION CEO "Can't Assign CEO to Project.";
SET TERM !! ;
CREATE PROCEDURE NEW_PROJECT
    (id CHAR(5), name VARCHAR(20), product VARCHAR(12))
    RETURNS (result VARCHAR(30), num smallint)
```

```
    AS
        DECLARE VARIABLE emp_wo_proj smallint;
        DECLARE VARIABLE i smallint;
    BEGIN
        id = UPPER(id); /* Project id must be in uppercase. */
        INSERT INTO PROJECT (PROJ_ID, PROJ_NAME, PRODUCT)
            VALUES (:id, :name, :product);
        result = "New Project Inserted OK.";
        SUSPEND;
/* Add Employees to the new project */
        i = 0;
        result = "Project Got Employee Number:";
        FOR SELECT EMP_NO FROM EMPLOYEE
            WHERE EMP_NO NOT IN (SELECT EMP_NO FROM EMPLOYEE_PROJECT)
            INTO :emp_wo_proj
        DO
        BEGIN
            IF (i < 3) THEN
            BEGIN
                IF (emp_wo_proj = 5) THEN
                    EXCEPTION CEO;
                EXECUTE PROCEDURE ADD_EMP_PROJ :emp_wo_proj, :id;
                num = emp_wo_proj;
                SUSPEND;
            END
            ELSE
                EXIT;
            i = i + 1;
            WHEN EXCEPTION CEO DO
            BEGIN
                EXECUTE PROCEDURE ADD_EMP_PROJ 28, :id;
                num = 28;
                SUSPEND;
            END
        END
/* Error Handling */
    WHEN SQLCODE -625 DO
    BEGIN
        IF ((:product <> "software") OR (:product <> "hardware") OR
            (:product <> "other") OR (:product <> "N/A")) THEN
            result = "Enter product: software, hardware, other, or N/A";
    END
    WHEN SQLCODE -803 DO
        result = "Could not insert into table - Duplicate Value";
```

```
END!!
SET TERM ; !!
```

This procedure can be called with a statement such as:

```
SELECT * FROM NEW_PROJECT("XYZ", "Alpha project", "software");
```

With results (in **isql**) such as:

```
RESULT                     NUM
==========================  ======
New Project Inserted OK. <null>
Project Got Employee Number: 28
Project Got Employee Number: 29
Project Got Employee Number: 36
```

10

Creating Triggers

A *trigger* is a self-contained routine associated with a table or view that automatically performs an action when a row in the table or view is inserted, updated, or deleted.

A trigger is never called directly. Instead, when an application or user attempts to INSERT, UPDATE, or DELETE a row in a table, any triggers associated with that table and operation are automatically executed, or *fired*.

Triggers can make use of exceptions, named messages called for error handling. When an exception is raised by a trigger, it returns an error message, terminates the trigger, and undoes any changes made by the trigger, unless the exception is handled with a WHEN statement in the trigger.

The advantages of using triggers are:

- Automatic enforcement of data restrictions, to make sure users enter only valid values into columns.

- Reduced application maintenance, since changes to a trigger are automatically reflected in all applications that use the associated table without the need to recompile and relink.

- Automatic logging of changes to tables. An application can keep a running log of changes with a trigger that fires whenever a table is modified.

- Automatic notification of changes to the database with event alerters in triggers.

Working with triggers

With **isql**, you can create, alter, and drop triggers and exceptions. Each of these operations is explained in this chapter. There are two ways to create, alter, and drop triggers with **isql**:

- Interactively

- With an input file containing data definition statements

It is preferable to use data definition files, because it is easier to modify these files and provide a record of the changes made to the database. For simple changes to existing triggers or exceptions, the interactive interface can be convenient.

Using a data definition file

To create or alter a trigger through a data definition file, follow these steps:

1. Use a text editor to write the data definition file.

2. Save the file.

3. Process the file with **isql**. Use the command:

   ```
   isql -input filename database_name
   ```

   where *filename* is the name of the data definition file and *database_name* is the name of the database used. Alternatively, from within **isql**, you can interactively process the file using the command:

   ```
   SQL> input filename;
   ```

Note If you do not specify the database on the command line or interactively, the data definition file must include a statement to create or open a database.

The data definition file may include:

- Statements to create, alter, or drop triggers. The file can also include statements to create, alter, or drop procedures and exceptions. Exceptions must be created and committed before they can be referenced in procedures and triggers.

- Any other **isql** statements.

Creating triggers

A trigger is defined with the CREATE TRIGGER statement, which is composed of a *header* and a *body*. The trigger header contains:

- A trigger name, unique within the database.

- A table name, identifying the table with which to associate the trigger.

- Statements that determine when the trigger fires.

  The trigger 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. These statements are performed when the trigger fires. A block can itself include other blocks, so that there may be many levels of nesting.

IMPORTANT   Because each statement in the trigger body must be terminated by a semicolon, you must define a different symbol to terminate the trigger body itself. In **isql**, include a SET TERM statement before CREATE TRIGGER to specify a terminator other than a semicolon. After the body of the trigger, include another SET TERM to change the terminator back to a semicolon.

CREATE TRIGGER syntax

The syntax of CREATE TRIGGER is:

```
CREATE TRIGGER name FOR {table | view}
[ACTIVE | INACTIVE]
{BEFORE | AFTER} {DELETE | INSERT | UPDATE}
[POSITION number]
AS <trigger_body>

<trigger_body> =
    [<variable_declaration_list>]
    <block>

<variable_declaration_list> =
    DECLARE VARIABLE variable datatype;
    [DECLARE VARIABLE variable datatype; …]

<block> =
BEGIN
    <compound_statement>
    [<compound_statement> …]
END
```

`<compound_statement> = {<block> | statement;}`

Argument	Description
name	Name of the trigger. The name must be unique in the database.
table	Name of the table or view that causes the trigger to fire when the specified operation occurs on the table or view.
ACTIVE|INACTIVE	Optional. Specifies trigger action at transaction end:
	ACTIVE: (Default). Trigger takes effect.
	INACTIVE: Trigger does not take effect.
BEFORE|AFTER	Required. Specifies whether the trigger fires:
	BEFORE: Before associated operation.
	AFTER: After associated operation.
	Associated operations are DELETE, INSERT, or UPDATE.
DELETE|INSERT | UPDATE	Specifies the table operation that causes the trigger to fire.
POSITION *number*	Specifies firing order for triggers before the same action or after the same action. *number* must be an integer between 0 and 32,767, inclusive. Lower-number triggers fire first. Default: 0 = first trigger to fire.
	Triggers for a table need not be consecutive. Triggers on the same action with the same position number will fire in alphabetic order by name.
DECLARE VARIABLE *var* *<datatype>*	Declares local variables used only in the trigger. Each declaration must be preceded by DECLARE VARIABLE and followed by a semicolon (;).
	var: Local variable name, unique in the trigger.
	<datatype>: The datatype of the local variable.
statement	Any single statement in InterBase procedure and trigger language. Each statement except BEGIN and END must be followed by a semicolon (;).
terminator	Terminator defined by the SET TERM statement which signifies the end of the trigger body. Used in **isql** only.

TABLE 10.1   Arguments of the CREATE TRIGGER statement

InterBase procedure and trigger language

InterBase procedure and trigger language is a complete programming language for stored procedures and triggers. It includes:

- SQL data manipulation statements: INSERT, UPDATE, DELETE, and singleton SELECT.
- SQL operators and expressions, including UDFs linked with the database server and generators.
- Powerful extensions to SQL, including assignment statements, control-flow statements, context variables, event-posting statements, exceptions, and error-handling statements.

Although stored procedures and triggers are used in entirely different ways and for different purposes, they both use procedure and trigger language. Both triggers and stored procedures may use any statements in procedure and trigger language, with some exceptions:

- Context variables are unique to triggers.
- Input and output parameters, and the SUSPEND and EXIT statements which return values are unique to stored procedures.

The following table summarizes the language extensions for triggers:

Statement	Description
BEGIN … END	Defines a block of statements that executes as one. The BEGIN keyword starts the block; the END keyword terminates it. Neither should be followed by a semicolon.
variable = expression	Assignment statement which assigns the value of *expression* to local variable, *variable*.
/* *comment_text* */	Programmer's comment, where *comment_text* can be any number of lines of text.
EXCEPTION *exception_name*	Raises the named exception. An exception is a user-defined error, which returns an error message to the calling application unless handled by a WHEN statement.
EXECUTE PROCEDURE *proc_name* [*var* [, *var* …]] [RETURNING_VALUES *var* [, *var* …]]	Executes stored procedure, *proc_name*, with the listed input arguments, returning values in the listed output arguments. Input and output arguments must be local variables.

TABLE 10.2   Procedure and trigger language extensions

Statement	Description
FOR *<select_statement>* DO *<compound_statement>*	Repeats the statement or block following DO for every qualifying row retrieved by *<select_statement>*.
	<select_statement>: a normal SELECT statement, except the INTO clause is required and must come last.
<compound_statement>	Either a single statement in procedure and trigger language or a block of statements bracketed by BEGIN and END.
IF (*<condition>*) THEN *<compound_statement>* [ELSE *<compound_statement>*]	Tests *<condition>*, and if it is TRUE, performs the statement or block following THEN, otherwise performs the statement or block following ELSE, if present.
	<condition>: a Boolean expression (TRUE, FALSE, or UNKNOWN), generally two expressions as operands of a comparison operator.
NEW.*column*	New context variable that indicates a new column value in an INSERT or UPDATE operation.
OLD.*column*	Old context variable that indicates a column value before an UPDATE or DELETE operation.
POST_EVENT *event_name*	Posts the event, *event_name*.
WHILE (*<condition>*) DO *<compound_statement>*	While *<condition>* is TRUE, keep performing *<compound_statement>*. First *<condition>* is tested, and if it is TRUE, then *<compound_statement>* is performed. This sequence is repeated until *<condition>* is no longer TRUE.
WHEN {*<error>* [, *<error>* …]|ANY} DO *<compound_statement>*	Error-handling statement. When one of the specified errors occurs, performs *<compound_statement>*. WHEN statements, if present, must come at the end of a block, just before END.
	<error>: EXCEPTION *exception_name,* SQLCODE *errcode* or GDSCODE *number*.
	ANY: handles any errors.

TABLE 10.2   Procedure and trigger language extensions (*continued*)

▸ *Using* SET TERM *in isql*

Because each statement in a trigger body must be terminated by a semicolon, you must define a different symbol to terminate the trigger body itself. In **isql,** include a SET TERM statement before CREATE TRIGGER to specify a terminator other than a semicolon. After the body of the trigger, include another SET TERM to change the terminator back to a semicolon.

The following example illustrates the use of SET TERM for a trigger. The terminator is temporarily set to a double exclamation point (!!).

```
SET TERM !! ;
CREATE TRIGGER SIMPLE FOR EMPLOYEE
AFTER UPDATE AS
BEGIN
…
END !!
SET TERM ; !!
```

There must be a space after SET TERM. Each SET TERM is itself terminated with the current terminator.

▸ *Syntax errors in triggers*

InterBase may generate errors during parsing if there is incorrect syntax in the CREATE TRIGGER statement. Error messages look similar to this:

```
Statement failed, SQLCODE = -104
Dynamic SQL Error
-SQL error code = -104
-Token unknown - line 4, char 9
-tmp
```

The line numbers are counted from the beginning of the CREATE TRIGGER statement, not from the beginning of the data definition file. Characters are counted from the left, and the unknown token indicated will either be the source of the error or immediately to the right of the source of the error. When in doubt, examine the entire line to determine the source of the syntax error.

The trigger header

Everything before the AS clause in the CREATE TRIGGER statement forms the trigger header. The header must specify the name of the trigger and the name of the associated table or view. The table or view must exist before it can be referenced in CREATE TRIGGER.

The trigger name must be unique among triggers in the database. Using the name of an existing trigger or a system-supplied constraint name results in an error.

The remaining clauses in the trigger header determine when and how the trigger fires:

- The *trigger status*, ACTIVE or INACTIVE, determines whether a trigger is activated when the specified operation occurs. ACTIVE is the default, meaning the trigger fires when the operation occurs. Setting status to INACTIVE with ALTER TRIGGER is useful when developing and testing applications and triggers.

- The *trigger time indicator*, BEFORE or AFTER, determines when the trigger fires relative to the specified operation. BEFORE specifies that trigger actions are performed before the operation. AFTER specifies that trigger actions are performed after the operation.

- The *trigger statement indicator* specifies the SQL operation that causes the trigger to fire: INSERT, UPDATE, or DELETE. Exactly one indicator must be specified. To use the same trigger for more than one operation, duplicate the trigger with another name and specify a different operation.

- The optional *sequence indicator*, POSITION *number*, specifies the order in which the trigger fires in relation to other triggers on the same table and event. *number* can be any integer between zero and 32,767. The default is zero. Lower-numbered triggers fire first. Multiple triggers can have the same position number; they will fire in random order.

The following example demonstrates how the POSITION clause determines trigger firing order. Here are four headers of triggers for the ACCOUNTS table:

```
CREATE TRIGGER A FOR ACCOUNTS BEFORE UPDATE POSITION 5 AS …
CREATE TRIGGER B FOR ACCOUNTS BEFORE UPDATE POSITION 0 AS …
CREATE TRIGGER C FOR ACCOUNTS AFTER UPDATE POSITION 5 AS …
CREATE TRIGGER D FOR ACCOUNTS AFTER UPDATE POSITION 3 AS …
```

When this update takes place:

```
UPDATE ACCOUNTS SET C = "canceled" WHERE C2 = 5;
```

The following sequence of events happens: trigger B fires, A fires, the update occurs, trigger D fires, then C fires.

The trigger body

Everything following the AS keyword in the CREATE TRIGGER statement forms the procedure body. The body consists of an optional list of local variable declarations followed by a block of statements.

A block is composed of statements in the InterBase procedure and trigger language, bracketed by BEGIN and END. A block can itself include other blocks, so that there may be many levels of nesting.

InterBase procedure and trigger language includes all standard InterBase SQL statements except data definition and transaction statements, plus statements unique to procedure and trigger language.

Statements unique to InterBase procedure and trigger language include:

- Assignment statements, to set values of local variables.

- Control-flow statements, such as IF … THEN, WHILE … DO, and FOR SELECT … DO, to perform conditional or looping tasks.

- EXECUTE PROCEDURE statements to invoke stored procedures.

- Exception statements, to return error messages, and WHEN statements, to handle specific error conditions.

- NEW and OLD context variables, to temporarily hold previous (old) column values and to insert or update (new) values.

- Generators, to generate unique numeric values for use in expressions. Generators can be used in procedures and applications as well as triggers, but they are particularly useful in triggers for inserting unique column values.

Note All of these statements (except context variables) can be used in both triggers and stored procedures. For a full description of these statements, see Chapter 9, "Working with Stored Procedures."

▶ NEW *and* OLD *context variables*

Triggers can use two context variables, OLD, and NEW. The OLD context variable refers to the current or previous values in a row being updated or deleted. OLD is not used for inserts. NEW refers to a new set of INSERT or UPDATE values for a row. NEW is not used for deletes. Context variables are often used to compare the values of a column before and after it is modified.

The syntax for context variables is as follows:

```
NEW.column
OLD.column
```

where *column* is any column in the affected row. Context variables can be used anywhere a regular variable can be used.

New values for a row can only be altered *before* actions. A trigger that fires after INSERT and tries to assign a value to NEW.*column* will have no effect. The actual column values are not altered until after the action, so triggers that reference values from their target tables will not see a newly inserted or updated value unless they fire after UPDATE or INSERT.

For example, the following trigger fires after the EMPLOYEE table is updated, and compares an employee's old and new salary. If there is a change in salary, the trigger inserts an entry in the SALARY_HISTORY table.

```
SET TERM !! ;
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 !!
SET TERM ; !!
```
Note Context variables are never preceded by a colon, even in SQL statements.

▶ *Using generators*

A *generator* is a database object which is automatically incremented each time the special function, GEN_ID(), is called. GEN_ID() can be used in a statement anywhere that a variable can be used. Generators are typically used to ensure that a number inserted into a column is unique, or in sequential order. Generators can be used in procedures and applications as well as triggers, but they are particularly useful in triggers for inserting unique column values.

A generator is created with the CREATE GENERATOR statement and initialized with the SET GENERATOR statement. If not initialized, a generator starts with a value of one. For more information about creating and initializing a generator, see the *Language Reference*.

A generator must be created with CREATE GENERATOR before it can be called by GEN_ID(). The syntax for using GEN_ID() in a statement is:

```
GEN_ID(genname, step)
```

genname must be the name of an existing generator, and *step* is the amount by which the current value of the generator is incremented. *step* may be an integer or an expression that evaluates to an integer.

The following trigger uses GEN_ID() to increment a new customer number before values are inserted into the CUSTOMER table:

```
SET TERM !! ;
CREATE TRIGGER SET_CUST_NO FOR CUSTOMER
BEFORE INSERT AS
BEGIN
    NEW.cust_no = GEN_ID(CUST_NO_GEN, 1);
END !!
SET TERM ; !!
```

Note This trigger must be defined to fire before the insert, since it assigns values to NEW.*cust_no.*

Altering triggers

To update a trigger definition, use ALTER TRIGGER. A trigger can be altered only by its creator.

ALTER TRIGGER can change:

- Only trigger header information, including the trigger activation status, when it performs its actions, the event that fires the trigger, and the order in which the trigger fires compared to other triggers.

- Only trigger body information, the trigger statements that follow the AS clause.

- Both trigger header and trigger body information. In this case, the new trigger definition replaces the old trigger definition.

To alter a trigger defined automatically by a CHECK constraint on a table, use ALTER TABLE to change the table definition. For more information on the ALTER TABLE statement, see Chapter 6, "Working with Tables."

The ALTER TRIGGER syntax is as follows:

```
ALTER TRIGGER name
    [ACTIVE | INACTIVE]
    [{BEFORE | AFTER} {DELETE | INSERT | UPDATE}]
    [POSITION number]
    AS <trigger_body>;
```

The syntax of ALTER TRIGGER is the same as CREATE TRIGGER, except:

- The CREATE keyword is replaced by ALTER.

- FOR *table* is omitted. ALTER TRIGGER cannot be used to change the table with which the trigger is associated.

- The statement need only include parameters that are to be altered in the existing trigger, with certain exceptions listed in the following sections.

Altering a trigger header

When used to change only a trigger header, ALTER TRIGGER requires at least one altered setting after the trigger name. Any setting omitted from ALTER TRIGGER remains unchanged.

The following statement makes the trigger, SAVE_SALARY_CHANGE, inactive:

```
ALTER TRIGGER SAVE_SALARY_CHANGE INACTIVE;
```

If the time indicator (BEFORE or AFTER) is altered, then the operation (UPDATE, INSERT, or DELETE) must also be specified. For example, the following statement reactivates the trigger, VERIFY_FUNDS, and specifies that it fire before an UPDATE instead of after:

```
ALTER TRIGGER SAVE_SALARY_CHANGE
   ACTIVE
   BEFORE UPDATE;
```

Altering a trigger body

When a trigger body is altered, the new body definition replaces the old definition. When used to change only a trigger body, ALTER TRIGGER need contain any header information other than the trigger's name.

To make changes to a trigger body:

1. Copy the original data definition file used to create the trigger. Alternatively, use **isql -extract** to extract a trigger from the database to a file.

2. Edit the file, changing CREATE to ALTER, and delete all trigger header information after the trigger name and before the AS keyword.

3. Change the trigger definition as desired. Retain whatever is still useful. The trigger body must remain syntactically and semantically complete.

For example, the following ALTER statement modifies the previously introduced trigger, SET_CUST_NO, to insert a row into the (assumed to be previously defined) table, NEW_CUSTOMERS, for each new customer.

```
SET TERM !! ;
ALTER TRIGGER SET_CUST_NO
BEFORE INSERT AS
BEGIN
   new.cust_no = GEN_ID(CUST_NO_GEN, 1);
   INSERT INTO NEW_CUSTOMERS(new.cust_no, TODAY)
END !!
SET TERM ; !!
```

Dropping triggers

During database design and application development, a trigger may no longer be useful. To permanently remove a trigger, use DROP TRIGGER.

The following restrictions apply to dropping triggers:

- Only the creator of a trigger can drop it.

- Triggers currently in use cannot be dropped.

To temporarily remove a trigger, use ALTER TRIGGER and specify INACTIVE in the header.

The DROP TRIGGER syntax is as follows:

```
DROP TRIGGER name;
```

The trigger *name* must be the name of an existing trigger. The following example drops the trigger, SET_CUST_NO:

```
DROP TRIGGER SET_CUST_NO;
```

Note You cannot drop a trigger if it is in use by a CHECK constraint (a system-defined trigger). Use ALTER TABLE to remove or modify the CHECK clause that defines the trigger.

Using triggers

Triggers are a powerful feature with a variety of uses. Among the ways that triggers can be used are:

- To make correlated updates. For example, to keep a log file of changes to a database or table.

- To enforce data restrictions, so that only valid data is entered in tables.

- Automatic transformation of data. For example, to automatically convert text input to uppercase.

- To notify applications of changes in the database using event alerters.

- To perform cascading referential integrity updates.

Triggers are stored as part of a database, like stored procedures and exceptions. Once defined to be ACTIVE, they remain active until deactivated with ALTER TRIGGER or removed from the database with DROP TRIGGER.

A trigger is never explicitly called. Rather, an active trigger automatically fires when the specified action occurs on the specified table.

IMPORTANT If a trigger performs an action that causes it to fire again—or fires another trigger that performs an action that causes it to fire—an infinite loop results. For this reason, it is important to ensure that a trigger's actions never cause the trigger to fire, even indirectly. For example, an endless loop will occur if a trigger fires on INSERT to a table and then performs an INSERT into the same table.

Triggers and transactions

Triggers operate within the context of the transaction in the program where they are fired. Triggers are considered part of the calling program's current unit of work.

If triggers are fired in a transaction, and the transaction is rolled back, then any actions performed by the triggers are also rolled back.

Triggers and security

Triggers can be granted privileges on tables, just as users or procedures can be granted privileges. Use the GRANT statement, but instead of using TO *username*, use TO TRIGGER *trigger_name*. Triggers' privileges can be revoked similarly using REVOKE. For more information about GRANT and REVOKE, see Chapter 13, "Planning Security."

When a user performs an action that fires a trigger, the trigger will have privileges to perform its actions if:

- The trigger has privileges for the action.
- The user has privileges for the action.

So, for example, if a user performs an UPDATE of table A, which fires a trigger, and the trigger performs an INSERT on table B, the INSERT will occur if the user has INSERT privileges on the table or the trigger has insert privileges on the table.

If there are insufficient privileges for a trigger to perform its actions, InterBase will set the appropriate SQLCODE error number. The trigger can handle this error with a WHEN clause. If it does not handle the error, an error message will be returned to the application, and the actions of the trigger and the statement which fired it will be undone.

Triggers as event alerters

Triggers can be used to post events when a specific change to the database occurs. For example, the following trigger, POST_NEW_ORDER, posts an event named "new_order" whenever a new record is inserted in the SALES table:

```
SET TERM !! ;
CREATE TRIGGER POST_NEW_ORDER FOR SALES
AFTER INSERT AS
BEGIN
    POST_EVENT "new_order";
END !!
SET TERM ; !!
```

In general, a trigger can use a variable for the event name:

```
POST_EVENT :event_name;
```

The parameter, *event_name,* is declared as a string variable, the statement could post different events, depending on the value of the string variable, *event_name.* Then, for example, an application can wait for the event to occur, if the event has been declared with EVENT INIT and then instructed to wait for it with EVENT WAIT:

```
EXEC SQL
EVENT INIT order_wait empdb ("new_order")
EXEC SQL
EVENT WAIT order_wait;
```

For more information on event alerters, see the *Programmer's Guide.*

Updating views with triggers

Views that are based on joins—including reflexive joins—and on aggregates cannot be updated directly. You can, however, write triggers that will perform the correct writes to the base tables when a DELETE, UPDATE, or INSERT is performed on the view. This InterBase feature turns non-updatable views into updatable views.

TIP You can specify nondefault behavior for updatable views, as well. InterBase does not perform writethroughs on any view that has one or more triggers defined on it. This means that you can have complete control of what happens to any base table when users modify a view based on it.

For more information about updating and read-only views, see "Types of views: read-only and updatable" on page 111.

The following example creates two tables, creates a view that is a join of the two tables, and then creates three triggers—one each for DELETE, UPDATE, and INSERT—that will pass all updates on the view through to the underlying base tables.

```
CREATE TABLE Table1 (
    ColA INTEGER NOT NULL,
    ColB VARCHAR(20),
    CONSTRAINT pk_table PRIMARY KEY(ColA)
  );

CREATE TABLE Table2 (
    ColA INTEGER NOT NULL,
    ColC VARCHAR(20),
    CONSTRAINT fk_table2 FOREIGN KEY REFERENCES Table1(ColA)
);

CREATE VIEW TableView AS
    SELECT Table1.ColA,
        Table1.ColB,
        Table2.ColC
    FROM Table1, Table2
    WHERE Table1.ColA = Table2.ColA;

CREATE TRIGGER TableView_Delete FOR TableView BEFORE DELETE AS
BEGIN
    DELETE FROM Table1
    WHERE ColA = OLD.ColA;
    DELETE FROM Table2
    WHERE ColA = OLD.ColA;
END;

CREATE TRIGGER TableView_Update FOR TableView BEFORE UPDATE AS
BEGIN
    UPDATE Table1
    SET ColB = NEW.ColB
    WHERE ColA = OLD.ColA;
    UPDATE Table2
    SET ColC = NEW.ColC
    WHERE ColA = OLD.ColA;
END;

CREATE TRIGGER TableView_Insert FOR TableView BEFORE INSERT AS
BEGIN
    INSERT INTO Table1 values (
```

```
        NEW.ColA,NEW.ColB);
    INSERT INTO Table2 values (
        NEW.ColA,NEW.ColC);
END;
```

Exceptions

An *exception* is a named error message that can be raised from a trigger or a stored procedure. Exceptions are created with CREATE EXCEPTION, modified with ALTER EXCEPTION, and removed from the database with DROP EXCEPTION. For more information about these statements, see Chapter 9, "Working with Stored Procedures."

When raised in a trigger, an exception returns an error message to the calling program and terminates the trigger, unless the exception is handled by a WHEN statement in the trigger. For more information on error handling with WHEN, see Chapter 9, "Working with Stored Procedures."

For example, a trigger that fires when the EMPLOYEE table is updated might compare the employee's old salary and new salary, and raise an exception if the salary increase exceeds 50%. The exception could return an message such as:

```
New salary exceeds old by more than 50%. Cannot update record.
```

IMPORTANT   Like procedures and triggers, exceptions are created and stored in a database, where they can be used by any procedure or trigger in the database. Exceptions must be created and committed before they can be used in triggers.

Raising an exception in a trigger

To raise an existing exception in a trigger, use the following syntax:

```
EXCEPTION name;
```

Where *name* is the name of an exception that already exists in the database. Raising an exception:

- Terminates the trigger, undoing any changes caused (directly or indirectly) by the trigger.
- Returns the exception message to the application which performed the action that fired the trigger. If an **isql** command fired the trigger, the error message is displayed on the screen.

Note If an exception is handled with a WHEN statement, it will behave differently. For more information on exception handling, see Chapter 9, "Working with Stored Procedures."

For example, suppose an exception is created as follows:

```
CREATE EXCEPTION RAISE_TOO_HIGH "New salary exceeds old by more than 50%.
Cannot update record.";
```

The trigger, SAVE_SALARY_CHANGE, might raise the exception as follows:

```
SET TERM !! ;
CREATE TRIGGER SAVE_SALARY_CHANGE FOR EMPLOYEE
AFTER UPDATE AS
DECLARE VARIABLE pcnt_raise;
BEGIN
    pcnt_raise = (new.salary - old.salary) * 100 / old.salary;
        IF (old.salary <> new.salary) THEN
            IF (pcnt_raise > 50) THEN
                EXCEPTION RAISE_TOO_HIGH;
            ELSE
                BEGIN
                    INSERT INTO SALARY_HISTORY (EMP_NO, CHANGE_DATE,
                        UPDATER_ID, OLD_SALARY, PERCENT_CHANGE)
                        VALUES (old.emp_no, "now", USER, old.salary,
                        pcnt_raise);
END !!
SET TERM ; !!
```

Error handling in triggers

Errors and exceptions that occur in triggers may be handled using the WHEN statement. If an exception is handled with WHEN, the exception does not return a message to the application and does not necessarily terminate the trigger.

Error handling in triggers works the same as for stored procedures: the actions performed in the blocks up to the error-handling (WHEN) statement are undone and the statements specified by the WHEN statement are performed.

For more information on error handling with WHEN, see Chapter 9, "Working with Stored Procedures."

11

Declaring User-Defined Functions and BLOB Filters

User-defined functions (UDFs) are host-language programs for performing customized, often-used tasks in applications. UDFs enable the programmer to modularize an application by separating it into more reusable and manageable units.

BLOB filters are host-language programs that convert BLOB data from one format to another.

You can access UDFs and BLOB filters through **isql** or a host-language program. You can also access UDFs in stored procedures and trigger definitions.

IMPORTANT   UDFs and BLOB filters are not supported on NetWare servers.

Creating user-defined functions

The steps in creating a UDF are explained in detail in the *Programmer's Guide*. Essentially, you must code and compile the UDF in a host language, then build a shared function library containing the UDF. The only data definition task associated with UDFs is declaring them to a database with DECLARE EXTERNAL FUNCTION.

InterBase 5.0's single multi-threaded process requires some modification in the way memory is allocated and released in user-defined functions (UDFs) and in the way these UDFs are declared. In the single-process, multi-thread architecture, memory allocated dynamically is not released, since the process does not end. In addition, users running UDFs concurrently will conflict in their use of the same static memory space.

UDFs written in C or C++ must allocate memory using **malloc** rather than static arrays in order to be thread-safe. You can use local variables only if you can guarantee that only one user at a time will be accessing UDFs.

InterBase's new FREE_IT keyword allows InterBase users to write thread-safe UDF functions without memory leaks.

DECLARE EXTERNAL FUNCTION syntax

Whenever a UDF returns a value by reference to dynamically allocated memory, you must declare it using the new FREE_IT keyword in order to free the allocated memory.

Note You must not use FREE_IT with UDFs that return a pointer to static data, as in the "multi-process version" example below.

```
DECLARE EXTERNAL FUNCTION name [<datatype> | CSTRING (int)
    [, <datatype> | CSTRING (int) ...]]
RETURNS {<datatype> [BY VALUE] | CSTRING (int)} [FREE_IT]
ENTRY_POINT "<entryname>"
MODULE_NAME "<modulename>";
```

The following table explains DECLARE EXTERNAL FUNCTION arguments:

Argument	Description
name	Name of the UDF.
<datatype>	Datatype of an input or return parameter. All input parameters are passed to a UDF by reference. Return parameters can be passed by value. Cannot be an array element.
RETURNS	Specifies the return value of a function.

TABLE 11.1   Arguments to DECLARE EXTERNAL FUNCTION

Argument	Description
BY VALUE	Specifies that a return value should be passed by value, rather than by reference.
CSTRING (*int*)	Specifies a UDF that returns a null-terminated string *int* bytes in length.
FREE_IT	Frees memory of the return value after the UDF finishes running; use this directive only if the memroy is allocated dynamically in the UDF with malloc or similar mechanism. See the *Language Reference*, Chapter 5.
"*<entryname>*"	Quoted string specifying the name of the UDF as stored in the UDF library.
"*<modulename>*"	Quoted file specification identifying the object module in which the UDF resides.

TABLE 11.1   Arguments to DECLARE EXTERNAL FUNCTION

name specifies the name of the UDF to use in SQL statements. It can be different from the name of the function specified after the ENTRY_POINT keyword. "*entryname*" is the name of the function in the source code. "*modulename*" is the name of the object module containing the function definition.

DECLARE EXTERNAL FUNCTION example

The following statement declares the TOPS() function to a database:

```
DECLARE EXTERNAL FUNCTION TOPS
CHAR(256), INTEGER, BLOB
RETURNS INTEGER BY VALUE
ENTRY_POINT 'te1' MODULE_NAME 'tm1';
```

The following code shows how to use the FREE_IT keyword:

```
DECLARE EXTERNAL FUNCTION lowers VARCHAR(256)
RETURNS CSTRING(256) FREE_IT
ENTRY POINT 'fn_lower' MODULE_NAME 'udflib.lib'
```

Declaring BLOB filters

You can use BLOB filters to convert data from one BLOB subtype to another. You can access BLOB filters from any host-language program that contains SQL statements.

BLOB filters are user-written utility programs that convert data in columns of BLOB datatype from one InterBase or user-defined subtype to another. Declare the filter to the database with the DECLARE FILTER statement. For example:

```
DECLARE FILTER BLOB_FORMAT
    INPUT_TYPE 1 OUTPUT_TYPE -99
    ENTRY_POINT "Text_filter" MODULE_NAME "Filter_99";
```

InterBase invokes BLOB filters through SQL statements embedded in a host-language program or interactively through **isql**.

isql automatically uses a built-in ASCII BLOB filter for a BLOB defined without a subtype, when asked to display the BLOB. It also automatically filters BLOB data defined with subtypes to text, if the appropriate filters have been defined.

To use BLOB filters, follow these steps:

1. Write the filters and compile them into object code.

2. Create a shared filter library.

3. Make the filter library available to InterBase at run time.

4. Define the filters to the database using DECLARE FILTER.

5. Write an application that requests filtering.

You can use BLOB subtypes and BLOB filters to do a large variety of processing. For example, you can define one BLOB subtype to hold:

▪ Compressed data and another to hold decompressed data. Then you can write BLOB filters for expanding and compressing BLOB data.

▪ Generic code and other BLOB subtypes to hold system-specific code. Then you can write BLOB filters that add the necessary system-specific variations to the generic code.

▪ Word processor input and another to hold word processor output. Then you can write a BLOB filter that invokes the word processor.

For more information about creating and using BLOB filters, see the *Programmer's Guide*. For the complete syntax of DECLARE FILTER, see the *Language Reference*.

12

Working with Generators

This chapter explains how to create, alter, and use database generators.

About generators

A *generator* is a mechanism that creates a unique, sequential number that is automatically inserted into a column by the database when SQL data manipulation operations such as INSERT or UPDATE occur. Generators are typically used to produce unique values that can be inserted into a column that is used as a PRIMARY KEY. For example, a programmer writing an application to log and track invoices may want to ensure that each invoice number entered into the database is unique. The programmer can use a generator to create the invoice numbers automatically, rather than writing specific application code to accomplish this task.

Any number of generators can be defined for a database, as long as each generator has a unique name. A generator is global to the database where it is declared. Any transaction that activates the generator can or update the current sequence number. InterBase will not assign duplicate generator values across transactions.

Creating generators

To create a unique number generator in the database, use the CREATE GENERATOR statement. CREATE GENERATOR declares a generator to the database and sets its starting value to zero (the default). If you want to set the starting value for the generator to a number other than zero, use SET GENERATOR to specify the new value.

The syntax for CREATE GENERATOR is:

```
CREATE GENERATOR name;
```

The following statement creates the generator, EMPNO_GEN:

```
CREATE GENERATOR EMPNO_GEN;
```

Note Once defined, a generator cannot be deleted.

Setting or resetting generator values

SET GENERATOR sets a starting value for a newly created generator, or resets the value of an existing generator. The new value for the generator, *int*, can be an integer from -2^{31} to $2^{31}-1$. When the GEN_ID() function is called, that value is *int* plus the increment specified in the GEN_ID() *step* parameter.

The syntax for SET GENERATOR is:

```
SET GENERATOR name TO int;
```
The following statement sets a generator value to 1,000:

```
SET GENERATOR CUST_NO_GEN TO 1000;
```

IMPORTANT Don't reset a generator unless you are certain that duplicate numbers will not occur. For example, a generators are often used to assign a number to a column that has PRIMARY KEY or UNIQUE integrity constraints. If you reset such a generator so that it generates duplicates of existing column values, all subsequent insertions and updates fail with a "Duplicate key" error message.

Using generators

After creating the generator, the data definition statements that make the specific number generator known to the database have been defined; no numbers have been generated yet. To invoke the number generator, you must call the InterBase GEN_ID() function. GEN_ID() takes two arguments: the name of the generator to call, which must already be defined for the database, and a step value, indicating the amount by which the current value should be incremented (or decremented, if the value is negative). GEN_ID() can be called from within a trigger, a stored procedure, or an application whenever an INSERT, UPDATE, or DELETE operation occurs.

The syntax for GEN_ID() is:

```
GEN_ID(genname, step);
```

GEN_ID() can be called directly from within an application or stored procedure using INSERT, UPDATE, or DELETE statements. For example, the following statement uses GEN_ID() to call the generator, *g,* to increment a purchase order number in the SALES table by one:

```
INSERT INTO SALES (PO_NUMBER) VALUES (GEN_ID(g,1));
```

A number is generated by the following sequence of events:

1. The generator is created and stored in the database.

2. A trigger, stored procedure, or application references the generator with a call to GEN_ID().

3. A generator returns a value when a trigger fires, or when a stored procedure or application executes. It is up to the trigger, stored procedure, or application to use the value. For example, a trigger can insert the value into a column.

For more information on using generators in triggers, see Chapter 10, "Creating Triggers." For more information on using generators in stored procedures, see Chapter 9, "Working with Stored Procedures."

To stop inserting a generated number in a database column, delete or modify the trigger, stored procedure, or application so that it no longer invokes GEN_ID().

Note There is no "drop generator" statement. To remove a generator, delete it from the system table. For example:

```
DELETE FROM RDB$GENERATORS WHERE RDB$GENERATORS_NAME = 'EMP_NO';
```

13

Planning Security

This chapter describes the following:

- Available SQL access privileges.
- Granting access to a table.
- Granting privileges to execute stored procedures.
- Granting access to views.
- Revoking access to tables and views.
- Using views to restrict data access.
- Providing additional security.

Overview of SQL access privileges

SQL security is controlled at the table level with *access privileges*, a list of operations that a user is allowed to perform on a given table or view. The GRANT statement assigns access privileges for a table or view to specified users, to a role, or to objects such as stored procedures or triggers. GRANT can also enable users or stored procedures to execute stored procedures through the EXECUTE privilege and can grant roles to users. Use REVOKE to remove privileges assigned through GRANT.

GRANT can be used in the following ways:

- Grant SELECT, INSERT, UPDATE, DELETE, and REFERENCES privileges for a table to users, triggers, stored procedures, or views (optionally WITH GRANT OPTION)

- Grant SELECT, INSERT, UPDATE, and DELETE privileges for a view to users, triggers, stored procedures, or views (optionally WITH GRANT OPTION)

- Grant SELECT, INSERT, UPDATE, DELETE, and REFERENCES privileges for a table to a role

- Grant SELECT, INSERT, UPDATE, and DELETE privileges for a view to a role

- Grant a role to users (optionally WITH ADMIN OPTION)

- Grant EXECUTE permission on a stored procedure to users, triggers, stored procedures, or views (optionally WITH GRANT OPTION)

Default security and access

All tables and stored procedures are secured against unauthorized access when they are created. Initially, only a table's creator, its *owner*, has access to a table, and only its owner can use GRANT to assign privileges to other users or to procedures. Only a procedure's creator, its owner, can execute or call the procedure, and only its owner can assign EXECUTE privilege to other users or to other procedures.

InterBase also supports a SYSDBA user who has access to all database objects; furthermore, on platforms that support the concept of a superuser, or user with root or locksmith privileges, such a user also has access to all database objects.

Privileges available

The following table lists the SQL access privileges that can be granted and revoked:

Privilege	Access
ALL	SELECT, DELETE, INSERT, and UPDATE data
SELECT	Read data
DELETE	Delete data

TABLE 13.1   SQL access privileges

Privilege	Access
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 13.1   SQL access privileges

The privileges assigned by ALL include SELECT, DELETE, INSERT, UPDATE, and REFERENCES privileges using a single keyword. ALL does not grant a role or the EXECUTE privilege. SELECT, DELETE, INSERT, UPDATE, and REFERENCES privileges can also be granted or revoked singly or in combination.

Note Statements that grant or revoke either the EXECUTE privilege or a role cannot grant or revoke other privileges.

SQL ROLES

InterBase 5 implements features for assigning SQL privileges to groups of users, fully supporting SQL group-level security as described in the *ISO-ANSI Working Draft for Database Language* SQL sections 11.54. role definition, 11.53. GRANT statement, 11.58. REVOKE statement, and 11.57. DROP ROLE statement. It partially supports section 11.55 GRANT ROLE and 11.56 REVOKE ROLE.

Note These features replace the Security Classes feature in past versions of InterBase. In the past, group privileges could be granted only through the InterBase-proprietary GDML language. In Version 5, new SQL features have been added to assist in migrating InterBase users from GDML to SQL.

Using roles

Implementing roles is a four-step process.

1. Create a role using the CREATE ROLE statement.

2. Assign privileges to the role using GRANT *privilege* TO *rolename*.

3. Grant the role to users using GRANT *rolename* TO *user*.

4. Specify the role when attaching to a database.

These steps are described in detail in this chapter. In addition, the CONNECT, CREATE ROLE, GRANT, and REVOKE statements are described in the *Language Reference*.

Granting privileges

You can grant access privileges on an entire table or view or to only certain columns of the table or view. This section discusses the basic operation of granting privileges.

- Granting multiple privileges at one time, or granting privileges to groups of users is discussed in "Multiple privileges and multiple grantees" on page 183.
- "Using roles to grant privileges" on page 185 discusses both how to grant privileges to roles and how to grant roles to users.
- You can grant access privileges to views, but there are limitations. See "Granting access to views" on page 189.
- The power to grant GRANT authority is discussed in "Granting users the right to grant privileges" on page 187.
- Granting EXECUTE privileges on stored procedures is discussed in "Granting privileges to execute stored procedures" on page 189.

Granting privileges to a whole table

Use GRANT to give a user or object privileges to a table, view, or role. At a minimum, GRANT requires the following parameters:

- An access privilege
- The table to which access is granted
- The name of a user to whom the privilege is granted

The access privileges can be one or more of SELECT, INSERT, UPDATE, DELETE, REFERENCE. The privilege granted can also be a *role* to which one or more privileges have been assigned.

The user name is typically a user is the InterBase security database, *isc4.gdb*, but on UNIX systems can also be a user who is in */etc/password* on both the server and client machines. In addition, you can grant privileges to a stored procedure, trigger, or role.

The syntax for granting privileges to a table is:

```
GRANT{
    <privileges> ON [TABLE] {tablename | viewname}
        TO {<object> | <userlist> | GROUP UNIX_group}
        | <role_granted> TO {PUBLIC | <role_grantee_list>}};

<privileges> = {ALL [PRIVILEGES] | <privilege_list>}

<privilege_list> = {
    SELECT
    | DELETE
    | INSERT
    | UPDATE [(col [, col …])]
    | REFERENCES [(col [, col …])]
    [, <privilege_list> …]}}

<object> = {
    PROCEDURE procname
    | TRIGGER trigname
    | VIEW viewname
    | PUBLIC
    [, <object> …]}

<userlist> = {
    [USER] username
    | rolename
    | Unix_user}
    [, <userlist> …]
    [WITH GRANT OPTION]

<role_granted> = rolename [, rolename …]

<role_grantee_list> = [USER] username [, [USER] username …]
    [WITH ADMIN OPTION]
```

Notice that this syntax includes the provisions for restricting UPDATE or REFERENCES to certain columns, discussed on the next section, "Granting access to columns in a table."

The following statement grants SELECT privilege for the DEPARTMENTS table to a user, EMIL:

```
GRANT SELECT ON DEPARTMENTS TO EMIL;
```

The next example grants REFERENCES privileges on DEPARTMENTS to EMIL, permitting EMIL to create a foreign key that references the primary key of the DEPARTMENTS table, even though he doesn't own that table:

```
GRANT REFERENCES ON DEPARTMENTS(DEPT_NO) TO EMIL;
```

TIP   Views offer a way to further restrict access to tables, by restricting either the columns or the rows that are visible to the user. See Chapter 8, "Working with Views" for more information.

Granting access to columns in a table

In addition to assigning access rights for an entire table, GRANT can assign UPDATE or REFERENCES privileges for certain columns of a table or view. To specify the columns, place the comma-separated list of columns in parentheses following the privileges to be granted in the GRANT statement.

The following statement assigns UPDATE access to all users for the CONTACT and PHONE columns in the CUSTOMERS table:

```
GRANT UPDATE (CONTACT, PHONE) ON CUSTOMERS TO PUBLIC;
```

You can add to the rights already assigned to users at the table level, but you cannot subtract from them. To restrict user access to a table, use the REVOKE statement.

Granting privileges to a stored procedure or trigger

A stored procedure, view, or trigger sometimes needs privileges to access a table or view that has a different owner. To grant privileges to a stored procedure, put the PROCEDURE keyword before the procedure name. Similarly, to grant privileges to a trigger or view, put the TRIGGER or VIEW keyword before the object name.

IMPORTANT   When a trigger, stored procedure or view needs to access a table or view, it is sufficient for either the accessing object or the user who is executing it to have the necessary permissions.

The following statement grants the INSERT privilege for the ACCOUNTS table to the procedure, MONEY_TRANSFER:

```
GRANT INSERT ON ACCOUNTS TO PROCEDURE MONEY_TRANSFER;
```

TIP   As a security measure, privileges to tables can be granted to a procedure instead of to individual users. If a user has EXECUTE privilege on a procedure that accesses a table, then the user does not need privileges to the table.

Multiple privileges and multiple grantees

This section discusses ways to grant several privileges at one time, and ways to grant one or more privileges to multiple users or objects.

Granting multiple privileges

To give a user several privileges on a table, separate the granted privileges with commas in the GRANT statement. For example, the following statement assigns INSERT and UPDATE privileges for the DEPARTMENTS table to a user, LI:

```
GRANT INSERT, UPDATE ON DEPARTMENTS TO LI;
```

To grant a set of privileges to a procedure, place the PROCEDURE keyword before the procedure name. Similarly, to grant privileges to a trigger or view, precede the object name with the TRIGGER or VIEW keyword.

The following statement assigns INSERT and UPDATE privileges for the ACCOUNTS table to the MONEY_TRANSFER procedure:

```
GRANT INSERT, UPDATE ON ACCOUNTS TO PROCEDURE MONEY_TRANSFER;
```

The GRANT statement can assign any combination of SELECT, DELETE, INSERT, UPDATE, and REFERENCES privileges. EXECUTE privileges must be assigned in a separate statement.

Note REFERENCES privileges cannot be assigned for views.

Granting all privileges

The ALL privilege combines the SELECT, DELETE, INSERT, UPDATE, and REFERENCES privileges for a table in a single expression. It is a shorthand way to assign that group of privileges to a user or procedure. For example, the following statement grants all access privileges for the DEPARTMENTS table to a user, SUSAN:

```
GRANT ALL ON DEPARTMENTS TO SUSAN;
```

SUSAN can now perform SELECT, DELETE, INSERT, UPDATE, and REFERENCES operations on the DEPARTMENTS table.

Procedures can be assigned ALL privileges. When a procedure is assigned privileges, the PROCEDURE keyword must precede its name. For example, the following statement grants all privileges for the ACCOUNTS table to the procedure, MONEY_TRANSFER:

```
GRANT ALL ON ACCOUNTS TO PROCEDURE MONEY_TRANSFER;
```

Granting privileges to multiple users

There are a number of techniques available for granting privileges to multiple users. You can grant the privileges to a list of users, to a UNIX group, or to all users (PUBLIC). In addition, you can assign privileges to a role, which you then assign to a user list, a UNIX group, or to PUBLIC.

▶ *Granting privileges to a list of users*

To assign the same access privileges to a number of users at the same time, provide a comma-separated list of users in place of the single user name. For example, the following statement gives INSERT and UPDATE privileges for the DEPARTMENTS table to users FRANCIS, BEATRICE, and HELGA:

```
GRANT INSERT, UPDATE ON DEPARTMENTS TO FRANCIS, BEATRICE, HELGA;
```

▶ *Granting privileges to a UNIX group*

OS-level account names are implicit in InterBase security on UNIX. A client running as a UNIX user adopts that user identity in the database, even if the account is not defined in the InterBase security database (*isc4.gdb*). Now OS-level groups share this behavior, and database administrators can assign SQL privileges to UNIX groups through SQL GRANT/REVOKE statements. This allows any OS-level account that is a member of the group to inherit the privileges that have been given to the group. For example:

```
GRANT UPDATE ON table1 TO GROUP group_name;
```

where *group_name* is a UNIX-level group defined in */etc/group*.

Note Integration of UNIX groups with database security is not an SQL standard feature.

▶ *Granting privileges to all users*

To assign the same access privileges for a table to all users, use the PUBLIC keyword rather than listing users individually in the GRANT statement.

The following statement grants SELECT, INSERT, and UPDATE privileges on the DEPARTMENTS table to all users:

```
GRANT SELECT, INSERT, UPDATE ON DEPARTMENTS TO PUBLIC;
```

IMPORTANT PUBLIC grants privileges only to users, not to stored procedures, triggers, roles, or views. Privileges granted to users with PUBLIC can only be revoked from PUBLIC.

Granting privileges to a list of procedures

To assign privileges to a several procedures at once, provide a comma-separated list of procedures following the word PROCEDURE in the GRANT statement.

The following statement gives INSERT and UPDATE privileges for the DEPARTMENTS table to the procedures, ACCT_MAINT, and MONEY_TRANSFER:

```
GRANT INSERT, UPDATE ON DEPARTMENTS TO PROCEDURE ACCT_MAINT,
    MONEY_TRANSFER;
```

Using roles to grant privileges

In InterBase 5, you can assign privileges through the use of ROLEs. Acquiring privileges through a role is a four-step process.

1. Create a role using the CREATE ROLE statement.

   ```
   CREATE ROLE rolename;
   ```

2. Assign one or more privileges to that role using GRANT.

   ```
   GRANT privilegelist TO rolename;
   ```

3. Use the GRANT statement once again to grant the role to one or more users.

   ```
   GRANT rolename ON table TO userlist;
   ```

   The role can be granted WITH ADMIN OPTION, which allows users to grant the role to others, just as the WITH GRANT OPTION allows users to grant privileges to others.

4. At connection time, specify the role whose privileges you want to acquire for that connection.

   ```
   CONNECT "database" USER "username" PASSWORD "password" ROLE "rolename";
   ```

Use REVOKE to remove privileges that have been granted to a role or to remove roles that have been granted to users.

See the *Language Reference* for more information on CONNECT, CREATE ROLE, GRANT, and REVOKE.

Granting privileges to a role

Once a role has been defined, you can grant privileges to that role, just as you would to a user.

The syntax is as follows:

```
GRANT <privileges> ON [TABLE] {tablename | viewname}
      TO rolename;

<privileges> = {ALL [PRIVILEGES] | <privilege_list>}

<privilege_list> = {
    SELECT
    | DELETE
    | INSERT
    | UPDATE [(col [, col …])]
    | REFERENCES [(col [, col …])]
    [, <privilege_list> …]}}
```

See "Granting a role to users" for an example of creating a role, granting privileges to it, and then granting the role to users.

Granting a role to users

When a role has been defined and has been granted privileges, you can grant that role to one or more users, who then acquire the privileges that have been assigned to the role.

To permit users to grant the role to others, add WITH ADMIN OPTION to the GRANT statement when you grant the role to the users.

The syntax is as follows:

```
GRANT {rolename [, rolename …]} TO {PUBLIC
    | {[USER] username [, [USER] username …]} }[WITH ADMIN OPTION];
```

The following example creates the DOITALL role, grants ALL privileges on DEPARTMENTS to this role, and grants the DOITALL role to RENEE, who then has SELECT, DELETE, INSERT, UPDATE, and REFERENCES privileges on DEPARTMENTS.

```
CREATE ROLE DOITALL;
GRANT ALL ON DEPARTMENTS TO DOITALL;
GRANT DOITALL TO RENEE;
```

Granting users the right to grant privileges

Initially, only the owner of a table or view can grant access privileges on the object to other users. The WITH GRANT OPTION clause transfers the right to grant privileges to other users.

To assign grant authority to another user, add the WITH GRANT OPTION clause to the end of a GRANT statement.

The following statement assigns SELECT access to user EMIL and allows EMIL to grant SELECT access to other users:

```
GRANT SELECT ON DEPARTMENTS TO EMIL WITH GRANT OPTION;
```

Note You cannot assign the WITH GRANT OPTION to a stored procedure.

WITH GRANT OPTION clauses are cumulative, even if issued by different users. For example, EMIL can be given grant authority for SELECT by one user, and grant authority for INSERT by another user. For more information about cumulative privileges, see "Grant authority implications" on page 188.

Grant authority restrictions

There are only three conditions under which a user can grant access privileges (SELECT, DELETE, INSERT, UPDATE, and REFERENCES) for tables to other users or objects:

- Users can grant privileges to any table or view that they own.
- Users can grant any privileges on another owner's table or view when they have been assigned those privileges WITH GRANT OPTION.
- Users can grant privileges that they have acquired by being granted a role WITH ADMIN OPTION.

For example, in an earlier GRANT statement, EMIL was granted SELECT access to the DEPARTMENTS table WITH GRANT OPTION. EMIL can grant SELECT privilege to other users. Suppose EMIL is now given INSERT access as well, but *without* the WITH GRANT OPTION:

```
GRANT INSERT ON DEPARTMENTS TO EMIL;
```

EMIL can SELECT from and INSERT to the DEPARTMENTS table. He can grant SELECT privileges to other users, but *cannot* assign INSERT privileges.

To change a user's existing privileges to include grant authority, issue a second GRANT statement that includes the WITH GRANT OPTION clause. For example, to allow EMIL to grant INSERT privileges on DEPARTMENTS to others, reissue the GRANT statement and include the WITH GRANT OPTION clause:

```
GRANT INSERT ON DEPARTMENTS TO EMIL WITH GRANT OPTION;
```

Grant authority implications

Consider every extension of grant authority with care. Once other users are permitted grant authority on a table, they can grant those same privileges, as well as grant authority for them, to other users.

As the number of users with privileges and grant authority for a table increases, the likelihood that different users can grant the same privileges and grant authority to any single user also increases.

SQL permits duplicate privilege and authority assignment under the assumption that it is intentional. Duplicate privilege and authority assignments to a single user have implications for subsequent revocation of that user's privileges and authority. For more information about revoking privileges, see "Revoking user access" on page 192.

For example, suppose two users to whom the appropriate privileges and grant authority have been extended, GALENA and SUDHANSHU, both issue the following statement:

```
GRANT INSERT ON DEPARTMENTS TO SPINOZA WITH GRANT OPTION;
```

Later, GALENA revokes the privilege and grant authority for SPINOZA:

```
REVOKE INSERT ON DEPARTMENTS FROM SPINOZA;
```

GALENA now believes that SPINOZA no longer has INSERT privilege and grant authority for the DEPARTMENTS table. The immediate net effect of the statement is negligible because SPINOZA retains the INSERT privilege and grant authority assigned by SUDHANSHU.

When full control of access privileges on a table is desired, grant authority should not be assigned indiscriminately. In cases where privileges must be universally revoked for a user who might have received rights from several users, there are two options:

▪ Each user who assigned rights must issue an appropriate REVOKE statement.

▪ The table's owner must issue a REVOKE statement for all users of the table, then issue GRANT statements to reestablish access privileges for the users who should not lose their rights.

For more information about the REVOKE statement, see "Revoking user access" on page 192.

Granting privileges to execute stored procedures

To use a stored procedure, users or other stored procedures must have EXECUTE privilege for it, using the following GRANT syntax:

```
GRANT EXECUTE ON PROCEDURE procname TO {<object> | <userlist>}

<object> = {
    PROCEDURE procname
    | TRIGGER trigname
    | VIEW viewname
    | PUBLIC
    [, <object> …]}

<userlist> = {
    [USER] username
    | rolename
    | Unix_user}
    [, <userlist> …]
    [WITH GRANT OPTION]
```

You must give EXECUTE privileges on a stored procedure to any procedure or trigger that calls that stored procedure if the caller's owner is not the same as the owner of the called procedure.

Note If you grant privileges to PUBLIC, you cannot specify additional users or objects as grantees in the same statement.

The following statement grants EXECUTE privilege for the FUND_BALANCE procedure to two users, NKOMO, and SUSAN, and to two procedures, ACCT_MAINT, and MONEY_TRANSFER:

```
GRANT EXECUTE ON PROCEDURE FUND_BALANCE TO NKOMO, SUSAN, PROCEDURE
    ACCT_MAINT, MONEY_TRANSFER;
```

Granting access to views

To a user, a view looks—and often acts—just like a table. However, there are significant differences: the contents of a view are not stored anywhere in the database. All that is stored is the query on the underlying base tables. Because of this, any UPDATE, DELETE, INSERT to a view is actually a write to the table on which the view is based.

Any view that is based on a join or an aggregate is considered to be a *read-only* or *non-updatable* view, since it is not directly updateable. Views that are based on a single table which have no aggregates or reflexive joins are often updatable. See "Types of views: read-only and updatable" on page 111 for more information about this topic.

IMPORTANT   It is meaningful to grant INSERT, UPDATE, and DELETE privileges for a view *only* if the view is updatable. Although you can grant the privileges to a read-only view without receiving an error message, any actual write operation fails because the view is read-only. SELECT privileges can be granted on a view just as they are on a table, since reading data from a view does not change anything.

You cannot assign REFERENCES privileges to views.

TIP   If you are creating a view for which you plan to grant INSERT and UPDATE privileges, use the WITH CHECK OPTION constraint so that users can update only base table rows that are accessible through the view.

Updatable views

You can assign SELECT, UPDATE, INSERT, and DELETE privileges to *updatable* views, just as you can to tables. UPDATES, INSERTS, and DELETES to a view are made to the view's base tables. You cannot assign REFERENCES privileges to a view.

The syntax for granting privileges to a view is:

```
GRANT{<privileges> ON viewname
      TO {<object> | <userlist> | GROUP UNIX_group};

<privileges> = {SELECT
    | DELETE
    | INSERT
    | UPDATE [(col [, col …])]
    [, <privilege_list> …]}}

<object> = {
    PROCEDURE procname
    | TRIGGER trigname
    | VIEW viewname
    | PUBLIC
    [, <object> …]}
```

```
<userlist> = {
    [USER] username
    | rolename
    | Unix_user}
    [, <userlist> …]
    [WITH GRANT OPTION]
```

When a view is based on a single table, data changes are made directly to the view's underlying base table.

For UPDATE, changes to the view affect only the base table columns selected through the view. Values in other columns are invisible to the view and its users and are never changed. Views created using the WITH CHECK OPTION integrity constraint can be updated only if the UPDATE statement fulfills the constraint's requirements.

For DELETE, removing a row from the view, and therefore from the base table removes all columns of the row, even those not visible to the view. If SQL integrity constraints or triggers exist for any column in the underlying table and the deletion of the row violates any of those constraints or trigger conditions, the DELETE statement fails.

For INSERT, adding a row to the view necessarily adds a row with all columns to the base table, including those not visible to the view. Inserting a row into a view succeeds only when:

- Data being inserted into the columns visible to the view meet all existing integrity constraints and trigger conditions for those columns.
- All other columns of the base table are allowed to contain NULL values.

For more information about working with views, see Chapter 8, "Working with Views."

Read-only views

When a view definition contains a join of any kind or an aggregate, it is no longer a legally updatable view, and InterBase cannot directly update the underlying tables.

Note You can use triggers to simulate updating a read-only view. Be aware, however, that any triggers you write are subject to all the integrity constraints on the base tables. To see an example of how to use triggers to "update" a read-only view, see "Updating views with triggers" on page 165.

For more information about integrity constraints and triggers, see Chapter 10, "Creating Triggers."

Revoking user access

Use the REVOKE statement to remove privileges that were assigned with the GRANT statement.

At a minimum, REVOKE requires parameters that specify the following:

- One access privilege to remove
- The table or view to which the privilege revocation applies
- The name of the grantee for which the privilege is revoked.

In its full form, REVOKE removes all the privileges that GRANT can assign.

```
REVOKE <privileges> ON [TABLE] {tablename | viewname}
       FROM {<object> | <userlist> | GROUP UNIX_group};

<privileges> = {ALL [PRIVILEGES] | <privilege_list>}

<privilege_list> = {
    SELECT
    | DELETE
    | INSERT
    | UPDATE [(col [, col …])]
    | REFERENCES [(col [, col …])]
    [, <privilege_list> …]}}

<object> ={
    PROCEDURE procname
    | TRIGGER trigname
    | VIEW viewname
    | PUBLIC
    [, <object>]}

<userlist> = [USER] username [, [USER] username …]
```

The following statement removes the SELECT privilege for the user, SUSAN, on the DEPARTMENTS table:

```
REVOKE SELECT ON DEPARTMENTS FROM SUSAN;
```

The following statement removes the UPDATE privilege for the procedure, MONEY_TRANSFER, on the ACCOUNTS table:

```
REVOKE UPDATE ON ACCOUNTS FROM PROCEDURE MONEY_TRANSER;
```

The next statement removes EXECUTE privilege for the procedure, ACCT_MAINT, on the MONEY_TRANSFER procedure:

```
REVOKE EXECUTE ON PROCEDURE MONEY_TRANSER FROM PROCEDURE ACCT_MAINT;
```

For the complete syntax of REVOKE, see the *Language Reference*.

Revocation restrictions

The following restrictions and rules of scope apply to the REVOKE statement:

- Privileges can be revoked only by the user who granted them.

- Other privileges assigned by other users are not affected.

- Revoking a privilege for a user, A, to whom grant authority was given, automatically revokes that privilege for all users to whom it was subsequently assigned by user A.

- Privileges granted to PUBLIC can only be revoked for PUBLIC.

Revoking multiple privileges

To remove some, but not all, of the access privileges assigned to a user or procedure, list the privileges to remove, separating them with commas. For example, the following statement removes the INSERT and UPDATE privileges for the DEPARTMENTS table from a user, LI:

```
REVOKE INSERT, UPDATE ON DEPARTMENTS FROM LI;
```

The next statement removes INSERT and DELETE privileges for the ACCOUNTS table from a stored procedure, MONEY_TRANSFER:

```
REVOKE INSERT, DELETE ON ACCOUNTS FROM PROCEDURE MONEY_TRANSFER;
```

Any combination of previously assigned SELECT, DELETE, INSERT, and UPDATE privileges can be revoked.

Revoking all privileges

The ALL privilege combines the SELECT, DELETE, INSERT, and UPDATE privileges for a table in a single expression. It is a shorthand way to remove all SQL table access privileges from a user or procedure. For example, the following statement revokes all access privileges for the DEPARTMENTS table for a user, SUSAN:

```
REVOKE ALL ON DEPARTMENTS FROM SUSAN;
```

Even if a user does not have all access privileges for a table, ALL can still be used. Using ALL in this manner is helpful when a current user's access rights are unknown.

Note ALL does not revoke EXECUTE privilege.

Revoking privileges for a list of users

Use a comma-separated list of users to REVOKE access privileges for a number of users at the same time.

The following statement revokes INSERT and UPDATE privileges on the DEPARTMENTS table for users FRANCIS, BEATRICE, and HELGA:

```
REVOKE INSERT, UPDATE ON DEPARTMENTS FROM FRANCIS, BEATRICE, HELGA;
```

Revoking privileges for a role

If you have granted privileges to a role or granted a role to users, you can use REVOKE to remove the privileges or the role.

To remove privileges from a role:

```
REVOKE privileges ON table FROM rolenamelist;
```

To revoke a role from users:

```
REVOKE role_granted FROM {PUBLIC | role_grantee_list};
```

The following statement revokes UPDATE privileges from the DOITALL role:

```
REVOKE UPDATE ON DEPARTMENTS FROM DOITALL;
```

Now, users who were granted the DOITALL role no longer have UPDATE privileges on DEPARTMENTS, although they retain the other privileges—SELECT, INSERT, DELETE, and REFERENCES—that they acquired with this role.

IMPORTANT   If you drop a role using the DROP ROLE statement, all privileges that were conferred by that role are revoked.

Revoking a role from users

Use REVOKE to remove a role that you assigned to users.

The following statement revokes the DOITALL role from RENEE.

```
REVOKE DOITALL FROM RENEE;
```

RENEE no longer has any of the access privileges that she acquired as a result of membership in the DOITALL role. However, if any others users have granted the same privileges to her, she still has them.

Revoking EXECUTE privileges

Use REVOKE to remove EXECUTE privileges on a stored procedure. The syntax for revoking EXECUTE privileges is as follows:

```
REVOKE EXECUTE ON PROCEDURE procname FROM {<object> | <userlist>}

<object> ={
    PROCEDURE procname
    | TRIGGER trigname
    | VIEW viewname
    | PUBLIC
    [, <object>]}

<userlist> = [USER] username [, [USER] username …]
```

The following statement removes EXECUTE privilege for user EMIL on the MONEY_TRANSFER procedure:

```
REVOKE EXECUTE ON PROCEDURE MONEY_TRANSFER FROM EMIL;
```

Revoking privileges from objects

REVOKE can remove the access privileges for one or more procedures, triggers, or views. Precede each type of object by the correct keyword (PROCEDURE, TRIGGER, or VIEW) and separate lists of one object type with commas.

The following statement revokes INSERT and UPDATE privileges for the ACCOUNTS table from the MONEY_TRANSFER and ACCT_MAINT procedures and from the SHOW_USER trigger.

```
REVOKE INSERT, UPDATE ON ACCOUNTS FROM PROCEDURE MONEY_TRANSFER,
    ACCT_MAINT TRIGGER SHOW_USER;
```

Revoking privileges for all users

To revoke privileges granted to all users as PUBLIC, use REVOKE with PUBLIC. For example, the following statement revokes SELECT, INSERT, and UPDATE privileges on the DEPARTMENTS table for all users:

```
REVOKE SELECT, INSERT, UPDATE ON DEPARTMENTS FROM PUBLIC;
```

When this statement is executed, only the table's owner retains full access privileges to DEPARTMENTS.

IMPORTANT   PUBLIC does not revoke privileges for stored procedures. PUBLIC cannot be used to strip privileges from users who were granted them as individual users.

Revoking grant authority

To revoke a user's grant authority for a given privilege, use the following REVOKE syntax:

```
REVOKE GRANT OPTION FOR privilege [, privilege …] ON table
    FROM user;
```

For example, the following statement revokes SELECT grant authority on the DEPARTMENTS table from a user, EMIL:

```
REVOKE GRANT OPTION FOR SELECT ON DEPARTMENTS FROM EMIL;
```

Using views to restrict data access

In addition to using GRANT and REVOKE to control access to database tables, you can use views to restrict data access. A view is usually created as a subset of columns and rows from one or more underlying tables. Because it is only a subset of its underlying tables, a view already provides a measure of access security.

For example, suppose an EMPLOYEES table contains the columns, LAST_NAME, FIRST_NAME, JOB, SALARY, DEPT, and PHONE. This table contains much information that is useful to all employees. It also contains employee information that should remain confidential to almost everyone: SALARY. Rather than allow all employees access to the EMPLOYEES table, a view can be created which allows access to other columns in the EMPLOYEES table, but which excludes SALARY:

```
CREATE VIEW EMPDATA AS
    SELECT LAST_NAME, FIRST_NAME, DEPARTMENT, JOB, PHONE
        FROM EMPLOYEES;
```

Access to the EMPLOYEES table can now be restricted, while SELECT access to the view, EMPDATA, can be granted to everyone.

Note Be careful when creating a view from base tables that contain sensitive information. Depending on the data included in a view, it may be possible for users to recreate or infer the missing data.

14

Character Sets and Collation Orders

CHAR, VARCHAR, and text BLOB columns in InterBase can use many different character sets. A *character set* defines the symbols that can be entered as text in a column, and its also defines the maximum number of bytes of storage necessary to represent each symbol. In some character sets, such as ISO8859_1, each symbol requires only a single byte of storage. In others, such as UNICODE_FSS, each symbol requires from 1 to 3 bytes of storage.

Each character set also has an implicit *collation order* that specifies how its symbols are sorted and ordered. Some character sets also support alternative collation orders. In all cases, choice of character set limits choice of collation orders.

This appendix lists available character sets and their corresponding collation orders.

This appendix also describes how to specify:

- Default character set for an entire database.
- Alternative character set for a particular column in a table.
- Client application character set that the server should use when translating data between itself and the client.
- Collation order for a column.
- Collation order for a value in a comparison operation.

- Collation order in an ORDER BY clause.
- Collation order in a GROUP BY clause.

InterBase character sets and collation orders

The following table lists each character set that can be used in InterBase. For each character set, the minimum and maximum number of bytes used to store each symbol is listed, and all collation orders supported for that character set are also listed. The first collation order for a given character set is that set's implicit collation, the one that is used if no COLLATE clause specifies an alternative order. The implicit collation order cannot be specified in the COLLATE clause.

Character set	Character set ID	Maximum character size	Minimum character size	Collation orders
ASCII	2	1 byte	1 byte	ASCII
BIG_5	56	2 bytes	1 byte	BIG_5
CYRL	50	1 byte	1 byte	CYRL DB_RUS PDOX_CYRL
DOS437	10	1 byte	1 byte	DOS437 DB_DEU437 DB_ESP437 DB_FIN437 DB_FRA437 DB_ITA437 DB_NLD437 DB_SVE437 DB_UK437 DB_US437 PDOX_ASCII PDOX_INTL PDOX_SWEDFIN

TABLE 14.1   Character sets and collation orders

Character set	Character set ID	Maximum character size	Minimum character size	Collation orders
DOS850	11	1 byte	1 byte	DOS850 DB_DEU850 DB_ESP850 DB_FRA850 DB_FRC850 DB_ITA850 DB_NLD850 DB_PTB850 DB_SVE850 DB_UK850 DB_US850
DOS852	45	1 byte	1 byte	DOS852 DB_CSY DB_PLK DB_SLO PDOX_CSY PDOX_HUN PDOX_PLK PDOX_SLO
DOS857	46	1 byte	1 byte	DOS857 DB_TRK
DOS860	13	1 byte	1 byte	DOS860 DB_PTG860
DOS861	47	1 byte	1 byte	DOS861 PDOX_ISL
DOS863	14	1 byte	1 byte	DOS863 DB_FRC863
DOS865	12	1 byte	1 byte	DOS865 DB_DAN865 DB_NOR865 PDOX_NORDAN4
EUCJ_0208	6	2 bytes	1 byte	EUJC_0208
GB_2312	57	2 bytes	1 byte	GB_2312

TABLE 14.1   Character sets and collation orders (*continued*)

Character set	Character set ID	Maximum character size	Minimum character size	Collation orders
ISO8859_1	21	1 byte	1 byte	ISO8859_1 DA_DA DE_DE DU_NL EN_UK EN_US ES_ES FI_FI FR_CA FR_FR IS_IS IT_IT NO_NO PT_PT SV_SV
KSC_5601	44	2 bytes	1 byte	KSC_5601 KSC_DICTIONARY
NEXT	19	1 byte	1 byte	NEXT NXT_DEU NXT_FRA NXT_ITA NXT_US
NONE	0	1 byte	1 byte	NONE
OCTETS	1	1 byte	1 byte	OCTETS
SJIS_0208	5	2 bytes	1 byte	SJIS_0208
UNICODE_FSS	3	3 bytes	1 byte	UNICODE_FSS
WIN1250	51	1 byte	1 byte	WIN1250 PXW_CSY PXW_HUNDC PXW_PLK PXW_SLO
WIN1251	52	1 byte	1 byte	WIN1251 PXW_CYRL

TABLE 14.1   Character sets and collation orders (*continued*)

Character set	Character set ID	Maximum character size	Minimum character size	Collation orders
WIN1252	53	1 byte	1 byte	WIN1252 PXW_INTL PXW_INTL850 PXW_NORDAN4 PXW_SPAN PXW_SWEDFIN
WIN1253	54	1 byte	1 byte	WIN1253 PXW_GREEK
WIN1254	55	1 byte	1 byte	WIN1254 PXW_TURK

TABLE 14.1   Character sets and collation orders (*continued*)

Character set storage requirements

Knowing the storage requirements of a particular character set is important, because in the case of CHAR columns, InterBase restricts the maximum amount of storage in each field in the column to 32,767 bytes (VARCHAR is restricted to 32,765 bytes).

For character sets that require only a single byte of storage, the maximum number of symbols that can be stored in a single field corresponds to the number of bytes. For character sets that require up to three bytes per symbol, the maximum number of symbols that can be safely stored in a field is 1/3 of the maximum number of bytes for the datatype. For example, for a CHAR column defines to use the UNICODE_FSS character set, the maximum number of characters that can be specified is 10,922 (32,767/3):

```
. . .
CHAR(10922) CHARACTER SET UNICODE_FSS,
. . .
```

Paradox and dBASE character sets and collations

Many character sets and their corresponding collations are provided to support Borland Paradox for DOS, Paradox for Windows, dBASE for DOS, and dBASE for Windows.

Character sets for DOS

The following character sets correspond to MS-DOS code pages, and should be used to specify character sets for InterBase databases that are accessed by Paradox for DOS and dBASE for DOS:

Character set	DOS code page
DOS437	437
DOS850	850
DOS852	852
DOS857	857
DOS860	860
DOS861	861
DOS863	863
DOS865	865

TABLE 14.2   Character sets corresponding to DOS code pages

The names of collation orders for these character sets that are specific to Paradox begin "PDOX". For example, the DOS865 character set for DOS code page 865 supports a Paradox collation order for Norwegian and Danish called "PDOX_NORDAN4".

The names of collation orders for these character sets that are specific to dBASE begin "DB". For example, the DOS437 character set for DOS code page 437 supports a dBASE collation order for Spanish called "DB_ESP437".

For more information about DOS code pages, and Paradox and dBASE collation orders, see the appropriate Paradox and dBASE documentation and driver books.

Character sets for Microsoft Windows

There are five character sets that support Windows client applications, such as Paradox for Windows. These character sets are: WIN1250, WIN1251, WIN1252, WIN1253, and WIN1254.

The names of collation orders for these character sets that are specific to Paradox for Windows begin "PXW". For example, the WIN1250 character set supports a Paradox for Windows collation order for Norwegian and Danish called "PXW_NORDAN4".

For more information about Windows character sets and Paradox for Windows collation orders, see the appropriate Paradox for Windows documentation and driver books.

Additional character sets and collations

Support for additional character sets and collation orders is constantly being added to InterBase. To see if additional character sets and collations are available for a newly created database, connect to the database with **isql**, then use the following set of queries to generate a list of available character sets and collations:

```
SELECT RDB$CHARACTER_SET_NAME, RDB$CHARACTER_SET_ID
FROM RDB$CHARACTER_SETS
ORDER BY RDB$CHARACTER_SET_NAME;
SELECT RDB$COLLATION_NAME, RDB$CHARACTER_SET_ID
FROM RDB$COLLATIONS
ORDER BY RDB$COLLATION_NAME;
```

Specifying defaults

This section describes the mechanics of specifying character sets for databases, table columns, and client connections. In addition, it describes how to specify collation orders for columns, comparisions, ORDER BY clauses, and GROUP BY clauses.

Specifying a default character set for a database

A database's default character set designation specifies the character set the server uses to tag CHAR, VARCHAR, and text BLOB columns in the database when no other character set information is provided. When data is stored in such columns without additional character set information, the server uses the tag to determine how to store and transliterate that data. A default character set should always be specified for a database when it is created with CREATE DATABASE.

To specify a default character set, use the DEFAULT CHARACTER SET clause of CREATE DATABASE. For example, the following statement creates a database that uses the ISO8859_1 character set:

```
CREATE DATABASE "europe.gdb" DEFAULT CHARACTER SET ISO8859_1;
```

IMPORTANT   If you do not specify a character set, the character set defaults to NONE. Using character set NONE means that there is no character set assumption for columns; data is stored and retrieved just as you originally entered it. You can load any character set into a column defined with NONE, but you cannot later move that data into another column that has been defined with a different character set. In this case, no transliteration is performed between the source and destination character sets, and errors may occur during assignment.

For the complete syntax of CREATE DATABASE, see the *Language Reference*.

Specifying a character set for a column in a table

Character sets for individual columns in a table can be specified as part of the column's CHAR or VARCHAR datatype definition. When a character set is defined at the column level, it overrides the default character set declared for the database. For example, the following **isql** statements create a database with a default character set of ISO8859_1, then create a table where two column definitions include a different character set specification:

```
CREATE DATABASE "europe.gdb" DEFAULT CHARACTER SET ISO8859_1;
CREATE TABLE RUS_NAME(
    LNAME VARCHAR(30) NOT NULL CHARACTER SET CYRL,
    FNAME VARCHAR(20) NOT NULL CHARACTER SET CYRL,
);
```

For the complete syntax of CREATE TABLE, see the *Language Reference*.

Specifying a character set for a client connection

When a client application, such as **isql**, connects to a database, it may have its own character set requirements. The server providing database access to the client does not know about these requirements unless the client specifies them. The client application specifies its character set requirement using the SET NAMES statement *before* it connects to the database.

SET NAMES specifies the character set the server should use when translating data from the database to the client application. Similarly, when the client sends data to the database, the server translates the data from the client's character set to the database's default character set (or the character set for an individual column if it differs from the database's default character set).

For example, the following **isql** command specifies that **isql** is using the DOS437 character set. The next command connects to the *europe* database created above, in "Specifying a Character Set for a Column in a Table":

```
SET NAMES DOS437;
CONNECT "europe.gdb" USER "JAMES" PASSWORD "U4EEAH";
```

For the complete syntax of SET NAMES, see the *Language Reference*. For the complete syntax of CONNECT, see the *Language Reference*.

Specifying collation order for a column

When a CHAR or VARCHAR column is created for a table, either with CREATE TABLE or ALTER TABLE, the collation order for the column can be specified using the COLLATE clause. COLLATE is especially useful for character sets such as ISO8859_1 or DOS437 that support many different collation orders.

For example, the following **isql** ALTER TABLE statement adds a new column to a table, and specifies both a character set and a collation order:

```
ALTER TABLE "FR_CA_EMP"
    ADD ADDRESS VARCHAR(40) CHARACTER SET ISO8859_1 NOT NULL
        COLLATE FR_CA;
```

For the complete syntax of ALTER TABLE, see the *Language Reference*.

Specifying collation order in a comparison operation

When CHAR or VARCHAR values are compared in a WHERE clause, it can be necessary to specify a collation order for the comparisons if the values being compared use different collation orders.

To specify the collation order to use for a value during a comparison, include a COLLATE clause after the value. For example, in the following WHERE clause fragment from an embedded application, the value to the left of the comparison operator is forced to be compared using a specific collation:

```
WHERE LNAME COLLATE FR_CA = :lname_search;
```
For the complete syntax of the WHERE clause, see the *Language Reference*.

Specifying collation order in an ORDER BY clause

When CHAR or VARCHAR columns are ordered in a SELECT statement, it can be necessary to specify a collation order for the ordering, especially if columns used for ordering use different collation orders.

To specify the collation order to use for ordering a column in the ORDER BY clause, include a COLLATE clause after the column name. For example, in the following ORDER BY clause, the collation order for two columns is specified:

```
. . .
ORDER BY LNAME COLLATE FR_CA, FNAME COLLATE FR_CA;
```

For the complete syntax of the ORDER BY clause, see the *Language Reference*.

Specifying collation order in a GROUP BY clause

When CHAR or VARCHAR columns are grouped in a SELECT statement, it can be necessary to specify a collation order for the grouping, especially if columns used for grouping use different collation orders.

To specify the collation order to use for grouping columns in the GROUP BY clause, include a COLLATE clause after the column name. For example, in the following GROUP BY clause, the collation order for two columns is specified:

```
. . .
GROUP BY LNAME COLLATE FR_CA, FNAME COLLATE FR_CA;
```

For the complete syntax of the GROUP BY clause, see the *Language Reference*.?

InterBase Document Conventions

This appendix describes the InterBase 5 documentation set, the printing conventions used to display information in text and in code examples, and conventions for naming database objects and files in applications.

The InterBase documentation set

The InterBase documentation set is an integrated package designed for all levels of users. It consists of five printed books. Each of these books is also provided in Adobe Acrobat PDF format and is accessible on line through the Help menu. If Adobe Acrobat is not already installed on your system, you can find it on the InterBase distribution CD-ROM or at *http://www.adobe.com/prodindex/acrobat/readstep.html.* Acrobat is available for Windows NT, Windows 95, and most flavors of UNIX. Windows users also have help available through the WinHelp system.

Book	Description
Operations Guide	Provides an introduction to InterBase and an explanation of tools and procedures for performing administrative tasks on databases and database servers. Also includes full reference on InterBase utilities, including **isql**, **gbak**, Server Manager for Windows, and others.
Data Definition Guide	Explains how to create, alter, and delete database objects through ISQL.
Language Reference	Describes SQL and DSQL syntax and usage.
Programmer's Guide	Describes how to write embedded SQL and DSQL database applications in a host language, precompiled through gpre.
API Guide	Explains how to write database applications using the InterBase API.

TABLE A.1   Books in the InterBase 5 documentation set

Printing conventions

The InterBase documentation set uses various typographic conventions to identify objects and syntactic elements.

The following table lists typographic conventions used in text, and provides examples of their use:

Convention	Purpose	Example
UPPERCASE	SQL keywords, SQL functions, and names of all database objects such as tables, columns, indexes, and stored procedures.	The following SELECT statement retrieves data from the CITY column in the CITIES table.
italic	New terms, emphasized words, file names, and host- language variables.	The *isc4.gdb* security database is not accessible without a valid user name and password.
bold	Utility names, user-defined functions, and host-language function names. Function names are always followed by parentheses to distinguish them from utility names.	Use **gbak** to back up and restore a database. Use the **datediff()** function to calculate the number of days between two dates.

TABLE A.2 Text conventions

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 !!;`
italic	Parameters that cannot be broken into smaller units. For example, a table name cannot be subdivided.	`CREATE GENERATOR name;`
<italic>	Parameters in angle brackets that *can* be broken into smaller syntactic units.	`WHILE (<condition>) DO <compound_statement>`
[ ]	Optional syntax: you do not need to include anything that is enclosed in square brackets.	`CREATE [UNIQUE][ASCENDING|DESCENDING]`
{ }	One of the enclosed options *must* be included in actual statement use. If the contents are separated by a pipe symbol (|), you must choose only one.	`{SMALLINT	INTEGER	FLOAT	DOUBLE PRECISION}`
|	You can choose only one of a group whose elements are separated by this pipe symbol. When objects separated by this symbol occur within curly brackets, you *must* choose one; when they are within square brackets you can choose one or none.	`SET {DATABASE	SCHEMA}` `SELECT [DISTINCT	ALL]`
...	The clause in brackets with this symbol can be repeated as many times as necessary.	`(<col> [,<col>…])`

TABLE A.3   Syntax conventions

Index