Product Documentation

InterBase 2017

Update 2

Data Definition Guide

Embarcadero Technologies, Inc. is a leading provider of award-winning tools for application developers. Embarcadero enables developers to design systems right, build them faster and run them better, regardless of their platform or programming language. Ninety of the Fortune 100 and an active community of more than three million users worldwide rely on Embarcadero products to increase productivity, reduce costs and accelerate innovation. The company's flagship tools include: Embarcadero® RAD Studio™, Delphi®, C++Builder®, JBuilder®, and the IoT Award winning InterBase®. Founded in 1993, Embarcadero is headquartered in Austin, with offices located around the world. Embarcadero is online at www.embarcadero.com.

March, 2018

TABLE OF CONTENTS

WORKING WITH GENERATORS

PLANNING SECURITY

ENCRYPTING YOUR DATA

CHARACTER SETS AND COLLATION ORDERS

Data Definition Guide

The Data Definition Guide covers information on designing and building InterBase databases. Topics include:

- Specifying data types
- Working with domains, tables, and indexes
- Working with procedures, triggers, and generators
- Encrypting databases and columns
- Character sets and collation orders

Using the Data Definition Guide

The InterBase Data Definition Guide provides information on the following topics:

- Designing and creating databases
- Working with InterBase structures and objects, including data types, domains, tables, indexes, and views
- Working with tools and utilities such as stored procedures, triggers, Blob filters, and generators
- Planning and implementing database security
- Character sets and collation orders

NOTE

For additional information and support on Embarcadero's products, please refer to the Embarcadero web site at http://www.embarcadero.com.

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 Guide.

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***).

InterBase stores database metadata and other information about it in system tables, which are automatically created when you create a database. All system table names begin with "RDB$". Examples of system tables include RDB$RELATIONS, which has information about each table in the database, and RDB$FIELDS, which has information on the domains in the database.

Writing to these system tables without sufficient knowledge can corrupt a database. Therefore, public users can only select from them. The database owner and SYSDBA user have full read and write privileges and can assign these privileges to others if they wish. For more information about the system tables, see the Language Reference Guide.

IMPORTANT

If you have permission, 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 corrupt your database.

Who Should Use the Data Definition 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 chapter in the InterBase Operations Guide.

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.

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, data types, 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 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
- Collecting and Analyzing Data
- Identifying Entities and Attributes
- Designing Tables
- Determining Unique Attributes (Designing Databases)
- Developing a Set of Rules (Designing Databases)
- Establishing Relationships between Objects
- Planning Security (Designing Databases)

- Naming Objects

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.

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.

List of entities and attributes	
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
CUSTOMER	Customer Number Customer Name Contact Name Phone Number Address

List of entities and attributes	
Entities	**Attributes**
SALES	PO Number Customer Number Sales Rep Order Date Ship Date Order Status

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.

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.

EMPLOYEE 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

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 primary key of the table 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 a data type
- 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 Data Type

Once you have chosen a given attribute as a column in the table, you must choose a data type for the attribute. The data type defines the set of valid data that the column can contain. The data type 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 data types include:

- Character data types.
- Whole number (integer) data types.
- Fixed and floating decimal data types.

- Data types for dates and times.
- A Blob data type to represent data of unspecified length and structure, such as graphics and digitized voice; blobs can be numeric, text, or binary.

For more information about data types supported by InterBase, see Specifying Data Types.

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 data types. You must choose a collation order that is supported for the given character set of the column. 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.ib'
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 data type. The data type 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 Character Sets and Collation Orders.

Specifying Domains

When several tables in the database contain columns with the same definitions and data types, 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 Working with Domains.

Setting Default Values and NULL Status

When you define a column, you have the option of setting a ***DEFAULT*** value. This value is used whenever an ***INSERT*** or ***UPDATE*** on the table does not supply an explicit value for the column. 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; in a Y/N flag column for saving changes, "Y" could be the default. Column-level defaults override defaults set at the domain level. Some examples:

```
stringfld  VARCHAR(10)  DEFAULT  'abc'
integerfld  INTEGER  DEFAULT  1
numfld  NUMERIC(15,4)  DEFAULT  1.5
datefld1  DATE  DEFAULT  '5/5/2005'
datefld2  DATE  DEFAULT  'TODAY'
userfld  VARCHAR(12)  DEFAULT  USER
```

The last two lines show special InterBase features: '***TODAY***' defaults to the current date, and ***USER*** is the user who is performing an insert to the column.

Assign a ***NULL*** default to insert a ***NULL*** into 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 Working with Tables (Data Definition Guide). For more information on the reasons for using foreign keys, see Establishing Relationships between Objects.

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.

PROJECT 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

EMPLOYEE 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

For more information on using ***PRIMARY KEY*** and ***FOREIGN KEY*** constraints, see 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 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.

DEPARTMENT table				
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

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.

DEPARTMENT table - 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

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_LOCATIONS table	
DEPT_NO	**DEPT_LOCATION**
100	Monterey
100	Santa Cruz
600	San Francisco
100	Salinas

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.

PROJECT table					
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

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.

PROJECT table					
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

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.

PROJECT 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

EMPLOYEE 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

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 three levels deep, you should consider reducing your page size. In general, you should use a page size larger than your largest record, although the data compression of InterBase 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 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 2,048 buffers is adequate. If your application includes joins of five 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 small caches, that are un-usable. It is is better to tune your cache on a per-database basis using ***gfix-buffers.***

For more information about cache size, see the Embedded SQL Guide. For more information about using **gfix -buffers**, see the Operations Guide.

Creating a Multifile, Distributed Database

If you feel that your application performance is limited by disk bandwidth, you might consider creating a multifile database and distributing it across several disks. Multifile 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 Planning Security.

Naming Objects

Valid names for InterBase objects must use the 7-bit ASCII character set (character set ID 2) and must have the following characteristics:

- no spaces
- not case sensitive
- not InterBase keywords
- a maximum of 68 bytes long: 67 bytes plus a null terminator

Using delimited identifiers you create metadata names that are case sensitive, can contain spaces, and can be InterBase keywords by placing them double quotes. Such names in double quotes are called delimited identifiers.

TIP

When you use an object name *without* double quotes, InterBase maps all the characters to uppercase. For example, if you create a table with a double-quote delimited name in all uppercase, you can use the name subsequently without double quotes. For example:

```
CREATE TABLE "UPPERCASE_NAME"...
SELECT * FROM UPPERCASE_NAME;
```

Creating Databases

This chapter describes how to:

- Create a database with ***CREATE DATABASE***
- Modify the database with ***ALTER DATABASE***
- Delete a database with ***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 IBConsole, 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.

Database File Naming Conventions

In earlier versions, InterBase database files were given a file extension of gdb by convention. InterBase no longer recommends using gdb as the extension for database files, since on some versions of Windows, any file that has this extension is automatically backed up by the System Restore facility whenever it is touched. On those two platforms, using the gdb extension for database names can result in a significant detriment to performance. Linux and Solaris are not affected. InterBase now recommends using ***ib*** as the extension for

database names. Generally, InterBase fully supports each file naming conventions of a platform, including the use of node and path names.

Creating a Database 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 ***IBConsole***. For more information about command-line ***isql*** and IBConsole, see the Operations Guide.

Using CREATE DATABASE

CREATE DATABASE establishes a new database and populates its system tables, 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.

IMPORTANT

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>]
[WITH ADMIN OPTION];
<secondary_file> = FILE 'filespec' [<fileinfo>] [<secondary_file>]
<fileinfo> = LENGTH [=] INT [PAGE[S]] | STARTING [AT [PAGE]] INT
[<fileinfo>]
```

IMPORTANT

Use single quotes to delimit strings such as file names, user names, and passwords.

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.ib,*** in the current directory.

```
CREATE DATABASE 'employee.ib';
```

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.ib' 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 Multifile Database

A multifile database consists of a primary file and one or more secondary files. You cannot specify what information goes into each secondary file because InterBase handles this automatically. Each secondary file is typically assigned to a different disk than that of the main database. In a multifile database, InterBase writes to the primary file until it has filled the specified number of pages, then proceeds to fill the next specified secondary file.

When you define a secondary file, you can choose to specify its size in database pages (***LENGTH***), or you can specify the initial page number of the following file (***STARTING AT***). InterBase always treats the final file of a multifile database as dynamically sizeable: it grows the last file as needed. Although specifying a ***LENGTH*** for the final file does not return an error, a ***LENGTH*** specification for the last–or only–file of a database is meaningless.

IMPORTANT

Whenever possible, create the database locally. If the database is created locally, secondary file names can include a full file specification, including a host or node names as well as a path and database file name. If you create the database on a remote server, secondary file specifications cannot include a node name, and all secondary files must reside on the same node.

Using LENGTH to Specify a Secondary File

The ***LENGTH*** parameter specifies the number of database pages for the file. The eventual maximum file size is then the number of pages times the page size for the database. (See Specifying Database Page Size.) The following example creates a database with a primary file and three secondary files. The primary file and the first two secondary files are each 10,000 pages long.

```
CREATE DATABASE 'employee.ib'
FILE 'employee2.ib' STARTING AT PAGE 10001 LENGTH 10000 PAGES
FILE 'employee3.ib' LENGTH 10000 PAGES
FILE 'employee4.ib';
```

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.

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*** specifies the beginning page number for a secondary file. The ***PAGE*** keyword is optional. You can specify a combination of length and starting page numbers for secondary files.

If you specify a ***STARTING AT*** parameter that is inconsistent with a ***LENGTH*** parameter for the previous file, the ***LENGTH*** specification takes precedence:

```
CREATE DATABASE 'employee.ib' LENGTH 10000
FILE 'employee2.ib' LENGTH 10000 PAGES
FILE 'employee3.ib' LENGTH 10000 PAGES
FILE 'employee4.ib';
```

The following example produces exactly the same results as the previous one, but uses a mixture of ***LENGTH*** and ***STARTING AT***:

```
CREATE DATABASE 'employee.ib'
FILE 'employee2.ib' STARTING AT 10001 LENGTH 10000 PAGES
FILE 'employee3.ib' LENGTH 10000 PAGES
FILE 'employee4.ib';
```

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. Only the first eight characters of the password are significant.

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.ib' USER 'SALES' PASSWORD 'mycode';
```

Specifying Database Page Size

You can override the default page size of 4,096 bytes for database pages by specifying a different PAGE_SIZE. PAGE_SIZE can be 1024, 2048, 4096, 8192, or 16384. The next statement creates a single-file database with a page size of 2048 bytes:

```
CREATE DATABASE 'employee.ib' 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 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.ib'
DEFAULT CHARACTER SET 'ISO8859_1';
```

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

When there is No Default Character Set

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.

Creating Read-only Databases

By default, databases are in read-write mode at creation time. Such databases must be on a writable file system even if they are used only for ***SELECT***, because InterBase writes information about transaction states to a data structure in the database file.

You have the option of changing a database to read-only mode. Such databases can reside on read-only file systems, such as CD-ROMs. To change the mode of a database to read-only, you can either use gfix (or the equivalent choice in IBConsole), or you can back up the database and restore it in read-only mode. See the Operations Guide for details on how to change the mode of a database using gfix, gbak, or IBConsole.

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 Multifile Database.

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 "Shutting down and restarting databases" in the "Database Configuration and Maintenance" chapter of the Operations Guide.

The syntax for ***ALTER DATABASE*** is:

```
ALTER {DATABASE | SCHEMA}
  {ADD <add_clause> | DROP <drop_clause> | ENCRYPT <key_name> | DECRYPT <key_name> |
SET <set_clause>};

<add_clause> = FILE 'filespec' [fileinfo] [add_clause] | ADMIN OPTION

fileinfo = LENGTH [=] INT [PAGE[S]]
| STARTING [AT [PAGE]] INT [fileinfo]
<drop_clause> = ADMIN OPTION

<key_name> = ENCRYPT <|> DECRYPT

<set_clause> = {FLUSH INTERVAL <number> | NO FLUSH INTERVAL | GROUP COMMIT | NO
GROUP COMMIT |
LINGER INTERVAL <number> | NO LINGER INTERVAL | PAGE CACHE <number> | RECLAIM
INTERVAL <number> | NO RECLAIM INTERVAL | SYSTEM ENCRYPTION PASSWORD
<255-character_string> | NO SYSTEM ENCRYPTION PASSWORD} | PASSWORD DIGEST
'<digest_name>'}
```

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. For more details about the ***ALTER DATABASE*** syntax, see ALTER DATABASE.

NOTE

It is never necessary to specify a length for the last – or only – file, because InterBase always dynamically sizes the last file and will increase the file size as necessary until all the available space is used.

The first example adds two secondary files to the currently connected database by specifying the starting page numbers:

```
ALTER DATABASE
ADD FILE 'employee2.ib' STARTING AT PAGE 10001 LENGTH 10000
ADD FILE 'employee3.ib' STARTING AT PAGE 20001
```

The next example does nearly the same thing as the previous example, but it specifies the secondary file length rather than the starting page number. The difference is that in the previous example, the original file will grow until it reaches 10000 pages. In the second example, InterBase starts the secondary file at the next available page and begins using it immediately.

```
ALTER DATABASE
```

```
ADD FILE 'employee2.ib' LENGTH 10000
ADD FILE 'employee3.ib'
```

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.
- **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 (Creating Databases).
- **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 Creating a Database Shadow

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 Creating a Database Shadow

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. InterBase does not support shadowing to an NFS file system, mapped drive, tape, or other media.

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 (Using CREATE SHADOW).

- 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.

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 multifile 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>];
```

Where:

```
<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.ib, enter:

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

The shadow is associated with the currently connected database, ***employee.ib.*** 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.ib 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.

Shadow Location

On non-NFS systems, which includes all Microsoft Windows machines, the shadow must reside on the same host as the database. You cannot specify a different host name or a mapped drive as the location of the shadow.

On UNIX systems, it is possible to place the shadow on any NFS-mounted directory, but you run the risk of losing the shadow if you experience problems with NFS, so this is not a recommended procedure.

Creating a Multifile Shadow

You can create multifile shadows, similarly to the way you create multifile databases. To create a multifile 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 multifile shadow on a UNIX platform. They create the shadow files on the A, B, and C drives of the IB_bckup node.

The first example creates a shadow set consisting of three files. The primary file, employee.shd, is 10,000 database pages in length and the first secondary file is 20,000 database pages long. The final secondary file, as always, grows as needed.

```
CREATE SHADOW 1 'D:/shadows/employee.shd' LENGTH 10000
FILE 'D:/shadows/employee2.shd' LENGTH 5000
```

```
FILE 'D:/shadows/employee3.shd';
```

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 'D:/shadows/employee.shd' LENGTH 10000
FILE 'D:/shadows/employee2.shd' STARTING AT 10000
FILE 'D:/shadows/employee3.shd' STARTING AT 30000;
```

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.ib
Owner: SYSDBA
Shadow 1: "D:\SHADOWS\EMPLOYEE.SHD" auto length 10000
file D:\SHADOWS\EMPLOYEE2.SHD starting 10000
file D:\SHADOWS\EMPLOYEE3.SHD starting 30000
PAGE_SIZE 1024
Number of DB pages allocated = 462
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:

Auto vs. manual shadows

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	Halts database operation until the problem is fixed; needs intervention of the DBA

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 continues even if the shadow becomes inoperable. If the original shadow was created as a conditional shadow, a new shadow is automatically created. If the shadow was not conditional, you must create a new shadow manually. For more information about conditional shadows, see Conditional Shadows.

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, delete references to it, and create a new shadow.

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.ib 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 the system tables of a database. 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.ib database into the file, newdb.sql:

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

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

Specifying Data Types

This chapter describes the following:

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

About InterBase Data Types

When creating a new column in an InterBase table, the primary attribute that you must define is the data type, which establishes the set of valid data that the column can contain. Only values that can be represented by that data type are allowed. Besides establishing the set of valid data that a column can contain, the data type 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 data type 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 data types:

- ***INTEGER*** and ***SMALLINT***
- ***FLOAT*** and ***DOUBLE PRECISION***
- ***NUMERIC*** and ***DECIMAL***
- ***DATE, TIME,*** and ***TIMESTAMP***
- ***CHARACTER*** and ***VARYING CHARACTER***
- ***BOOLEAN***
- ***BLOB***

InterBase provides the Blob data type to store data that cannot easily be stored in one of the standard SQL data types. A BLOB is used to store data objects of indeterminate and variable size, such as bit-mapped 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 data types. An array is a matrix of individual items composed of any single InterBase data type (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 TIMESTAMP data type is supported that includes information about year, month, day of the month, and time. The TIMESTAMP data type is stored as two long integers, and requires conversion to and from InterBase when entered or manipulated in a host-language program. The DATE data type includes

information on the year, month, and day of the month. The TIME data type includes information about time in hours, minutes, seconds, and tenths, hundredths, and thousandths of seconds.

The following table describes the data types supported by InterBase:

Name	Size	Range/Precision	Description
BLOB	Variable	• None • Blob segment size is limited to 64K • Dynamically sizable data type for storing large data such as graphics, text, and digitized voice • Basic structural unit is the segment • Blob subtype describes Blob contents	
BOOLEAN	16 bits	• TRUE • FALSE • UNKNOWN • Represents truth values TRUE, FALSE, and UNKNOWN • Requires ODS 11 or higher, any dialect	
CHAR(<n>)	<n> characters	• 1 to 32,767 bytes • Character set character size determines the maximum number of characters that can fit in 32K • Fixed length CHAR or text string type • Alternate keyword: CHARACTER	
DATE	32 bits	1 Jan 100 a.d. to 29 Feb 32768 a.d.	Stores a date as a 32-bit longword
DECIMAL (<precision>, <scale>)	Variable (16, 32, or 64 bits)	• <precision> = 1 to 18; specifies at least <precision> digits of precision to store • <scale> = 0 to 18; specifies number of decimal places must be less than or equal to <precision> • Number with a decimal point <scale> digits from the right • Example: DECIMAL(10,3) holds numbers accurately in the following format: *ppppppp.sss*	
DOUBLE PRECISION	64 bits1	2.225 x 10–308 to 1.797 x 10308	IEEE double precision: 15 digits
FLOAT	32 bits	1.175 x 10–38 to 3.402 x 1038	IEEE single precision: 7 digits
INTEGER	32 bits	–2,147,483,648 to 2,147,483,647	Signed long (longword)

Name	Size	Range/Precision	Description
NUMERIC (<precision>, <scale>)	Variable (16, 32, or 64 bits)	• <precision> = 1 to 18; specifies exactly <precision> digits of precision to store • <scale> = 0 to 18; specifies number of decimal places and must be less than or equal to <precision> • Number with a decimal point <scale> digits from the right • Example: NUMERIC(10,3) holds numbers accurately in the following format: *ppppppp.sss*	
SMALLINT	16 bits	–32,768 to 32,767	Signed short (word)
TIME	32 bits	0:00 AM-23:59:59.9999 PM	Unsigned integer of InterBase type ISC_TIME: time of day, in units of 0.0001 seconds since midnight
TIMESTAMP	64 bits	1 Jan 100 a.d. to 29 Feb 32768 a.d.	InterBase type ISC_TIMESTAMP; combines DATE and TIME information
VARCHAR (<n>)	<n> characters	• 1 to 32,765 bytes • Character set character size determines the maximum number of characters that can fit in 32K • Variable length CHAR or text string type • Alternate keywords: CHAR VARYING, CHARACTER VARYING	

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

Where to Specify Data Types

A data type is assigned to a column in the following situations:

- Creating a table using ***CREATE TABLE***.
- Adding a new column to a table or altering a column using ***ALTER TABLE***.
- Creating a global column template using ***CREATE DOMAIN***.
- Modifying a global column template using ***ALTER DOMAIN***.

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

```
<data_type> =
{SMALLINT | INTEGER | FLOAT | DOUBLE PRECISION} [<array_dim>]
| {DATE | TIME | TIMESTAMP} [<array_dim>]
| {DECIMAL | NUMERIC} [(PRECISION [, scale])] [<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 data type using ***CREATE TABLE*** and ***ALTER TABLE***, see Working with Tables. For more information on using ***CREATE DOMAIN*** to define data types, see Working with Domains.

Defining Numeric Data Types

The numeric data types 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*** and ***NUMERIC).***

Integer Data Types

Integers are whole numbers. InterBase supports two integer data types: ***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. Both are exact numerics.

The next two statements create domains with the ***SMALLINT*** and INTEGER data types:

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

You can perform the following operations on the integer data types:

- 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 data types, InterBase automatically converts between ***INTEGER***, ***FLOAT***, and ***CHAR*** data types. For operations that involve comparisons of numeric data with other data types, 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 Data Types

InterBase supports two SQL data types, ***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 data types. These data types are also referred to as exact numerics.

- Precision is the total number or maximum number of digits, both significant and fractional, that can appear in a column of these data types. The allowable range for precision is from 1 to a maximum of 18.
- 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 scale is from zero to precision; in other words, 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*** data types without precision or scale, with precision only, or with both precision and scale.

NUMERIC data type

```
NUMERIC(x,y)
```

In the syntax above, InterBase stores exactly <x> digits. Of that number, exactly <y> digits are to the right of the decimal point. For example,

```
NUMERIC(5,2)
```

declares that a column of this type always holds numbers with exactly five digits, with exactly two digits to the right of the decimal point: ppp.ss.

DECIMAL data type

```
DECIMAL(x,y)
```

In the syntax above, InterBase stores at least <x> digits. Of that number, exactly <y> digits are to the right of the decimal point. For example,

```
DECIMAL(5,2)
```

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

How fixed-decimal Data Types are Stored

When you create a domain or column with a NUMERIC or DECIMAL data type, InterBase determines which data type to use for internal storage based on the precision and scale that you specify and the dialect of the database.

- NUMERIC and DECIMAL data types that are declared without either precision or scale are stored as INTEGER.

- Defined with precision, with or without scale, they are stored as SMALLINT, INTEGER, DOUBLE PRECISION or 64-bit integer. Storage type depends on both the precision and the dialect of the database.

Precision	Dialect 1	Dialect 3
1 to 4	• SMALLINT for NUMERIC data types • INTEGER for DECIMAL data types	SMALLINT
5 to 9	INTEGER	INTEGER
10 to 18	DOUBLE PRECISION	INT64

NUMERIC and DECIMAL data types with precision greater than 10 always produce an error when you create a dialect 2 database. This forces you to examine each instance during a migration. For more about migrating exact numerics, see Migrating Databases with NUMERIC and DECIMAL Data Types. For a broader discussion of migration issues, see the migration appendix in the InterBase Operations Guide.

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

NUMERIC and DECIMAL data type storage based on precision and scale	
Data type specified as...	**Data type stored as...**
NUMERIC	INTEGER
NUMERIC(4)	SMALLINT
NUMERIC(9)	INTEGER
NUMERIC(10)	• DOUBLE PRECISION in dialect 1 • INT64 in dialect 3
NUMERIC(4,2)	SMALLINT
NUMERIC(9,3)	INTEGER
NUMERIC(10,4)	• DOUBLE PRECISION in dialect 1 • INT64 in dialect 3
DECIMAL	INTEGER
DECIMAL(4)	INTEGER
DECIMAL(9)	INTEGER
DECIMAL(10)	• DOUBLE PRECISION in dialect 1 • INT64 in dialect 3
DECIMAL(4,2)	INTEGER
DECIMAL(9,3)	INTEGER
DECIMAL(10,4)	• DOUBLE PRECISION in dialect 1 • INT64 in dialect 3

Specifying NUMERIC and DECIMAL with Scale and Precision

When a NUMERIC or DECIMAL data type 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 64-bit integer 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 scale factor of –1 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.

See the Language Reference Guide information about arithmetic operations using exact and approximate numerics.

Numeric Input and Exponents

Any numeric string in ***DSQL*** or ***isql*** that can be stored as a DECIMAL(18,S) is evaluated exactly, without the loss of precision that might result from intermediate storage as a DOUBLE. A numeric string is recognized by the ***DSQL*** parser as a floating-point value only if it contains an "e" or "E" followed by an exponent, which may be zero. For example, ***DSQL*** recognizes 4.21 as a scaled exact integer, and passes it to the engine in that form. On the other hand, ***DSQL*** recognizes 4.21E0 as a floating-point value.

Specifying Data Types Using Embedded Applications

DSQL applications such as ***isql*** can correct for the scale factor for SMALLINT and INTEGER data types 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 data types with fractional portions when they are stored as SMALLINT or INTEGER types. To avoid this problem, create all NUMERIC and DECIMAL data types that are to be accessed from embedded applications with a precision of 10 or more, which forces them to be stored as 64-bit integer types. 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 64-bit integer without problem.

Considering Migration for NUMERIC and DECIMAL Data Types

NUMERIC and DECIMAL data types that have a precision greater than 9 are stored differently in dialect 1 and dialect 3 databases. As you migrate your databases to dialect 3, consider the following questions about columns defined with NUMERIC and DECIMAL data types:

- Is the precision less than 10? There is no issue. You can migrate without taking any action and there will be no change in the database and no effect on clients.

- For NUMERIC and DECIMAL columns with precision equal to or greater than 10, is DOUBLE PRECISION an appropriate way to store your data?
- In many cases, the answer is "yes." If you want to continue to store your data as DOUBLE PRECISION, change the audiotape of the column to DOUBLE PRECISION either before or after migrating your database to dialect 3. This doesn't change any functionality in dialect 3, but it brings the declaration into line with the storage mode. In a dialect 3 database, newly-created columns of this type are stored as INT64, but migrated columns are still stored as DOUBLE PRECISION. Changing the declaration avoids confusion.
- DOUBLE PRECISION might not be appropriate or desirable for financial applications and others that are sensitive to rounding errors. In this case, you need to take steps to migrate your column so that it is stored as INT64 in dialect 3. As you make this decision, remember that INT64 does not store the same range as DOUBLE PRECISION. Check whether you will lose information in this conversion and whether this is acceptable.

Migrating Databases with NUMERIC and DECIMAL Data Types

Read the "considering migration" section above to decide whether you have columns in a dialect 1 database that would be best stored as 64-bit INT values in a dialect 3 database. If this is the case, follow these steps for each column:

1. Back up your original database. Read the "migration" appendix in the Operations Guide to determine what preparations you need to make before migrating the database. Typically, this includes detecting metadata that uses double quotes around strings. After making necessary preparations as indicated in the migration chapter, back up the database using its current gbak version and restore it using the latest InterBase.
2. Use ***gfix -set_db_SQL_dialect 3*** to change the database to dialect 3.
3. Use the ***ALTER COLUMN*** clause of the ***ALTER DATABASE*** statement to change the name of each affected column to something different from its original name. If column position is going to be an issue with any of your clients, use ***ALTER COLUMN*** to change the positions as well.
4. Create a new column for each one that you are migrating. Use the original column names and if necessary, positions. Declare each one as a DECIMAL or NUMERIC with precision greater than 9.
5. Use ***UPDATE*** to copy the data from each old column to its corresponding new column:

```
UPDATE tablename
SET new_col_name = old_col_name;
```

6. Check that your data has been successfully copied to the new columns and drop the old columns.

Using Exact Numeric Data Types in Arithmetic

In SQL dialect 1, when you divide two integers or two DECIMAL(9,2) values, the quotient has type DOUBLE PRECISION; in other words, it is a floating-point value.

In SQL dialect 3, the quotient of two exact numeric values (SMALLINT, INTEGER, NUMERIC(n,m) or DECIMAL(n.m)) is an exact numeric, with scale factor equal to the sum of the scales of the dividend and divisor. Because a SMALLINT or INTEGER has a scale of 0, the quotient of two INTEGERs is an INTEGER, the quotient of a DECIMAL(9,2) and a DECIMAL(12,3) is a DECIMAL(18,5).

In dialect 1, the fraction 1/3 is 0.33333333333333e0; in dialect 3 it is 0. When an application does something that causes a CHECK condition to be checked, or a stored procedure to be executed, or a trigger to fire, the processing that takes place is based on the dialect under which the check, stored procedure, or trigger was defined, not the dialect in effect when the application causes the check, stored procedure, or trigger to be executed.

For example, suppose that a database is migrated from InterBase 5 and thus has dialect 1; that MYCOL1 and MYCOL2 are SQL INTEGERs; and that a table definition includes the following:

```
CHECK(MYCOL1/MYCOL2>0.5)
```

which was defined using client dialect 1.

Now suppose that a dialect 3 client tries to insert a row in which MYCOL1 is 3 and MYCOL2 is 5; because the CHECK was defined in dialect 1, the quotient will be 0.600000000000e0 and the row will pass the check condition, even though in the current client's dialect 3, the quotient would have been the integer 0 and the row would have failed the check, so the insertion would have been refused.

Floating-point Data Types

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

- 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 data types, InterBase automatically converts between INTEGER, FLOAT, and CHAR data types. For operations that involve comparisons of numeric data with other data types, 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));
```

Date and Time Data Types

InterBase supports DATE, TIME and TIMESTAMP data types.

- DATE stores a date as a 32-bit longword. Valid dates are from January 1, 100 a.d. to February 29, 32768 a.d.
- TIME stores time as a 32-bit longword. Valid times are from 00:00 AM to 23:59.9999 PM.
- TIMESTAMP is stored as two 32-bit longwords and is a combination of DATE and TIME.

The following statement creates TIMESTAMP columns in the SALES table:

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

In the previous example, ***NOW*** returns the system date and time.

Converting to the DATE, TIME, and TIMESTAMP Data Types

Most languages do not support the DATE, TIME and TIMESTAMP data types. Instead, they express them as strings or structures. These data types require conversion to and from InterBase when entered or manipulated in a host-language program. For example, you could convert to the DATE data type in one of the following ways:

- Create a string in a format that InterBase understands (for example, 1-JAN-1999). When you insert the date into a DATE column, InterBase automatically converts the text into the internal DATE format.
- 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 the first bullet 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,TIME and TIMESTAMP data types in C, and how to use the ***CAST()*** function for type conversion using ***SELECT*** statements, refer to "Using ***CAST( )*** to convert dates and times" in "Working with Dates and Times" in the Embedded SQL Guide.

How InterBase Stores Date Values

InterBase stores all date values correctly, including those after the year 2000. InterBase always stores the full year value in a DATE or TIMESTAMP column, never the two-digit abbreviated value. When a client application enters a two-digit year value, InterBase uses the "sliding window" algorithm, described below, to make an inference about the century and stores the full date value including the century. When you retrieve the data, InterBase returns the full year value including the century information. It is up to client applications to display the information with two or four digits.

InterBase uses the following sliding window algorithm to infer a century:

- Compare the two-digit year number entered to the current year modulo 100.
- If the absolute difference is greater than 50, then infer that the century of the number entered is 20, otherwise it is 19.

For a more detailed explanation of the InterBase algorithm and how it is applied, see the "Working with Dates and Times" chapter in the Embedded SQL Guide.

Character Data Types

InterBase supports four character string data types:

1. A fixed-length character data type, 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>) data type, 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>) data type, which is a variable-length national character string up to a maximum of <n> characters.

Specifying a Character Set

When you define the data type 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, either with ***SET NAMES*** in command-line isql, or with IBConsole using the Edit | Options selection to open the SQL options window where you can specify a character set on the Options tab. 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.

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,
. . .);
```

NOTE

Collation order does not apply to BLOB data.

For a list of the international character sets and collation orders that are supported by InterBase, see Character Sets and Collation Orders.

Characters vs. Bytes

InterBase limits a character column definition to 32,767 bytes. VARCHAR columns are restricted to 32,765 bytes. In the case of a single-byte character column, one character is stored in one byte, so you can define 32,767 (or 32,765 for VARCHAR) characters per single-byte column without encountering an error.

For multi-byte character sets, to determine the maximum number of characters allowed in a column definition, divide the internal byte storage limit for the data type by the number of bytes for each character. Thus, two-byte character sets have a character limit of 16,383 per field, and three-byte character sets have a limit of 10,922 characters per field. For VARCHAR columns, the numbers are 16,382 and 10.921 respectively.

The following examples specify a CHAR data type using the UNICODE_FSS character set, which has a maximum size of three bytes for a single character:

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

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 data type 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 Character Sets and Collation Orders.

Fixed-length Character Data

InterBase supports two fixed-length string data types: ***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>)*** data type 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 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 it uses the ISO8859_1 character set 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 data types: ***VARCHAR(<n>)***, or alternately ***CHAR(<n>) VARYING***, and ***NCHAR(<n>)***, or alternately ***NATIONAL CHAR(<n>) VARYING***.

NOTE

InterBase provides SQL syntax that allows you to use BLOBs and VARCHAR data interchangeably. For more information, see Using BLOBs with VARCHAR Data.

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>)*** data type.

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>)*** data type 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>)*** data type:

```
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.

The BOOLEAN Data Type

The ***BOOLEAN*** data type is a 16-bit data type that represents TRUE and FALSE values in a column. When not prohibited by a ***NOT NULL*** constraint, it also supports the ***UNKNOWN*** truth value.

For ESQL and DSQL, the following types are defined in ibase.h:

#define SQL_BOOLEAN 590

SQL Data Type	Macro expression	C data type or typedef	sqlind used?
BOOLEAN	SQL_BOOLEAN	Signed short	NO
BOOLEAN	SQL_BOOELAN +1	Signed short	YES

In ISQL and IBConsole, the output for a BOOLEAN, regardless of values given, is always TRUE, FALSE or UNKNOWN. However, using API function calls, UNKNOWN is treated as ***NULL***, TRUE returns 1, and FALSE returns 0.

NOTE

InterBase looks for Booleans of the form “literal <relop> literal” that evaluate to FALSE and returns a false Boolean inversion node to short-circuit data retrieval.

Examples: The following code illustrates the use of the BOOLEAN data type.

- SQL statements:

```
CREATE TABLE AWARDS_1 (isEligible BOOLEAN, name VARCHAR(20));
INSERT INTO AWARDS_1 VALUES (TRUE, 'Jim Smith');
INSERT INTO AWARDS_1 VALUES (FALSE, 'John Buttler');
SELECT * FROM AWARDS_1;
```

Result:

```
ISELIGIBLE NAME
```

```
========== ============
TRUE       Jim  Smith
FALSE      John  Buttler
```

- SQL statement:

```
SELECT * FROM AWARDS_1 WHERE isEligible = TRUE;
```

Result:

```
ISELIGIBLE  NAME
========== =========
TRUE       Jim  Smith
```

SQL statement:

```
SELECT * FROM AWARDS_1 WHERE isEligible;
```

Result:

```
ISELIGIBLE  NAME
========== =========
TRUE       Jim  Smith
```

- SQL statement:

```
SELECT * FROM AWARDS_1 WHERE NOT isEligible;
```

Result:

```
ISELIGIBLE  NAME
========== ============
FALSE      John  Buttler
```

Defining BLOB Data Types

InterBase supports a dynamically sizable data type called a BLOB to store data that cannot easily be stored in one of the standard SQL data types. A Blob is used to store very large data objects of indeterminate and variable size, such as bit-mapped 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 Embedded SQL Guide.

The BLOB data type 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 data type to avoid storing pointers to non-database files.

BLOB Columns

You define BLOB columns in database tables just as you do 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 BLOBsegments, which are units of BLOB data that are read and written in chunks. InterBase writes data to a BLOB one segment at a time. Similarly, it reads a BLOB one 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:

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 you define a BLOB column, you have the option of specifying a subtype. 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 subtypes	
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)
5	Encoded description of the current metadata of a table
6	Description of multi-database transaction that finished irregularly

NOTE

TEXT is a keyword and can be used in a BLOB column declaration instead of the subtype number.

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 data type in a particular application, then the application must ensure that only data of that data type 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 bit-mapped image format to another.

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 Embedded SQL Guide.

Using BLOBs with VARCHAR Data

All BLOB sub-types can be used interchangeably with VARCHAR data. However, with BLOB SUB_TYPE 1, the BLOB is considered to have a character type, essentially making the BLOB a CLOB data type. For BLOB columns of SUB_TYPE 1, the server converts character data to the column's character type before inserting, updating or comparing the data. For all other sub-types, the BLOB data type accepts character input and treats it just as it would all other binary data. Hence, the BLOB data type treats all textual data as an array of bytes. Text data used in ISQL has a character set associated with it. This will most likely be the character encoding of the machine running ISQL (or any other client).

The server does not perform any character set conversion in these cases. Again, the server treats the data as an array of bytes. To convert or store the textual data to a particular encoding (other than the system encoding), cast the character data to the required character set.

About Text BLOB Syntax

The general syntax for the SQL ***SELECT*** statement with a BLOB data type is:

```
SELECT CAST (<blob-column-name> AS CHAR[<n>]) FROM <table-name>;
```

To make text blobs interchangeable with VARCHAR data, you can use the following SQL syntax:

```
INSERT INTO <table-name> VALUES (<text values>, ....);
UPDATE <table_name> SET <BLOB COLUMN name> = <text value>;
```

And:

```
SELECT CAST (<BLOB COLUMN name> AS CHAR[128]) FROM TABLE;
```

```
SELECT * FROM <TABLE name> WHERE CAST (<BLOB column> AS VARCHAR[10]) =
"SMISTRY";
```

In addition, store procedures which accept a BLOB can now accept a text value as a parameter and implicitly be converted to a text blob. For example:

```
CREATE PROCEDURE MYTEST (AINT INTEGER, INBLOB BLOB)
AS
DECLARE variable var_blob BLOB;
BEGIN
INSERT
var_blob
```

This procedure can now be called using the following syntax:

```
EXECUTE PROCEDURE mytest (1, 'hello world');
```

You can use the ***SELECT CAST***, ***UPDATE***, and ***INSERT INTO*** statements with the InterBase Client APIs. In such cases, InterBase returns the values as C structures. Specifically, the returned XSQLVARS would be of the type SQLVARYING, with the length of the text followed by the text data.

The following example demonstrates the use of the new SQL syntax for text BLOBs.

Example:

```
/* Same syntax to create a table... */
/* Note all sub-types are supported; SUB_TYPE 1 forces conversion */
/* to the column's character data type. */
CREATE TABLE BLOB_TEST (B_ID INT, BLOB_CL BLOB SUB_TYPE 1);
COMMIT;

/* New functionality for the INSERT statement...

*/
INSERT INTO BLOB_TEST VALUES (1, 'Fellowship of the Ring');
INSERT INTO BLOB_TEST VALUES (2, 'The Two Towers');
INSERT INTO BLOB_TEST VALUES (3, 'Return of the Jedi');

/* New syntax for UPDATE... */
UPDATE BLOB_TEST SET BLOB_CL='Return of the King' WHERE B_ID=3;
COMMIT;

/* New syntax for SELECT. The BLOB will be returned as a TEXT string. */
SELECT B_ID, CAST (BLOB_CL AS VARCHAR(25)) FROM BLOB_TEST;
```

This table illustrates the result of these statements in ISQL:

Text BLOB Example Result	
B_ID	**BLOB_CL**
1	Fellowship of the Ring
2	The Two Towers
3	Return of the King

Defining Arrays

InterBase allows you to create arrays of data types. 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 data type.
- 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 data type except BLOB, and cannot be an array of arrays. All of the elements of a particular array are of the same data type.

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 data type specification of a column.

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 Embedded SQL 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 four dimensions respectively:

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

```
INT_ARR4  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_ARR4 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 the set of subscripts of each dimension 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 Data Types

Normally, you must use compatible data types to perform arithmetic operations, or to compare data in search conditions. If you need to perform operations on mixed data types, or if your programming language uses a data type that is not supported by InterBase, then data type conversions must be performed before the database operation can proceed. InterBase either automatically (dialect 1) converts the data to an equivalent data type (an implicit type conversion), or you can use the ***CAST()*** function (dialect 3) in search conditions to explicitly translate one data type into another for comparison purposes.

You cannot convert array data to any other data type, nor can any data type be converted to an array. Individual elements of an array, however, behave like members of the array's base data type. To see how BLOBs can be converted to VARCHARS, see Using BLOBs with VARCHAR Data.

Implicit Type Conversions

InterBase supports several types of implicit type conversion. For example, comparing a DATE or TIMESTAMP column to '6/7/2000' causes the string literal '6/7/2000' to be converted implicitly to a DATE entity. An expression mixing integers with scaled numeric types or float types implicitly converts the integer to a like type.

However, InterBase dialect 3 differs from dialect 1 in this respect: in dialect 3, implicit string-to-integer conversion is not supported. For example, in the following operation:

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

- InterBase dialect 1 automatically converts the character "1" to an INTEGER for the addition.
- InterBase dialect 3 returns an error.

In dialect 3, an explicit type conversion is needed:

```
3 + CAST('1' AS INT)
```

The next example returns an error in either dialect, 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 data type to another inside a ***SELECT*** statement. Typically, ***CAST()*** is used in the ***WHERE*** clause to compare different data types. The syntax is:

```
CAST (value
| NULL AS data_type)
```

Use ***CAST()*** to translate the following data types:

- ***DATE***, ***TIME***, or ***TIMESTAMP*** data type into a ***CHARACTER*** data type.

- CHARACTER data type into a ***DATE***, ***TIME***, or ***TIMESTAMP*** data type.
- ***TIMESTAMP*** data type into a ***TIME*** or ***DATE*** data type.
- ***TIME*** or ***DATE*** data type into a ***TIMESTAMP*** data type.
- ***BOOLEAN*** into a ***CHAR*** or ***VARCHAR***.
- ***BLOB*** subtype 1 into a ***VARCHAR***.

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

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

In the next example, ***CAST()*** is used to translate a ***DATE*** data type into a CHAR data type:

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

You can use ***CAST()*** to compare columns with different data types in the same table, or across tables. For more information, refer to "Working with Dates and Times" in the Embedded SQL Guide.

Converting a numeric data type to a character type requires a minimum length for the character type, as listed below.

Minimum character lengths for numeric conversions	
Data type	**Minimum length for converted character type**
Decimal	20
Double	22
Float	13
Integer	11
Numeric	22
Smallint	6

Working with Domains

This chapter covers the following topics:

- Creating Domains (Data Definition Guide)
- Altering Domains
- Dropping 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 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.

When you create a domain in the database, you must specify a unique name for the domain and specify the data type. Optionally, you provide default values and ***NULL*** status, ***CHECK*** constraints, and a collation order.

The syntax for ***CREATE DOMAIN*** is:

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

Specifying the Domain Data Type

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

Syntax

```
<data_type> =
{SMALLINT|INTEGER|FLOAT|DOUBLE PRECISION} [<array_dim>]
  | {DATE|TIME|TIMESTAMP} [<array_dim>]
```

```
   | {DECIMAL | NUMERIC} [(PRECISION [, scale])] [<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])]
   | BOOLEAN

<array_dim> = [x:y [, x1:y1 ...]]
```

NOTE

The <data_type> is the SQL data type for any column based on a domain. You cannot override the domain data type with a local column definition.

The general categories of SQL data types include:

- Character data types.
- Integer data types.
- Decimal data types, both fixed and floating.
- A DATE data type to represent the date, a TIME data type to represent the time, and a TIMESTAMP data type to represent both date and time.
- A BLOB data type to represent unstructured binary data, such as graphics and digitized voice.
- Arrays of data types (except for BLOB data).

See About InterBase Data Types for a complete list and description of data types that InterBase supports.

For more information about data types, see Specifying Data Types.

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

```
CREATE DOMAIN DEPTARRAY AS CHAR(67) [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 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.

IMPORTANT

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 CH***E***CK 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');
```

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 data types. 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 data type 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 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;
```

Altering Domains

ALTER DOMAIN changes any aspect of an existing domain except its **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 the **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
- Modify the domain name and data type
- Modify the data type of a column

The syntax for **ALTER DOMAIN** is:

```
ALTER DOMAIN { name | old_name TO new_name } {
[SET DEFAULT {literal | NULL
| USER}]
```

```
| [DROP DEFAULT]
| [ADD [CONSTRAINT] CHECK (<dom_search_condition>)]
| [DROP CONSTRAINT]
| new_col_name
| TYPE data_type
};
```

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

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

The following statement changes the name of the CUSTNO domain to CUSTNUM:

```
ALTER DOMAIN CUSTNO TO CUSTNUM;
```

The following statement changes the data type of the CUSTNUM domain to CHAR(20):

```
ALTER DOMAIN CUSTNUM TYPE CHAR(20);
```

The ***TYPE*** clause of ***ALTER DOMAIN*** does not allow you to make data type conversions that could lead to data loss. For example, it does not allow you to change the number of characters in a column to be less than the largest value in the column.

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;
```

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:

- Design, normalize, create, and connect to a database
- Determine what tables, columns, and column definitions to create
- Create the domain definitions in the database
- Declare the table if an embedded SQL application both creates a table and populates the table with data in the same program

For information on how to create, drop, and modify domains, see Working with Domains. 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.

For the complete syntax, see ***CREATE TABLE*** in the "SQL Statement and Function Reference" chapter of the Language Reference. This SQL reference is also available in HTML format.

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 Planning Security.

Metadata name length Database object names, including table, column, and domain names can be up to 68 types in length: 67 bytes plus a ***NULL*** terminator.

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

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 {data_type
| 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:
- A SQL data type (<data_type>).
- 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 data type. If you do not specify a character set, the column assumes the database character set as a default.

Specifying the Data Type

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

Syntax

```
<DATA type> =
{SMALLINT|INTEGER|FLOAT|DOUBLE PRECISION} [<array_dim>]
| {DATE|TIME|TIMESTAMP} [<array_dim>]
| {DECIMAL | NUMERIC} [(PRECISION [, scale])] [<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])]
| BOOLEAN
<array_dim> = [x:y [, x1:y1 ...]]
```

NOTE

<subtype_name> can be a TEXT value.

NOTE

The outermost brackets must be included when declaring arrays.

Supported data types

The general categories of data types that are supported include:

- Character data types.
- Integer data types.
- Decimal data types, both fixed and floating.
- A DATE data type to represent the date, a TIME data type to represent the time, and a TIMESTAMP data type to represent both the date and time.
- A BOOLEAN data type.
- A BLOB data type to represent unstructured binary data, such as graphics and digitized voice.
- Arrays of data types (except for BLOB data).

See About InterBase Data Types for a complete list and description of data types that InterBase supports.

Casting Data Types

If your application programming language does not support a particular data type, you can let InterBase automatically convert the data to an equivalent data type (an implicit type conversion), or you can use the ***CAST()*** function in search conditions to explicitly translate one data type into another for comparison purposes. For more information about specifying data types and using the ***CAST()*** function, see Specifying Data Types.

Defining a Character Set

The data type 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 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 data types. You must choose a collation order that is supported for the given character set of the column. 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 data type 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 default character set of the database. 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 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 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 data type 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 data type, InterBase calculates an appropriate one. <expr> is any arithmetic expression that is valid for the data types 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 Columns

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 on a Table

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, InterBase returns 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.

The EMPLOYEEtable				
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

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.

Using the FOREIGN KEY to Enforce Referential Integrity

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.

The PROJECT 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

The EMPLOYEE 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

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 referentialintegrity 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
RESTRICT ***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***

- 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.

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 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 primary key of the EMPLOYEE table, must be created before the DEPARTMENT table can define EMP_NO as its foreign key.

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 RDB$RELATION_CONSTRAINTS system table.

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 [(other_col [, other_col ...])]
[ON DELETE {RESTRICT|NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
[ON UPDATE {RESTRICT|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));
```

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. In the syntax for ***CREATE TABLE***, the <filespec> that accompanies the ***EXTERNAL*** keyword is the fully qualified file specification for the external data file. You can modify the external file outside of InterBase, since InterBase accesses it only when needed.

NOTE

The 2GB external file size limit has been removed from InterBase XE onward.

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.

IMPORTANT

For security reasons, it is extremely important that you not place files with sensitive content in the same directory with external tables.

Restrictions

The following restrictions apply to using the EXTERNAL FILE option:

- The default location for external files is ***<InterBase_home>/ext.*** InterBase can always find external files that you place here. If you want to place them elsewhere, you must specify the location in the ibconfig configuration file using the EXTERNAL_FILE_DIRECTORY entry.

NOTE

If you are migrating from InterBase 6.x or older to InterBase 7.x or newer, and your database includes external table files, you must either move these files to ***<InterBase_home>/ext*** or note their locations in ***ibconfig*** using the EXTERNAL_FILE_DIRECTORY entry.

- 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 an EOL symbol of a particular system (usually one or two bytes). For most versions of UNIX, it is 1 byte. For Microsoft Windows, 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 two-byte unsigned short indicates the number of bytes in the actual string, and the string immediately follows. Because it is not readily portable, using ***VARCHAR*** data in an external file is not recommended.

- You can perform only ***INSERT*** and ***SELECT*** operations on an external table. You cannot perform ***UPDATE*** or ***DELETE*** operations on it; 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

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 one byte. If the example file was on a Windows platform, you would need two characters for NEW_LINE.

```
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(s). 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 such as Paradox for Windows or dBASE for Windows. If you create the file 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. The number of characters in the EOL is platform-specific. You need to know how many characters are contained in the EOL of your platform (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: When exporting data to or from an external file, the file must already exist before you begin the operation. Also, you must specify a directory path whenever you reference the external file.
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
```

NOTE

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.

NOTE

When exporting data to or from an external file, the file must already exist before you begin the operation. Also, you must specify a directory path whenever you reference the external file.

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***. Be sure to specify the file directory.

```
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.

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.
- Modify the column name, data type, and position.

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.

NOTE

Any one table (and its triggers) can be altered at most 255 times before you must back up and restore the database.

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 five-step process:

1. Add a temporary column to the table whose definition mirrors the current column to be changed (the "old" column).
2. Copy the data from the old column to the temporary column.
3. Modify the old column.
4. Copy the data from the temporary column to the old column.
5. Drop the temporary column.

For example, suppose the EMPLOYEE table contains a column, OFFICE_NO, defined to hold a data type of CHAR (3), and suppose that the size of the column needs to be increased by one.

An example:

The following example describes each step and provides sample code:

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

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

Move existing data from OFFICE_NO to TEMP_NO to preserve it:

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

Modify OFFICE_NO, specifying the data type and new size:

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

Move the data from TEMP_NO to OFFICE_NO:

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

Finally, drop the TEMP_NO column:

```
ALTER TABLE DROP TEMP_NO;
```

NOTE

This is the safest, most conservative method for altering a column, following the rule that you should always save existing data before modifying metadata. But for experienced InterBase users, there is a faster, one-step process. You can alter the column without first copying the data, for example: ***ALTER TABLE EMPLOYEE ALTER COLUMN OFFICE_NO TYPECHAR(4)*** which achieves the same end as the five-step process example.

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 sub-queries 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.
- Modify column names, data types, and position

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 {<data_type>
| [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 {RESTRICT|NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
[ON UPDATE {RESTRICT|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.

Modifying Columns in a Table

The syntax for modifying a column with ***ALTER TABLE*** is:

```
ALTER TABLE table ALTER [COLUMN]simple_column_name alter_rel_field
alter_rel_field = new_col_name | new_col_type | new_col_pos
```

```
new_col_name = TO simple_column_name
new_col_type = TYPE data_type_or_domain
new_col_pos = POSITION integer
```

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

For example, the following statement moves a column, EMP_NO, from the third position to the second position in the EMPLOYEE table:

```
ALTER TABLE EMPLOYEE ALTER EMP_NO POSITION 2;
```

You could also change the name of the EMP_NO column to EMP_NUM as in the following example:

```
ALTER TABLE EMPLOYEE ALTER EMP_NO TO EMP_NUM;
```

The next example shows how to change the data type of the EMP_NUM column to CHAR(20):

```
ALTER TABLE EMPLOYEE ALTER EMP_NUM TYPE CHAR(20);
```

Conversions from non-character to character data are allowed with the following restrictions:

- Blob and array types are not convertible.
- Field types (character or numeric) cannot be shortened.
- The new field definition must be able to hold the existing data (for example, the new field has too few ***CHAR*** values or the data type conversion is not supported) or an error is returned.

NOTE

Conversions from character data to non-character data are not allowed.

IMPORTANT

Any changes to the field definitions may require the indexes to be rebuilt.

The table below graphs all valid conversions; if the conversion is valid (converting from the item on the side column to the item in the top row) it is marked with an X.

Valid data type conversions using ALTER COLUMN and ALTER DOMAIN													
Convert:	**Blob**	**Boolean**	**Char**	**Date**	**Dec.**	**Dble**	**Flo**	**Int.**	**Num.**	**Tstmp**	**Time**	**Smlint**	**Var.**
Blob													
Boolean		X	X										X
Char		X	X										X
Date			X	X						X			
Decimal			X		X				X				X
Double			X			X	X						X
Float			X			X	X						X
Integer			X		X	X		X	X				X
Numeric			X						X				X
Timestamp			X							X	X		
Time			X							X	X		
Smallint			X		X	X	X	X	X			X	X
Varchar		X	X										X

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 data type 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>***.
- Modify column names, data types, and positions

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 the data, metadata, and indexes of a table 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 the ***CHECK*** constraint of another table.

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;
```

Global Temporary Tables

Use global temporary tables to allow an application to pass intermediate result sets from one section of an application to another section of the same application.

Creating a Global Temporary Table

A global temporary table is declared to a database schema via the normal ***CREATE TABLE*** statement with the following syntax:

```
CREATE GLOBAL TEMPORARY TABLE <table>
(<<col_def>> [, <<col_def>> | <<tconstraint>> ...])
[ON COMMIT {PRESERVE | DELETE} ROWS];
```

The first argument that you supply ***CREATE GLOBAL TEMPORARY TABLE*** is the temporary 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.

The ***ON COMMIT*** clause describes whether the rows of the temporary table are deleted on each transaction commit (***ON COMMIT DELETE***) or are left in place (***ON COMMIT PRESERVE***) to be used by other transactions in the same database attachment. If the ***ON COMMIT*** is not specified then the default behavior is to ***DELETE ROWS*** on transaction commit.

There is a change in behavior in the ***GLOBAL TEMPORARY TABLE*** Support with the InterBase XE3 Update 2 release. When an SQL script is executed ISQL reported a "deadlock" if ***EXIT*** is called without ***COMMIT/ ROLLBACK*** on a global temporary table. To resolve this issue, the ***GLOBAL TEMPORARY TABLES*** function has been redesigned which changes the behavior and corrects the deadlock error.
It is no longer possible for transactions emanating from the same connection to see each other's rows in a transaction-specific (***ON COMMIT DELETE***) temporary table. To do that, you must use a session-specific (***ON COMMIT PRESERVE***) temporary table that makes all rows visible to transactions starting in the same session. This is still not the same in that the rows will persist until the connection is finished.

An Global temporary table is dropped from a database schema using the normal ***DROP TABLE*** statement.

Altering a Global Temporary Table

A temporary table can be altered in the same way as a permanent base table although there is no official support to toggle the behavior of the ***ON COMMIT*** clause. The specification offers an ***ALTER TABLE*** syntax to toggle that behavior.

```
ALTER TABLE <table> ON COMMIT {PRESERVE | DELETE} ROWS
{RESTRICT CASCADE}
```

RESTRICT will report an error if there are dependencies by other temporary tables on the current table scope. ***CASCADE*** will automatically propagate this table scope change to other temporary tables to maintain compliance. The default action is ***RESTRICT***.

For example, assume that TT1 is a temporary table with ***ON COMMIT PRESERVE*** and has a foreign reference to temporary table TT2 which is also ***ON COMMIT PRESERVE***. If an attempt is made to modify TT2 to ***ON COMMIT DELETE***, an error is raised because an ***ON COMMIT PRESERVE*** table is not allowed by the SQL standard to have a referential constraint on an ***ON COMMIT DELETE*** table. ***RESTRICT*** returns this error while ***CASCADE*** would also alter TT1 to have ***ON COMMIT DELETE***. Thus, ***CASCADE*** implements transitive closure when ***ON COMMIT*** behavior is modified.

NOTE

This specification of ***ALTER TABLE*** extension does not allow a table to be toggled between temporary and persistent.

Requirements and Constraints

A transaction which has been specified as ***READ ONLY*** is allowed to update temporary tables.

Granting privileges on a temporary table to an entity must specify all privileges.

There are some semantic restrictions between how permanent tables and temporary tables are allowed to interact. For the most part, general constraints and referential integrity constraints require that for a given table on which those constraints are defined, the tables those constraints reference must have the same table scope as that of the source table. Permanent tables can only have referential and check constraints to other permanent tables, and temporary tables can only have constraints against other temporary tables.

Another example is the check constraint with a subquery component; the table on which the check constraint is defined must match in table scope the table referenced in the subquery.

Domains are not allowed to reference temporary tables in check constraints.

gbak backs up a temporary tables' metadata only, not its data. isql adds an ***ON COMMIT*** descriptive line in the ***SHOW TABLE*** command. isql extract adds ***GLOBAL TEMPORARY*** and ***ON COMMIT*** clauses when extracting temporary tables. ***GRANT*** privileges are always extracted as ***ALL PRIVILEGES***.

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. The engine will use an index to look up a subset of columns, as long as that subset of columns forms a prefix of a multi-column index definition.

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, ***SHOWINDEX*** *<**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.

NOTE

When working with encrypted columns, the ***MIN***, ***MAX***, ***BETWEEN*** and ***ORDER BY*** operations cannot use an index based on those fields due to the nature of the index key that is formed from the encrypted field value. So while the index is not useful for the above operations, it is still userful for equality matches and ***JOIN*** operations.

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 SET STATISTICS: Recomputing Index Selectivity.

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 ***DROPINDEX*** and ***CREATE INDEX***.
- Back up and restore the database with ***gbak.***

ALTER INDEX: Deactivating an 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 set ***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***.

SET STATISTICS: Recomputing Index Selectivity

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***.

Dropping a User-defined 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.

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 to Views

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 Update-able.

A view can be created from:

- **A vertical subset of columns from a single table** For example, the table, JOB, in the employee.ib 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 the ***SELECT*** statement, 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 columns 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 otherwise update-able 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. For an explanation of views that can be updated, see Types of Views: Read-only and Update-able.

NOTE

The ***SELECT*** statement used to create a view 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 Update-able

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 update-able. 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 update-able 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 update-able if all of the following conditions are met:

- It is a subset of a single table or another update-able view.
- All base table columns excluded from the view definition allow ***NULL*** values.
- The ***SELECT*** statement of the view does not contain sub-queries, 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 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 update-able view, the creator needs ***ALL*** privileges to the underlying tables.

For more information on SQL privileges, see Planning Security.

Examples of Views

The following statement creates an update-able 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 update-able.
- 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.

Using WITH CHECK OPTION

WITH CHECK OPTION specifies rules for modifying data through views. This option can be included only if the views are update-able. 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 un-referenced 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 ***SELECT*** portion of the ***CREATE VIEW*** 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 Change Views

The Change Views™ feature uses InterBase multigenerational architecture to capture changes to data. This feature allows you to quickly answer the question, "What data has changed since I last viewed it?"

Previously it involved triggers, logging, and/or transaction write-ahead log scraping. This was time-consuming for the developer and affected the database performance for a certain transaction load or change volume. Now with Change Views, there is no performance overhead on existing transactions because it maintains a consistent view of changed data observable by other transactions.

The Change Views mechanism is not dependent on its own underlying data, but is based on data already stored for existing base tables or views derived from base tables. This implicit view mechanism is temporal based and returns data that have changed since the prior transaction in which the implicit view was observed.

Change Views can be subscribed to (Create Subscriptions to Implement Change Views) in order to view data that has changed across database connections. The effect is a long-lived transaction spanning multiple database connections.

- Specifically, the subscription tracks all row inserts, updates, and deletes to one or more tables at a column-level granularity over a disconnected, extended period of time.
- The InterBase SQL query language is modified to search on columns where data has changed since the prior observation.
- These data changes are tracked at a column granularity.

Getting Started with Change Views

ODS Platform Updates

- Change Views requires underlying modification of InterBase ODS, so existing databases must be backed up and restored to the ODS version that supports this feature. Due to the new ODS format, the restored database will not be attachable by older InterBase editions.

Migration Issues and Dependencies

- A subscription defined with ***FOR ROW*** (***..., DELETE***) returns a row with column values that existed before the row was deleted. Exercise caution in reporting applications on existing data to not have such a subscription bound to the transaction. Otherwise, the resultset will include data that no longer exists.

Requirements and Constraints

Requirements

- It is required to use the "odd-numbered" SQL DATA TYPES (SQLVAR variable, sqltype) when working with change views. This allows the SQL indicator variable in the SQLVAR to receive status on the returned column data.

- It is required to use transaction SNAPSHOT isolation level when selecting from a change view. Any isolation level can be used when modifying data in a change view.
- InterBase tables can have up to 256 record format versions. When a subscription references a table, this results in a new record format for the table. When issuing a large number of ***CREATE SUBSCRIPTION*** statements, disable autocommit DDL operation so that a single record format can be constructed from multiple table references from many subscriptions.

Constraints

- ***TRANSACTION ID AS PROXY FOR SUBSCRIPTION TIMESTAMP***
- The feature relies heavily on transaction ID as a proxy for last observed timestamp of a changed data view and assuring that changed data results are not resent in duplicate. Currently, database backup does not save transaction IDs of committed data and restores a database with next transaction ID reset to zero. A 32-bit transaction ID space can be and has been exhausted at customer sites, which necessitates a database backup and restore.
- This project assumes conversion to 64-bit (more likely 48-bit) transaction IDs with the assumption they could not be exhausted in a reasonable time frame. For example, a database with a 4KB page size could run continuously in excess of 10,000 tps for 100 years with a 48-bit transaction ID. In addition to supporting changed data views, it also ensures that an InterBase database never has to be shutdown for backup and restore because transaction IDs have been exhausted.

Backup/Restore Considerations

- A logical backup/restore will backup and restore all subscription definitions. But, data that tracks subscribers using those subscriptions will not be backed up or restored. Subscribers need to initiate/activate their subscriptions on new (or restored) databases.

Database Restore from a Backup

- If the database is restored from a logical backup file or physical dump file, subscribers may retrieve subscription changes that they have already received. There is metadata RDB$SUBSCRIPTIONS.RDB $CHECK_OUT_TIMESTAMP that a subscriber could save to help with the issue of duplicate change processing.

Creating Subscriptions to Change Views

To establish interest in observing changed data on a set of tables beyond the natural boundary of a database connection, a subscription must be created on a list of tables (base tables or views):

Syntax for CREATE SUBSCRIPTION

```
CREATE SUBSCRIPTION <subscription_name> ON<
        <table_name>[(column_name_comma_list)][FOR ROW (CHANGE | {INSERT, UPDATE,
DELETE})]
        [, <table_name>[(column_name_comma_list)][FOR ROW (CHANGE | {INSERT, UPDATE,
DELETE})] ...]
```

```
    [DESCRIPTION  user-description];
```

- The **FOR ROW** clause describes what types of row modifications will cause column-level changes to be tracked for the subscription.
- If the **FOR ROW** clause is omitted, then the behavior defaults to **FOR ROW**(**CHANGE**). The **CHANGE** option tracks **INSERT** and **UPDATE** data changing operations and returns a row as soon as any tracked columns changes value.
- When **INSERT**, **UPDATE**, and **DELETE** are specified, then change status on every tracked column is determined. This is called "deep record introspection" and the **CHANGE** option is called "shallow record introspection". It is expected that Change Views performance is faster with the **CHANGE** option, but that the **INSERT**, **UPDATE**, and **DELETE** combinations are more complete. **DELETE** is not tracked by default, unless explicitly specified.
- Deep record introspection is required when it is necessary to account for every tracked column and to know exactly which operation(s) caused a column's value change.
- Shallow record introspection is for the use case where it is only necessary to know that one or more columns changed and not how every tracked column value changed or if they even changed value at all; it indicates that the row changed somehow with respect to the subscription.
- If a table alone is specified then all columns of the table are tracked.
- If only a subset of columns is desired to be tracked, then an optional list of columns can be specified by the subscription. For example,

Sample for CREATE SUBSCRIPTION

```
CREATE SUBSCRIPTION sub_employee_changes ON EMPLOYEE (EMP_NO, DEPT_NO, SALARY)
DESCRIPTION  'Subscribe to changes in EMPLOYEE table';

CREATE SUBSCRIPTION sub_customer_deletes ON CUSTOMER FOR ROW (DELETE) DESCRIPTION
'Subscribe to deletes in CUSTOMER table';

CREATE SUBSCRIPTION sub_various_changes
ON EMPLOYEE FOR ROW (INSERT, UPDATE, DELETE),
    CUSTOMER FOR ROW (INSERT, UPDATE, DELETE),
    SALES FOR ROW (UPDATE),
    DEPARTMENT (LOCATION) FOR ROW (UPDATE)
DESCRIPTION 'Subscribe to various changes on multiple tables';
```

An optional list of columns is specified for the EMPLOYEE table so that only changes on those columns are tracked. Since no **FOR ROW** clause is specified for EMPLOYEE table, the default for **FOR ROW** is the **CHANGE** option, which causes all insert, update, and delete changes are tracked by the subscription and that any change will return the row. The CUSTOMER table clause specifies that only row deletions are tracked.

DROP SUBSCRIPTION

NOTE

This new function was introduced with InterBase XE7 Update 1.

To eliminate interest in observing a set of change views, a subscription must be dropped.

- If ***RESTRICT*** is specified, then a check of existing subscribers is performed. If there are subscribers, then an error is returned without dropping the subscription.
- If ***CASCADE*** is specified, then all subscribers of this subscription are also dropped.
- If neither ***RESTRICT*** nor ***CASCADE*** is specified, then ***RESTRICT*** is assumed.

Syntax for DROP SUBSCRIPTION

```
DROP SUBSCRIPTION <subscription_name> [RESTRICT | CASCADE];
```

Grant Subscribe

A user is then granted **SUBSCRIBE** privilege to subscribe to the subscription in order to track changes on the listed tables:

Syntax for GRANT SUBSCRIBE

```
GRANT SUBSCRIBE ON SUBSCRIPTION <subscription_name> TO <user_name>;
REVOKE SUBSCRIBE ON SUBSCRIPTION <subscription_name> FROM <user_name>;
```

Set Subscription

To set a subscription as active, an application issues a ***SET SUBSCRIPTION*** statement. The ***SET SUBSCRIPTION*** statement allows multiple subscriptions to be activated and includes an AT clause to denote a destination or device name as a recipient of subscribed changes. The subscriber user name is implied by the user identity of the database connection.

The notion of multiple subscriptions against the same schema object for a user, via the AT clause, is motivated by two observations:

- First, each subscription for a user might connote a separate device among many that have a disconnected interest in a change set that is queried independently at different times for different purposes.
- Second, some multiuser applications use pooled database connections under the umbrella of a single user name (e.g. CRM_User or even SYSDBA). In these cases, an alternate identifier must be provided to distinguish which subscription should be used to query a change set. That additional identifier can be loosely thought of as a destination or a "device name".

The ***SET SUBSCRIPTION*** statement activates/deactivates a subscription for a subscriber and binds/unbinds that subscription to the transaction that executed the command. All subscribed tables in a subscription behave as change views for table references in user-level queries when executed by a transaction with bound subscriptions. At transaction termination, the subscription is necessarily unbound, so it is necessary to issue a ***SET SUBSCRIPTION*** in subsequent transactions if a change view is still wanted.

At transaction commit, the subscriber transaction state is updated to allow the subscriber to see changes that occur after the commit, but not the former changes. A transaction rollback keeps the subscriber transaction state unchanged, so the change view still sees the former changes but cannot see newer

changes. If a transaction does not read any subscribed tables from a bound subscription then the subscriber transaction context for that subscription is not updated.

syntax and example

```
SET SUBSCRIPTION [<subscription_name> [, <subscription_name> ...]] [AT <destination>] {ACTIVE
| INACTIVE};

SET SUBSCRIPTION sub_employee_changes, sub_customer_deletes AT 'smartphone_123'
ACTIVE;
SELECT EMP_NO, DEPT_NO, SALARY FROM EMPLOYEE;
SELECT * FROM CUSTOMER;
COMMIT OR COMMIT RETAINING;
```

This example activates two subscription and returns changed data sets from the subscribed tables.

- The ***COMMIT*** updates all subscriptions for schema objects referenced during the transaction to set the last observed timestamp and transaction context.
- The ***COMMIT RETAINING*** does not change the last observed state and maintains the current snapshot as always. The subscription is unbound for the transaction at commit, which makes any subsequent queries against subscribed schema objects return normal data sets, without regard to changed data status. Any number of subscriptions can be activated simultaneously by a transaction.

Statement Execution

Once a statement is prepared, it is unnecessary to re-prepare the statement due to subscription activation or deactivation. A statement dynamically adjusts to the subscription environment of the transaction when it begins execution. Statement execution is also consistent in that once it begins, it returns change view result sets even if the subscription is deactivated before the full resultset has been fetched.

PREPARE Q;	PREPARE Q
SET SUBSCRIPTION ... ACTIVE	EXECUTE Q
EXECUTE Q	FETCH Q
FETCH Q	SET SUBSCRIPTION ... ACTIVE
SET SUBSCRIPTION ... INACTIVE	FETCH Q
FETCH Q	CLOSE Q
CLOSE Q	

Subscribed table references in statement Q will return change view rows even after the subscription has been made inactive. The converse also holds in that a regular (non-change view) resultset will remain consistent when the ***FETCH*** of the resultset is interspersed with subscription activation.

Change Views API Support

Change Views API support is provided through the extended SQLVAR structure, XSQLVAR, via a new interpretation of the SQLIND member. To review, a developer places a pointer to a variable in XSQLVAR.SQLIND to request NULL state. When the query is executed, InterBase places a zero at that pointer address if the column value for the returned row is non-***NULL*** and sets it to -1 if it is ***NULL***.

Under the new interpretation, the dual concepts of ***NULL*** state and ***CHANGE*** state are overlayed in the SQLIND member variable. The lower bits of the SQLIND variable are reserved as column change indicators: Bit 0 indicates ***INSERT***; Bit 1 indicates ***UPDATE***; Bit 2 indicates ***DELETE*** and Bit 3 designates ***CHANGE***. To check for a ***NULL*** state, a developer should check if SQLIND is less than 0 rather than for an explicit -1 value. A value greater than or equal to 0 stored at the SQLIND address indicates a non-NULL value.

NOT NULL	NULL	CHANGE STATUS
SQLIND = 0	SQLIND = -1	Legacy SQLIND values returned when Change Views and Subscriptions not used. Change status is undefined.
SQLIND >= 0	SQLIND < 0	New SQLIND values. Change status may be present and can be tested (as below.
SQLIND > 0	SQLIND < -1	Change status is present.

Once it is determined that the SQLIND value contains change status, it is necessary to clear the possible presence of the SQLIND_NULL bit before testing for specific state. The following SQLIND_xxxxxx defintions are included in <ibase.h>. By performing various bitwise OR operations on these definitions it is possible to test for interesting change status.

SQLIND VALUE	CHANGE STATUS
SQLIND_CHANGE_VIEW	The column value is <same> value; it did not change.
SQLIND_CHANGE_VIEW | SQLIND_CHANGE	It is <unknown> whether the column value changed.
SQLIND_CHANGE_VIEW | [SQLIND_CHANGE] | { SQLIND_INSERT | SQLIND_UPDATE | SQLIND_DELETE }	Some combination of SQL operations changed the column value.

SQLIND_CHANGE_VIEW is a tag bit to indicate the presence of change status. The remaining definitions correspond to the ***FOR ROW*** clause options of ***CHANGE***, ***INSERT***, ***UPDATE***, and ***DELETE*** respectively. The ***CHANGE*** option can cause <unknown> change state because a changed row is returned as soon as the presence of any column value change is detected. Those column values that did change return definite state, while the others return <unknown> state.

Changes in data made by a subscription is not visible by that subscription when observed at a later time. This is a bow to a possible application of change views as a component in bi-directional replication. When one side of a replication pair updates the other side with their local changes, that side does not want those changes to be reflected back when the other side replicates in the reverse direction.

Change Views SQL Language Support

The following example shows a retooling of the **ISQL** command-line utility that supports change views.

To display a list of subscriptions defined in the database, you can execute the ***SHOW SUBSCRIPTIONS*** command. To display details for a particular subscription, you can execute ***SHOW SUBSCRIPTION <name>***

ISQL SHOW SUBSCRIPTION

```
SHOW SUBSCRIPTIONS;
Subscription Name
============================================================
SUB_CUSTOMER_DELETES
SUB_EMPLOYEE_CHANGES
SUB_VARIOUS_CHANGES
```

```
SHOW SUBSCRIPTION sub_employee_changes;
Subscription name: SUB_EMPLOYEE_CHANGES
Owner: SYSDBA
Description: Subscribe TO changes IN EMPLOYEE TABLE
             EMPLOYEE (SALARY, DEPT_NO, EMP_NO)

SHOW SUBSCRIPTION sub_customer_deletes;
Subscription name: SUB_CUSTOMER_DELETES
Owner: SYSDBA
Description: Subscribe TO deletes IN CUSTOMER TABLE
             CUSTOMER FOR ROW (DELETE)

SHOW SUBSCRIPTION sub_various_changes;
Subscription name: SUB_VARIOUS_CHANGES
Owner: SYSDBA
Description: Subscribe TO various changes ON multiple TABLES
             EMPLOYEE FOR ROW (INSERT, UPDATE, DELETE),
             CUSTOMER FOR ROW (INSERT, UPDATE, DELETE),
             SALES FOR ROW (UPDATE),
             DEPARTMENT (LOCATION) FOR ROW (UPDATE)
```

ISQL SET CHANGES Command

ISQL has a collection of ***SET*** statements that toggle a display set. The ***SET CHANGES*** display toggle alternates between showing the column data value with its change status as a subordinated annotation. The <change> column is a pseudo column that shows the type of DML statement that modified the value of a column. All of this change state is returned by the XSQLVAR.SQLIND member of the new XSQLDA structure.

Retrieving Change Views from ISQL

```
<Another USER reassigns an existing employee TO another department AND gives another
employee a raise>

SET SUBSCRIPTION sub_employee_changes ACTIVE;
SELECT EMP_NO, DEPT_NO, SALARY FROM EMPLOYEE;

  EMP_NO        DEPT_NO        SALARY
     37             120         50000
     109            600         75000

SET CHANGES;

SELECT EMP_NO, DEPT_NO, SALARY FROM EMPLOYEE;
EMP_NO     <change>   DEPT_NO      <change>          SALARY         <change>
--------              ----------                    -----------
37          <same>     120          <update>         50000          <same>
109         <same>     600          <same>           75000          <update>

COMMIT;
```

SQL Extensions for Change Views

NOTE

This function was introduced with InterBase XE7 Update 1.

InterBase SQL provides support for Change Views with the ***IS [NOT] {CHANGED | INSERTED | UPDATED | DELETED}*** clause as the following example illustrates:

Using IS [NOT] UPDATED in SELECT queries

```
SET SUBSCRIPTION sub_employee_changes ACTIVE;
SELECT EMP_NO, DEPT_NO, SALARY FROM EMPLOYEE WHERE SALARY IS UPDATED;
EMP_NO          DEPT_NO                    SALARY
--------      ----------              ----------
109             600                         75000
```

We see that EMP_NO=37 employee's department reassignment is not returned since he received no compensation adjustment for a lateral move. The ***IS CHANGED*** clause will detect the modification of a column due to any kind of SQL operation.

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 re-link 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.

Working with 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 Embedded SQL 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 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 data types 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 data types.

The procedure body contains:

- An optional list of local variables and their data types.
- 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.

CREATE PROCEDURE syntax

```
CREATE PROCEDURE name
[(param data_type [, param data_type ...])]
 [RETURNS (param data_type [, param data_type ...])]
 AS
 <procedure_body>;
<procedure_body> = [<variable_declaration_list>]
 <block>
<variable_declaration_list> =
 DECLARE VARIABLE var data_type;
[DECLARE VARIABLE var data_type; ...]
<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> <data_type>	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. • <data_type>: An InterBase data type
RETURNS *<param data_type>*	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. • <data_type>: An InterBase data type • 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> <data_type>*	Declares local variables used only in the procedure • Each declaration must be preceded by ***DECLARE VARIABLE*** and followed by a semicolon (;). • <var>: 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 (;).

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 stored procedure and trigger language does not include many of the statement types available in DSQL or **gpre**. The following statement types are not supported in triggers or stored procedures:

- Data definition language statements: ***CREATE***, ***ALTER***, ***DROP***, ***DECLARE EXTERNAL FUNCTION***, and ***DECLARE FILTER***
- Transaction control statements: ***SET TRANSACTION***, ***COMMIT***, ***ROLLBACK***
- Dynamic SQL statements: ***PREPARE***, ***DESCRIBE***, ***EXECUTE***
- ***CONNECT/DISCONNECT***, and sending SQL statements to another database
- ***GRANT/REVOKE***
- ***SET GENERATOR***
- ***EVENT INIT/WAIT***
- ***BEGIN/END DECLARE SECTION***
- ***BASED ON***
- ***WHENEVER***
- ***DECLARE CURSOR***
- ***OPEN***
- ***FETCH***

The following table summarizes the language extensions for stored procedures.

Procedure and trigger language extensions	
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 */ ***or*** ***-- comment_text***	Programmer's comment. See Comment for more information and examples.
EXCEPTION ***<exception_name>***	Raises the named exception. Exception: A user-defined error that 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. Enables nested procedures and recursion. Input and output parameters must be variables defined within the procedure.
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 that 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***.

Procedure and trigger language extensions	
Statement	**Description**
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.

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 data types. 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 data types. 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 data_type [, var data_type ...])
. . .
```

Each input parameter declaration has two parts: a name and a data type. The name of the parameter must be unique within the procedure, and the data type can be any standard SQL data type except arrays of data types. 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 data_type [, var data_type ...])]
AS
...
```

Each output parameter declaration has two parts: a name and a data type. The name of the parameter must be unique within the procedure, and the data type can be any standard SQL data type except 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 F***OR 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.

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 data_type;
```

where <var> is the name of the local variable, unique within the procedure, and <data_type> is the data type, which can be any SQL data type 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.

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.

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 With Procedures.

In a ***SELECT*** statement that retrieves values from a procedure, the column names must match the names and data types 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 data types 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 data type 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 Embedded SQL 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:

```
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 ;
```

If this procedure is called from isql with the command:

```
EXECUTE PROCEDURE SUM_INT 4;
```

then the results are:

```
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 Embedded SQL 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.

There are two different types of comments that you can use:

1. The **simple comment**: A comment that starts with a special symbol and ends with a new line.

NOTE

The **simple comment** syntax is only available starting with database engine version InterBase 2017.

```
-- comment text
```

2. The **bracketed comment**: A comment that starts and ends with a special symbol. It may be multi-line.

```
/* comment text
more comment text
another line of comment text
*/
```

Regardless of the type of comment that you use, you may start a comment anywhere in a line, but with a simple comment you need to keep in mind that the comment area stops after new line. In order to use the simple comment syntax for a multi-line comment, you need to start each line with the special symbol. For example:

- A multi-line bracketed comment:

```
/* my multi-line
comment is this
text */
```

- A multi-line simple comment:

```
-- my multi-line
-- comment is this
-- text
```

You can place comments on the same line as code, which makes them inline comments.

It is good programming practice to state the input and output parameters of a procedure in a comment preceding the procedure. It is also often useful to comment local variable declarations to indicate what each variable is used for.

Examples The following isql samples illustrate some ways to use comments:

```
/*
* Procedure DELETE_EMPLOYEE : Delete an employee.
*
* Parameters:
* employee number
* Returns:
* --
*/
CREATE PROCEDURE DELETE_EMPLOYEE (EMP_NUM INTEGER)
AS
DECLARE VARIABLE ANY_SALES INTEGER; -- Number of sales for emp.
BEGIN
. . .
```

```
/* This script sets up Change Views Subscriptions
    on the EMPLOYEE table.
*/
CONNECT "emp.ib" USER 'SYSDBA' password 'masterkey';
COMMIT;

CREATE SUBSCRIPTION sub ON EMPLOYEE FOR ROW (INSERT, UPDATE, DELETE);
COMMIT;
```

```
-- Create a subscription on Employee table
CREATE SUBSCRIPTION sub1 ON EMPLOYEE FOR ROW (INSERT, UPDATE);
COMMIT;
```

- Simple comment followed by another SLC

```
-- One more comment
CREATE SUBSCRIPTION sub2 ON EMPLOYEE FOR ROW (INSERT);
COMMIT;
```

- Simple comment followed by another SLC with leading whitespace

```
      -- One more comment followed by leading whitespace before CREATE below
CREATE SUBSCRIPTION sub3 ON EMPLOYEE FOR ROW (INSERT, UPDATE, DELETE);
COMMIT;

SHOW SUBSCRIPTIONS;

SELECT COUNT(*)
      -- inline comment 1
      FROM RDB$DATABASE;

SELECT COUNT(*) -- inline comment 2
      FROM RDB$DATABASE;

COMMIT;

SET TERM ^;
```

- Create a stored procedure with inline comments

```
CREATE PROCEDURE test_proc (
   p1 INTEGER, -- Param 1
   p2 VARCHAR(68)   -- Param 2
)
RETURNS (op1 INTEGER)     -- Output param
AS
DECLARE variable v1 INTEGER;
DECLARE variable v2 VARCHAR(150); -- Variable 2
BEGIN
   -- sample comment 1
   -- sample comment 2
-- return input value multiplied by 10
   v1 = p1 * 10;
   op1 = v1;
   SUSPEND;
END^
SET TERM ;^
COMMIT;
SHOW PROCEDURE test_proc;
SELECT op1 FROM test_proc (2, NULL);
```

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.

```
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 ;
```

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 With Procedures

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:

SUSPEND, EXIT, and END			
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

Consider the following procedure:

```
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 ;
```

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 a SQLCODE error, GDSCODE error, or user-defined exception—all statements since the last ***SUSPEND*** are undone.

Since select procedures can have multiple ***SUSPEND***s, 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 and Dropping Stored Procedures

This section describes techniques and issues for changing and deleting procedures.

TIP

To see a list of database procedures and their dependencies, use the ***isql*** command:

```
SHOW PROCEDURES;
```

Altering Stored Procedures

To change a stored procedure, use ***ALTER PROCEDURE***. This statement changes the definition of an existing stored procedure while preserving its dependencies according to which metadata objects reference the stored procedure, and which objects the stored procedure references.

Changes made to a procedure are transparent to all client applications that use the procedure; you do not have to rebuild the applications. However, see Altering and Dropping Procedures in Use for issues of managing versions of stored procedures.

Only SYSDBA and the owner 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 you alter a procedure, the new procedure definition replaces the old one. 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.

ALTER PROCEDURE syntax

The syntax for ***ALTER PROCEDURE*** is similar to ***CREATE PROCEDURE*** as shown in the following syntax:

```
ALTER PROCEDURE name
[(var data_type [, var data_type ...])]
[RETURNS (var data_type [, var data_type ...])]
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 those for ***CREATE PROCEDURE*** (see Arguments of the ***CREATE PROCEDURE*** statement on the page CREATE PROCEDURE syntax).

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.

The following restrictions apply to dropping procedures:

- Only SYSDBA and the owner of a procedure can drop it.
- You can't drop a procedure used by other procedures, triggers, or views; alter the other metadata object so that it does not reference the procedure, then drop the procedure.
- You can't drop a procedure that is recursive or in a cyclical dependency with another procedure; you must alter the procedure to remove the cyclical dependency, then drop the procedure.
- You can't drop a procedure that is currently in use by an active transaction; commit the transaction, then drop the procedure.
- You can't drop a procedure with embedded SQL; use dynamic SQL.

If you attempt to drop a procedure and receive an error, make sure you have entered the procedure name correctly.

Drop Procedure Syntax

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;
```

Altering and Dropping Procedures in Use

You must make special considerations when making changes to stored procedures that are currently in use by other requests. A procedure is in use when it is currently executing, or if it has been compiled internally to the metadata cache by a request.

Changes to procedures are not visible to client applications until they disconnect and reconnect to the database; triggers and procedures that invoke altered procedures don't have access to the new version until there is a point in which all clients are disconnected.

To simplify the task of altering or dropping stored procedures, it is highly recommended to perform this task during a maintenance period when no client applications are connected to the database. By doing this, all client applications see the same version of a stored procedure before and after you make an alteration.

TIP

You can minimize the maintenance period by performing the procedure alteration while the database is in use, and then briefly closing all client applications. It is safe to alter procedures while the database is in use.

Internals of the technology:

Below is a detailed description of the internal maintenance of stored procedure versions, to help explain the behavior of the technology.

When any request invokes a stored procedure, the current definition for that stored procedure is copied at that moment to a metadata cache. This copy persists for the lifetime of the request that invoked the stored procedure.

A request is one of the following:

- A client application that executes the stored procedure directly
- A trigger that executes the stored procedure; this includes system triggers that are part of referential integrity or check constraints
- Another stored procedure that executes the stored procedure

Altering or dropping a stored procedure takes effect immediately; new requests that invoke the altered stored procedure see the latest version. However, outstanding requests continue to see the version of the stored procedure that they first saw, even if a newer version has been created after the request's first invocation of the stored procedure. There is no method to force these outstanding requests to update their metadata cache.

A trigger or stored procedure request persists in the metadata cache while there are one or more clients connected to the database, regardless of whether the client makes use of the trigger or stored procedure. These requests never update as long as any client is connected to the database. These requests are emptied from the metadata cache only when the last client disconnects from the database.

IMPORTANT

The only way to guarantee that all copies of a stored procedure are purged from the metadata cache is for all connections to the database to terminate. Only then are all metadata objects emptied from the metadata cache. Subsequent connections and triggers spawned by them are new requests, and they see the newest version of the stored procedure.

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 Embedded SQL 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.

NOTE

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.

```
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
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 data type 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:

```
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  ;
```

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.

Stored Procedure 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.

NOTE

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.

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.

NOTE

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.

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.

```
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;!
```

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 Guide.

Examples of Error Behavior and Handling

When a procedure encounters an error – either a 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 ***SUSPEND*** statemets, 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.

```
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 ;
```

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
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 employee number is selected, raises the exception, CEO, which is handled with a ***WHEN*** statement that assigns the CEO 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.';
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 ;
```

This procedure can be called with a statement such as:

```
SELECT * FROM NEW_PROJECT('XYZ', 'Alpha project', 'software');
```

With results such as the following:

```
RESULT                                  NUM
============================            ======
New Project Inserted OK.                <null>
Project Got Employee Number:            28
Project Got Employee Number:            29
Project Got Employee Number:            36
```

Working with Triggers

This chapter covers the following topics:

- What triggers are, and the advantages of using them
- How to create, modify, and drop triggers
- How to use triggers
- How to raise exceptions in triggers

About 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 re-link.
- **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.

Working with Triggers 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 data types.
- 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.

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 data_type;
[DECLARE VARIABLE variable data_type; ...]
<block> =
BEGIN
<compound_statement>
  [<compound_statement> ...]
END
<compound_statement> = <block> | statement;
```

Arguments of the CREATE TRIGGER 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> <data_type>	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. <data_type>: The data type 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; deprecated in InterBase 7.0. [No longer needed]

InterBase 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 stored procedure and trigger language does not include many of the statement types available in DSQL or **gpre**. The following statement types are not supported in triggers or stored procedures:

- Data definition language statements: ***CREATE***, ***ALTER***, ***DROP***, ***DECLARE EXTERNAL FUNCTION***, and ***DECLARE FILTER***
- Transaction control statements: ***SET TRANSACTION***, ***COMMIT***, ***ROLLBACK***
- Dynamic SQL statements: ***PREPARE***, ***DESCRIBE***, ***EXECUTE***
- ***CONNECT/DISCONNECT***, and sending SQL statements to another database
- ***GRANT/REVOKE***
- ***SET GENERATOR***
- ***EVENT INIT/WAIT***
- ***BEGIN/END DECLARE SECTION***
- ***BASED ON***
- ***WHENEVER***
- ***DECLARE CURSOR***
- ***OPEN***
- ***FETCH***

The following table summarizes the language extensions for stored procedures.

Procedure and trigger language extensions	
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 */ ***or*** ***-- comment_text***	Programmer's comment. See Comment for more information and examples.
EXCEPTION <exception_name>	Raises the named exception. Exception: A user-defined error that can be handled with ***WHEN***.

Procedure and trigger language extensions	
Statement	**Description**
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. Enables nested procedures and recursion. Input and output parameters must be variables defined within the procedure.
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 that 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.
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.

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. In read-only databases, generators can return their current value but cannot increment.

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 Working with Stored Procedures (Data Definition Guide).

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.

```
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 ;
```

NOTE

Context variables are never preceded by a colon, even in SQL statements.

Using Generators in the Trigger Body

In a read-write database, a generator is a database object that automatically increments each time the special function, ***GEN_ID()***, is called.

IMPORTANT

Generators cannot be used in read-only databases.

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 in triggers, but they are particularly useful in triggers for inserting unique column values.

Use the ***CREATE GENERATOR*** statement the create a generator and ***SET GENERATOR*** to initialize it. If not otherwise initialized, a generator starts with a value of one. For more information about creating and initializing a generator, see ***CREATE GENERATOR*** and ***SET GENERATOR*** in 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 SQL 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> can 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:

```
CREATE TRIGGER SET_CUST_NO FOR CUSTOMER
BEFORE INSERT AS
BEGIN
NEW.CUST_NO = GEN_ID(CUST_NO_GEN, 1);
END ;
```

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 Working with Tables (Data Definition Guide).

NOTE

Direct metadata operations, such as altering triggers, increase the metadata version. At most 255 such operations can be performed before you must back up and restore the database.

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.

NOTE

This example assumes that you have a table named NEW_CUSTOMERS with a column cust_no.

```
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 ;
```

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;
```

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.

NOTE

Direct metadata operations, such as dropping triggers, increase the metadata version. At most 255 such operations can be performed before you must back up and restore the database.

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>. Privileges of triggers can be revoked similarly using ***REVOKE***. For more information about ***GRANT*** and ***REVOKE***, see 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:

```
CREATE TRIGGER POST_NEW_ORDER FOR SALES
AFTER INSERT AS
BEGIN
POST_EVENT 'NEW_ORDER';
END ;
```

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 Embedded SQL 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-update-able views into update-able views.

If you define ***BEFORE*** triggers for a view that the InterBase engine considers to be directly update-able, on an ***UPDATE***, ***DELETE***, or ***INSERT*** operation the ***BEFORE*** trigger will fire; also, the default action attempted by the ***UPDATE/DELETE/INSERT*** statement will be executed, generating two actions and hence unexpected results.

NOTE

Not all views can be made update-able by defining triggers for them. For example, this read-only view attempts to count records from the client; but regardless of the triggers you define for it, all operations except ***SELECT*** always fail:

```
CREATE VIEW AS SELECT 1 FROM MyTable;
```

TIP

You can specify non-default behavior for update-able views, as well. InterBase does not perform write-throughs 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 Update-able.

Example: 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;
```

Trigger 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 Working with Stored Procedures (Data Definition Guide).

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 Working with Stored Procedures (Data Definition Guide).

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 Working with Stored Procedures (Data Definition Guide).

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:

```
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 END ;
```

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 Working with Stored Procedures (Data Definition Guide).

Working with Generators

This chapter covers the following topics:

- What a generator is
- How to create, modify, and drop generators
- Using generators

About Generators

A generator is a mechanism that creates a unique, sequential number that is automatically inserted into a column in a read-write 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 use 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^{63} to $2^{63}-1$. 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

Once a generator has been created using the ***CREATE GENERATOR*** statement, it exists within the database but no numbers have actually been generated. 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. Applications can also use ***GEN_ID()*** with ***SELECT*** statements to obtain a generator value for inclusion as part of an ***INSERT*** statement.

The syntax for ***GEN_ID()*** is:

```
GEN_ID(genname, step);
```

To generate a number, follow these steps:

1. Create the generator.
2. Within a trigger, stored procedure, or application, reference the generator with a call to ***GEN_ID()***.
3. The 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.

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()***.

IMPORTANT

Generators return a 64-bit value. You should define the column that holds the generated value as an ISC_INT64 variable with a DECIMAL or NUMERIC data type.

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));
```

For more information on using generators in triggers, see Triggers (Data Definition Guide). For more information on using generators in stored procedures, see Working with Stored Procedures.

Dropping Generators

To drop a generator from a database, use the following syntax:

```
DROP GENERATOR generator_name
```

The ***DROP GENERATOR*** command checks for any existing dependencies on the generator (as in triggers or UDFs) and fails if such dependencies exist. The statement fails if generator_name is not the name of a generator defined on the database. An application that tries to call a deleted generator returns runtime errors.

NOTE

In previous versions of InterBase that lacked the ***DROP GENERATOR*** command, users issued a SQL statement to delete the generator from the appropriate system table. This approach is strongly discouraged now that the ***DROP GENERATOR*** command is available, since modifying system tables always carries with it the possibility of rendering the entire database unusable as a result of even a slight error or miscalculation.

Planning Security

This chapter discusses the following topics:

- 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
- Additional security measures

NOTE

For information about **the InterBase encryption feature**, which enables encryption at the database and column levels, and about the privileges needed to grant and revoke encrypt and decrypt permissions, see Encrypting Your Data.

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***, ***DECRYPT***, and ***REFERENCES*** privileges for a table to a role.
- Grant ***SELECT***, ***INSERT***, ***UPDATE***, ***DECRYPT***, and ***DELETE*** privileges for a view to a role.
- Grant ***ENCRYPT ON ENCRYPTION*** permission to a user.
- 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 creator of a table, 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 creator of a procedure, 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:

SQL access privileges	
Privilege	**Access**
ALL	Select, insert, update, delete data, and reference a primary key from a foreign key.
SELECT	Read data.
INSERT	Write new data.
UPDATE	Modify existing data.
DELETE	Delete data.
ENCRYPT ON ENCRYPTION	Enables the database owner or individual table owner to use a specific encryption key to encrypt a database or column. Only the SYSDSO (Data Security Owner) can grant encrypt permission. For information about the InterBase encryption feature, which enables encryption at the database and column levels, and about the privileges needed to grant and revoke encrypt and decrypt permissions, see Encrypting Your Data.
DECRYPT	After encrypting a column, the database owner or the individual table owner can grant decrypt permission to users who need to access the values in an encrypted column. For information about the InterBase encryption feature, which enables encryption at the database and column levels, and about the privileges needed to grant and revoke encrypt and decrypt permissions, see Encrypting Your Data.
REFERENCES	Reference a primary key with a foreign key.
EXECUTE	Execute or call a stored procedure.
role	All privileges assigned to the role.

The ***ALL*** keyword provides a mechanism for assigning ***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 implements features for assigning SQL privileges to groups of users, fully supporting SQL group-level security with the ***GRANT***, ***REVOKE***, and ***DROP ROLE*** statements. It partially supports ***GRANT ROLE*** and ***REVOKE ROLE***.

NOTE

These features replace the Security Classes feature in versions prior to InterBase 5. In the past, group privileges could be granted only through the InterBase-proprietary GDML language. In Version 5, new SQL features were 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 role name TO user***.
4. Users 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 Guide.

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.
- Using Roles to Grant Privileges 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.
- The power to grant ***GRANT*** authority is discussed in Granting Users the Right to Grant Privileges.
- Granting ***EXECUTE*** privileges on stored procedures is discussed in Granting Privileges to Execute Stored Procedures.

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, (***admin.ib*** by default), 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> [WITH GRANT OPTION]
  | GROUP <UNIX_group>}
  | EXECUTE ON PROCEDURE procname TO { <object>
  | <userlist>}
```

```
 | <role_granted> TO {PUBLIC | <role_grantee_list> }[WITH GRANT OPTION];
<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> = {
 &amp;nbsp;&amp;nbsp;[USER] username
 | rolename
 | UNIX_user }[, <userlist> ...]
<role_granted> = rolename[, rolename ...]
<role_grantee_list> = [USER] username[, [USER] username ...]
```

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 does not 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 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. Now OS-level groups share this behavior, and database administrators can assign SQL privileges to UNIX groups through SQL ***GRANT*** /***REVOK***E 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 a 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, 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 ON tablename TO rolename;
```

3. Use the ***GRANT*** statement once again to grant the role to one or more users.

```
GRANT rolename 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 Guide 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 the following section 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.

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 user'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.

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.

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 view, since it is not directly update-able. Views that are based on a single table which have no aggregates or reflexive joins are often update-able. See Types of Views: Read-only and Update-able 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 update-able. 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.

Update-able Views

You can assign **SELECT**, **UPDATE**, **INSERT**, and **DELETE** privileges to update-able 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> = ALL [PRIVILEGES] | <privilege_list>

<privilege_list> = {
 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 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.

For more information about integrity constraints and triggers, see Triggers (Data Definition Guide).

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 REVOKE.

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.

Encrypting Your Data

This chapter provides information and instruction on the following topics:

- Overview of InterBase encryption
- An overview of encryption tasks
- A description of encryption users
- Encrypt and decrypt permissions
- Using isql to enable and perform encryption
- Using IBConsole to enable and perform encryption
- Encrypting backup files

The InterBase encryption feature is available in InterBase 2009 and after.

About InterBase Encryption

Encryption is the process of applying an invertible algorithm to a block of data (plaintext) so that the encrypted data (ciphertext) bears no resemblance to the plaintext. Typically, an encryption key is applied to the plaintext to produce the ciphertext. The same encryption key is used to convert (decrypt) the ciphertext back to plaintext. The purpose of encryption is to protect data from being deciphered by unauthorized viewers or users.

InterBase enables you to encrypt information at one or both of the following levels:

- **Database-level Encryption:**

When you specify database-level encryption, InterBase encrypts all the database pages that contain user information. Non-user database pages are not encrypted. Non-user pages include the header page, log page, inventory pages, pointer pages, transaction inventory pages, index root pages, and generator pages.

You cannot specify which pages in a database to encrypt. Instead, you issue the encrypt database command from the database to which you are connected, and InterBase encrypts all the user-related pages in that database.

- **Column-level Encryption:**

Column-level encryption is both more flexible and more specific. To encrypt a column, you specify the table that contains the column, followed by the name of the column. You can encrypt all of the columns in a table, or only individual columns you specify. For example, you can encrypt a payroll column in an Employee table so that **both** payroll and HR employees can access it. Then you might encrypt SSN information in the same table so that **only** payroll employees can access it. Users who need to access data in encrypted columns can be given decrypt privileges for that column.

NOTE

When working with encrypted columns, the ***MIN***, ***MAX***, ***BETWEEN***, and ***ORDER BY*** operations cannot use an index based on those fields due to the nature of the index key that is formed from the encrypted field value. So while the index is not useful for the above operations, it is still useful for equality matches and ***JOIN*** operations

Generally speaking, encrypting all of the user pages of a database takes much greater overhead than selectively encrypting individual columns. In addition, database performance can be adversely affected when a large number of concurrent queries access the same encrypted columns.

Encrypting Database Backup Files

To maintain the security and confidentiality of encrypted databases, you must also encrypt database backup files. The GBAK utility provides three additional switches to facilitate encrypt and decrypt operations on database backups. For instructions on how to encrypt and decrypt backup files, see Encrypting Backup Files.

Encrypting Network Communication (InterBase Encryption)

Data passed to a remote InterBase client from a database server is unencrypted during the transmission process. For information on how to encrypt information that is passed over a network, see Encrypting Your Data the InterBase Operations Guide.

About Industry Encryption Standards

InterBase encryption supports the use of the following industry encryption standards:

- **Data Encryption Standard (DES)** is a 25-year-old industry standard. DES is a weak encryption standard but does not require a license to use.
- **Advanced Encryption Standard (AES)** was adopted as a federal standard in 2002. AES enables a larger number of bits with which to scrambled data than DES does. Because AES offers much stronger encryption protection, the United States regulates its export. To address U.S. export regulations, users must obtain an InterBase license to use AES with InterBase. InterBase includes AES in the Server, ToGo and Desktop editions. Please, see About AES for more information about this stardard.

NOTE

You do not need an AES addon license starting with InterBase XE3 release. InterBase 2009 and InterBase XE required the addon license.

You specify the standard you want to use with InterBase when you create an encryption key. Instructions on how to create the encryption key are provided later in this chapter.

Who Can Create Encryption?

Encryption tasks, which are summarized in the table on the page An Overview of Encryption Tasks, are primarily performed by the following users: a **SYSDSO**, the **database owner**, and any **individual table owners** who are given permission to encrypt specific columns in a table. InterBase requires the creation of the **System Database Security Owner (SYSDSO)** user to implement specific encryption tasks. SYSDSO is a reserved user name, similar to SYSDBA.

The **database owner** is typically the person who creates the database. The database owner may or may not also be the administrator of the database.

The **SYSDSO** role controls three significant steps in the encryption process:

- Creates a System Encryption Password (SEP).
- Creates the encryption keys.
- Grants the database owner access to the encryption keys, which s/he then uses to encrypt the database and/or its columns.

However, the SYSDSO cannot encrypt databases or columns, nor can s/he grant or revoke access to encrypted data. **Only adatabase owner** and/or an **individual table ownercan** actually **encrypt a database or columns in a database**; the SYSDSO simply creates the tools (the encryption keys) that are needed to perform the encryption. Requiring that multiple users set up and implement encryption, rather than just one, adds an additional layer of database security.

In addition, only the user who encrypts a column or database can grant decrypt privileges to those who need to view or modify the encrypted data. For more information about granting decrypt permission, see Granting Decrypt Permission.

Generally speaking, only the user who grants the permission can revoke the permission. For more information, see Encrypting Your Data.

NOTE

Decrypt permission is only required for column-level encryption. It is not required for database-level encryption.

Creating the SYSDSO User

The database owner uses the following syntax to create the SYSDSO user:

```
CREATE USER SYSDSO SET PASSWORD 'PASSWORD';
```

You must keep the SYSDSO user for as long as you use the encryption keys created by that same SYSDSO.

If the SET PASSWORD clause is not specified, the default SYSDSO password will be the password of the person who creates the account. This makes it easier for the account creator to temporarily acquire SYSDSO privileges to create and test encryptions during development without having to login to do so. When the SYSDSO password is subsequently changed, the account creator loses this privilege. Presumably, this handoff would occur at deployment time, when transferring these duties to a security authority.

An Overview of Encryption Tasks

The following list identifies the tasks that need to be performed to encrypt a database and/or its columns, and to give users the appropriate access rights. The steps are typically performed by a SYSDSO and a database owner unless additional individuals are given encrypt privileges to specific columns. See Who Can Create Encryption? for more information about how the SYSDSO and database owner use the InterBase encryption feature.

To implement encryption using InterBase, perform the tasks listed in the table. The following sections provide detailed instructions on how to perform steps 3-7.

Step #	Task	Performed by
1	Ensure that Embedded User Authentication (EUA) is enabled on the database you plan to encrypt. For instructions on how to enable EUA using isql, see the InterBase Operations	Database owner

Step #	Task	Performed by
	Guide. For instructions on how to enable EUA using IBConsole, see Encrypting a Database with IBConsole of this chapter.	
2	Create a System Database Security Owner (SYSDSO) account using the command on Creating the SYSDSO User.	Database owner
3	Create a System Encryption Password (SEP).	SYSDSO
4	Create an encryption key for the database and/or the columns you want the database or table owner to encrypt.	SYSDSO
5	Grant the database owner privileges to use the encryption keys to perform encryption.	SYSDSO
6	Encrypt the database and/or columns.	Database owner or individual table owner
7	Grant or revoke decrypt privileges to other users.	Database owner or individual table owner

Requirements and Support

InterBase encryption is supported on all InterBase platforms. Before using it, you must install or do the following:

- **Embedded User Authentication (EUA)** must be enabled to grant specified users decrypt privileges to access data in encrypted columns. For instructions on how to enable EUA using isql, see the InterBase Operations Guide. For instructions on how to enable EUA using IBConsole, see Encrypting a Database with IBConsole of this chapter.
- Due to government regulation of strong encryption, you must obtain a **license** from InterBase **to use AES with InterBase**.You do not need a special license to use DES with InterBase.
- **This feature requires ODS 13** and is not available on older ODS databases. Therefore, a backup and restore to ODS 13 is required for pre-existing databases to use InterBase encryption. For more information about performing backups and restores, see the InterBase Operations Guide.

The table below shows which primary ODS version corresponds to each InterBase release:

InterBase version	Primary ODS version
2017	17
XE7	16
XE3	15
XE	15
2009	13
2007	12
7.0	11
6.0	10

NOTE

The InterBase release may support certain older ODS versions as well (see the Migration Issues section of the Readme for that release). There is no official release of ODS 14 as a primary ODS.

NOTE

InterBase uses **OpenSSL** or a derivation of that version to support InterBase encryption. InterBase embeds OpenSSL libraries in InterBase server/client to help implement encryption-related features. OpenSSL contains libraries for the most widely known encryption and message digest algorithms in use today. InterBase uses these libraries as the basis for supporting database and column-level encryption functionality.

Using isql to Enable and Implement Encryption

This section explains how to enable and implement encryption using isql. For instructions on to use IBConsole to perform the same encryption tasks, see Encrypting a Database with IBConsole.

Setting the System Encryption Password (SEP)

InterBase uses a System Encryption Password (SEP) to protect the encryption keys that are used to encrypt the database and/or database columns. If you are managing multiple databases that use InterBase encryption, it is recommended that you create a different SEP for each database.

Altering the Database to Create the SEP

The SYSDSO uses the ***ALTER DATABASE*** command to create the SEP.

To create a SEP, use the following syntax:

```
alter database set system encryption password <255-character string>
```

The string can be up to 255 characters long and can include spaces. The system encryption password is encrypted with a key derived from machine specific information and stored in the database. This effectively node locks the database to the machine but allows the database to be attached without a user having to pass the system encryption password in plaintext. Thus, subsequent connections on the same machine need not provide the SEP.

However, if the database file is copied and installed on a different machine, the node-lock feature disallows direct loading of the database without the user providing the SEP. After moving a database with a node-locked SEP to another machine, you must login as SYSDSO with the current SEP set via the SEP environment variable or DPB. The SYSDSO can then perform ***ALTER DATABASE SET SYSTEM ENCRYPTION PASSWORD*** to create a new SEP.

Just "setting" the SEP to connect to the database does not redefine or re-node-lock the SEP. You can continue to provide the SEP externally though you may want to alter the sep command to re-node-lock it to the new machine.

Using the External Option when Creating a SEP

Though an unauthorized person would not have decrypt permission for any encrypted columns, he or she might be able to bit edit the database file to artificially grant decrypt permission. The password attribute of an encryption key can mitigate this risk because the user needs the passwords as well as decrypt permission. For database-level encryption, the data would be visible immediately because only the SEP is needed to see it.

Adding the **external** setting to a SEP statement can make it more difficult for unauthorized users to access an encrypted database on a mobile device such as a laptop computer, or on an a poorly secured desktop computer.

```
alter database set system encryption password <255-character string> [external]
```

The external form of setting the SEP requires the first database attach to pass the isc_dpb_sys_encrypt_password parameter with the value of the password, or to set the environment variable isc_system_encrypt_password. Subsequent database attachments are not required to pass the SEP as the database server already has it in memory.

For security reasons, programs should not hardcode the SEP with isc_dpb_sys_encrypt_password but query the user, then generate this database attachment parameter dynamically. The ISC_SYSTEM_ENCRYPT_PASSWORD environment variable should never be hardcoded in scripts and if entered at the console should be unset as soon as possible.

Removing the System Encryption Password (SEP)

The SYSDSO can remove the SEP when the database is no longer encrypted, and when there are no remaining column-level encryptions stored in the RDB$ENCRYPTIONS table.

To remove a SEP, use the following syntax:

```
alter database set no system encryption password
```

Creating Encryption Keys

The SYSDSO uses the ***CREATE ENCRYPTION*** command to create encryption keys. **An encryption key** is used to encrypt the pages and/or columns of a database. The database owner uses an encryption key to perform encryption on a specific database or column. InterBase stores encryption keys in the RDB $ENCRYPTIONS system table.

The following statement provides an example of a simple ***CREATE ENCRYPTION*** statement:

```
CREATE ENCRYPTION payroll_key for AES
```

where ***CREATE ENCRYPTION*** is the command, and "payroll_key" is the name of the key created. Thus, the basic syntax for creating an encryption key is:

```
CREATE ENCRYPTION key-name for AES | for DES
```

To create an encryption key using all of the available isql statement options, use the following syntax:

```
CREATE ENCRYPTION key-name [as default] [for {AES| DES}] [with length number-of-bits [bits]]
[password {'user-password' | system encryption password}] [init_vector {NULL | random}] [pad
{NULL | random}] [description 'some user description']
```

For example:

```
CREATE ENCRYPTION revenue_key FOR AES WITH LENGTH 192 BITS INIT_VECTOR RANDOM
```

See the following table for a description of each encryption key option:

Option	Description
Key name	Identifies the encryption key by a unique name.
Default	This key is used as the database default when no explicit key is named for database or column encryption.
AES	Advanced Encryption Standard algorithm. This encryption scheme is considered strong and requires an InterBase license.
DES	Data Encryption Standard algorithm. This is a weak encryption scheme that requires no special license.
Length	Specifies key length. If using DES, 56 bits is the default. If using AES, you can specify 128, 192, or 256 bits. For AES, 128 is the default.
Password	Available only for column encryption keys. Associating a custom password with an encryption key provides an additional layer of protection. For more information about associating a custom password with an encryption key, see Setting a User-defined Password for an Encryption Key.
Init-vector	Random enables Cipher Block Chaining (CBC) encryption technique so that equal values have different ciphertext. If NULL is specified, then Electronic Cookbook (ECB) is used. NULL is the default value.
Pad	Random padding can cause equal values to have different ciphertext. NULL specifies that random padding should not occur. NULL is the default value.
Description	A user-level comment that describes the purpose of the encryption.

NOTE

A random initialization vector or random padding prevents an encrypted column from being used in an index, and raises an error if a create index DDL statement tries to do so. The ***NULL*** defaults for both of these options favor index-enabled access optimization over a more stringent level of protection afforded by the random counterparts.

Setting a User-defined Password for an Encryption Key

As noted in the table above, you can assign each column encryption key a custom password, which adds an additional level of protection for your data. When you associate a password with a column encryption key, you must give it to the database owner or the table owner so that s/he can use the key to encrypt the column. You must also give it to any end users who need to change or view the values in the encrypted column.

If an encryption key was defined with a user-defined password, then users must set the password during a database session before accessing columns that have been encrypted with the key:

NOTE

The System Encryption Password (SEP) is the default if no PASSWORD clause is provided to CREATE ENCRYPTION.

```
set password '<user-password>' for {encryption <encryption_name> | column
<table.column_name>}
```

Assuming the same user also has decrypt and access permissions on the column, he or she can now access all columns encrypted by that key.

Dropping an Encryption Key

An encryption key can be dropped (deleted) from the database. Only the SYSDSO can execute this command. The command will fail if the encryption key is still being used to encrypt the database or any table columns when "restrict" is specified, which is the default drop behavior. If "cascade" is specified, then all columns using that encryption are decrypted and the encryption is dropped.

To drop an encryption key, use the following syntax:

```
key-name [restrict | cascade]
```

Granting Encryption Permission to the Database Owner

In order for the database owner to use an encryption key to encrypt a database or column, the SYSDSO must first grant **encrypt** permission to the database or table owner to use the key. Only the SYSDSO can grant **encrypt** permission.

To grant permission to encrypt, use the following syntax:

```
GRANT ENCRYPT ON ENCRYPTION key-name to user-name;
```

For example, if a SYSDBA is the database owner:

```
GRANT ENCRYPT ON ENCRYPTION expenses_key to SYSDBA;
```

gives the SYSDBA permission to use the payroll-key to encrypt a database or a column.

IMPORTANT

Only the user who encrypts a column or database can grant decrypt privileges to those who need to view the encrypted data. Only the database owner can grant decrypt privileges.

Encrypting Data

As indicated at the beginning of this chapter, InterBase can be used to encrypt data at the database-level, and to encrypt specific columns in a database. Generally speaking, encrypting at the column-level offers the greatest data protection. When you encrypt at the database- or column-level, you must also encrypt the backup files of the database. For instructions on how to do so, see Encrypting Backup Files.

About the Encryption Commands

InterBase provides two encryption commands: one to encrypt a database, and the other to encrypt database columns.

To encrypt a database, use the following syntax:

```
ALTER DATABASE ENCRYPT [[with] key-name]
```

For example, the statement:

```
ALTER DATABASE ENCRYPT with fin_key
```

uses the fin_key to encrypt all the database pages in the current database (i.e. in the database to which you are connected).

To encrypt a column in an existing table, use the following syntax:

```
ALTER TABLE table-name ALTER COLUMN column-name ENCRYPT [[with] key-name]
```

For example, the following statement:

```
ALTER TABLE sales ALTER COLUMN total_value ENCRYPT with expenses_key
```

uses the expenses_key to encrypt data in the total_value column.

To encrypt a column when creating a table, use the following syntax:

```
CREATE TABLE table-name column-name data-type ENCRYPT [[with] key-name]
```

Setting a Decrypt Default Value for a Column

When encrypting a column, the database or table owner can specify a decrypt default value that displays when a user who does not have decrypt privileges for that column tries to access the column's data. If a decrypt default value is not specified, the user will get an error message. A decrypt default value also allows existing reports and applications to run without raising permission exceptions when columns are encrypted.

To specify a decrypt default value, use the following syntax:

```
create table table-name (column-name data-type encrypt [[with] key_name] [decrypt default
value], ...)
```

A decrypt default can be changed or dropped from a column. Note that a decrypt default is not automatically dropped when a column is decrypted.

```
alter table table-name alter [column] column-name [no] decrypt default value
```

Encrypting Blob Columns

Blob columns can be encrypted like any other column data type. However, due to their large size, blob encryption can be time-consuming. Typically, a large blob is created before its creator knows which column it will belong to. If the final column destination is encrypted, then the unencrypted blob will need to be re-read and encrypted with the column's encryption key.

To avoid blob re-encryption overhead, two blob parameter items have been added, and can be passed to **isc_blob_create2()** to indicate the column to which the blob will be assigned. The items **isc_bpb_target_relation_name** and **isc_bpb_target_field_name** denote the column to which the blob will be assigned by the developer. These items are passed via the blob parameter block in the same way that blob filter and character set blob parameter items are sent. The blob parameter byte string includes the following:

- The blob parameter;
- One "length" byte; and
- "Length" bytes for the target name.

isc_blob_gen_bpb() and isc_blob_gen_bpb2() can generate these new blob parameter items if the target blob descriptor argument has both blob_desc_relation_name and blob_desc_field_name string members.

If a blob ID is assigned between two columns with different encryptions, the blob assigned to the destination column is automatically translated between the two encryptions. This means that the source blob is decrypted internally to plaintext and the destination blob is encrypted with the new ciphertext.

The workaround described here also pertains to special cases in which one of the blobs is not encrypted. If an encrypted blob ID is assigned to a blob column with no encryption, the assignment is allowed but a warning error is returned.

Decrypting Data

Only the database owner can perform database-level decryption. Decrypting a database causes all pages to be decrypted and rewritten in plaintext.

To decrypt a database, use the following syntax:

```
alter database decrypt
```

An **isc_database_info()call** can be made to determine if database-level encryption is enabled, by passing an **isc_info_db_encrypted** info item. A value of 1 is returned if the database is encrypted and a value of 0 if not. GSTAT indicates the database is encrypted in the Variable header data section of the header page display and isql does likewise with the Show Database command.

Decrypting Columns

A column can be re-encrypted with another key or decrypted. The table needs exclusive access before this operation can proceed. All rows in the table are re-encrypted and the former column data, including blobs, are zeroed from the database so that it is no longer visible. If more than a single column in a table is altered

for a change in encryption, you should disable auto-commit of DDL statements. This allows the multiple columns to be re-encrypted in a single pass over the table, which can save time on very large tables.

To decrypt a column, use the following syntax:

```
alter table table-name alter [column] column-name decrypt
```

Granting Decrypt Permission

After encrypting a column, the database owner or the individual table owner, grants **decrypt** permission to users who need to access the values in an encrypted column. Generally speaking, these are end users who already have, or who need to have, select, insert, update, and/or delete privileges on the same data. You can grant decrypt permission to individual users and to groups of users by role, view, trigger, and stored procedure.

To grant decrypt privileges to an individual user, use the following syntax:

```
grant decrypt[(column-name, ...)] on table-name to user-name
```

NOTE

If the database owner or the individual table owner has explicitly granted execute and select privileges to users on stored procedures and views, respectively, a chain of ownership implicitly grants decrypt privilege on any referenced encrypted columns in those schema elements owned by that schema owner.

Permissions for Roles and Views

When a number of users need to access the same encrypted columns, you can save time and effort by assigning the users to the same role, and granting decrypt permission to the role rather than to each individual user.

For example, suppose you have a table called "Employee" which contains columns that are used by people in the same department. You could create a role called "HR_Role," assign individual HR employees to the role, and then grant decrypt privileges to the role. The code sample that follows shows you how to create users, assign them to a role, then provide decrypt privileges to the role:

```
CREATE USER J_Smith PASSWORD 'Smith'
CREATE USER J_Doe PASSWORD 'Doe'
CREATE USER B_Jones PASSWORD 'Jones'
CREATE ROLE HR_Role
GRANT HR_Role to J_Smith, J_Doe, B_Jones
GRANT DECRYPT (A) on Employee to HR_Role
GRANT DECRYPT (B) on Employee to HR_Role
```

After issuing these commands, all the members in the HR_role can use their role affiliation to decrypt columns A and B.

Similarly, you can give users access to a view that has decrypt access to encrypted columns. First you create the view:

```
CREATE VIEW Payroll_View as SELECT
C, D, E, I FROM Employee
```

Payroll_View now contains data from columns C, D, E, and I. Next, you can grant decrypt access to encrypted columns on Employee to view Payroll_View:

```
GRANT DECRYPT (C, D, E, I) ON Employee TO VIEW Payroll_View
```

Next, you can grant access to Payroll_View to individual users:

```
GRANT SELECT ON Payroll_View TO D_Gibson
```

or to all the users assigned to a role (after creating the role), as shown below:

```
GRANT SELECT ON Payroll_View TO Payroll_Role
```

Revoking Encrypt and Decrypt Permissions

There are two revoke commands associated with the InterBase encryption feature:

- **Revoke ENCRYPT ON ENCRYPTION** is used to revoke encryption permission. Only the SYSDSO can revoke encryption permission.
- **Revoke DECRYPT** can be used by the database or table owner to revoke decrypt permission from a user, role, or view.

To revoke encryption permission, the SYSDSO uses the following syntax:

```
revoke ENCRYPT ON ENCRYPTION key-name from user-name;
```

To revoke decrypt permission, the database or table owner uses the following syntax:

```
revoke decrypt[(column-name, ...)] on table-name from {user-name {{!}} role- name {{!}} public}
```

Encrypting a Database with IBConsole

This section explains how to enable Embedded User Authentication (EUA) and perform encryption using IBConsole. Before enabling and performing encryption, please read About InterBase Encryption.

For more information about using IBConsole, see the InterBase IBConsole.

Enabling EUA and Performing Encryption When Creating a New Database

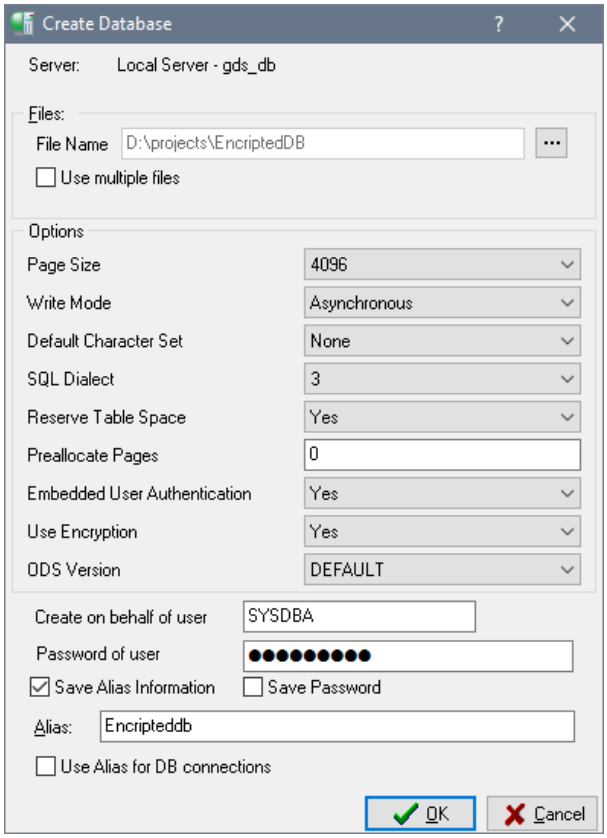
To enable EUA and perform encryption when creating a new database, do the following:

1. Login to IBConsole as a SYSDBA or as a database owner.
2. Click **Databases > Create Database** from the menu.
3. In **Create Database**, click the browse button in the **File** field and the **Specify Database Name** dialog opens.
4. In the **Save In** field, select the folder where you want to save the database.
5. Specify a file name and click **Save** and the dialog closes.

NOTE

The **Alias** field is automatically populated with the database name you just created in the File Name field.

6. Change the value in the **Embedded User Authentication** field to **Yes**. The **Use Encryption** field is now visible.
7. In the **Use Encryption** field, change the value to **Yes**, as shown in the figure:



8. Click **OK** to create the database.

NOTE

You can also specify if the database is created for another user. In the **Alias** field, you can specify the name of the user. In the **Create on behalf of user** you would specify the name of the user.

9. Click **OK** to create the database.The dialog closes and the **Database Connect** dialog opens.
10. Enter your connection information and click **Connect**.
11. The database is created and the Encryption Wizard opens.

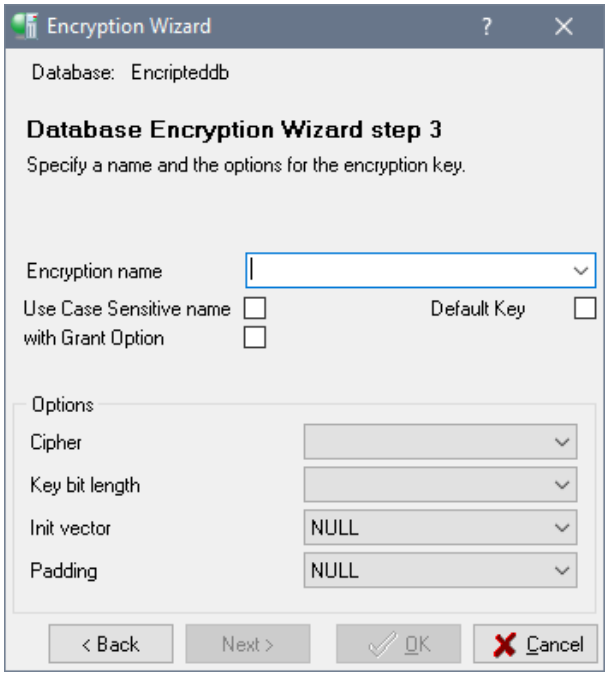
12. After reading the page, click Next.
13. In Step 1, type the SYSDSO **Password** and click **Next**.

14. In Step 2, create and confirm a System Encryption Password (SEP) and then click **Next**.

NOTE

For more information about **External**, see Using the External Option when Creating a SEP.

15. In Step 3, type a name for the Encryption Key in the **Encryption Name** field. Change the fields in the **Options** section as desired.



- You can specify any name for the encryption key.
- The encryption key used to encrypt a backup needs to be password-protected. You can create a key without a password, but that will cause problems if that key is used at the time of backup.
- It is recommended that you select the **Default Key** check-box. However it is not required.
- In this example **DES** is used as Cipher because it is free.

16. In Step 4, create a password-protected key and then click **OK**.

The dialog closes and a message opens informing you that the encryption of database is complete.

NOTE

For more information about Embedded User Authentication, see the InterBase Operations Guide.

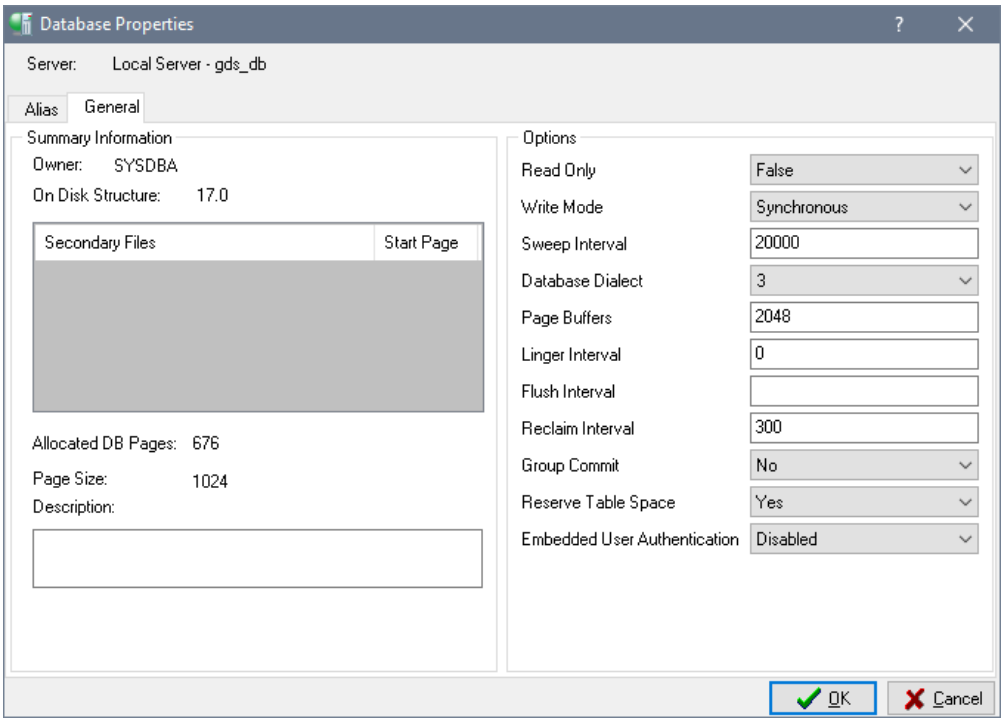
Enabling EUA and Performing Encryption on an Existing Database

NOTE

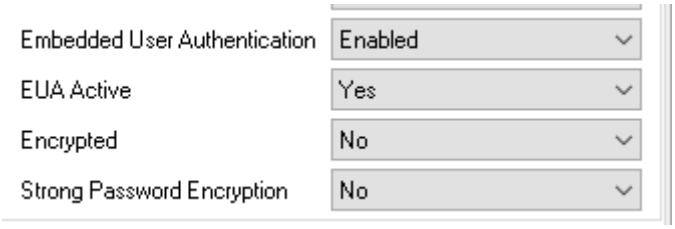
The encryption feature requires **ODS 13** and is not available on older ODS databases. Therefore, a backup and restore to ODS 13 is required for pre-existing databases to use InterBase encryption. For information about performing backups and restores, see the InterBase Operations Guide.

To enable EUA and perform encryption on an existing database, take the following steps:

1. Connect to the database you want to encrypt as the owner.
2. In the left pane, right-click on the database name and select **Properties** from the context menu or select **Database > Properties** from the menu.
3. In the **Database Properties** dialog, click on the **General** tab.



4. In the **Embedded User Authentication** field, change the value to **Enabled**. Notice that the dialog now shows three new options.



- You do not have to change the setting in the **EUA Active** field because the default setting is Yes.
- In the **Encrypted** field select **Yes** from the drop-down menu.
- You do not need to change the **Strong Password Encryption** setting.

5. Click **OK** to enable EUA. The dialog closes and the Encryption Wizard opens.

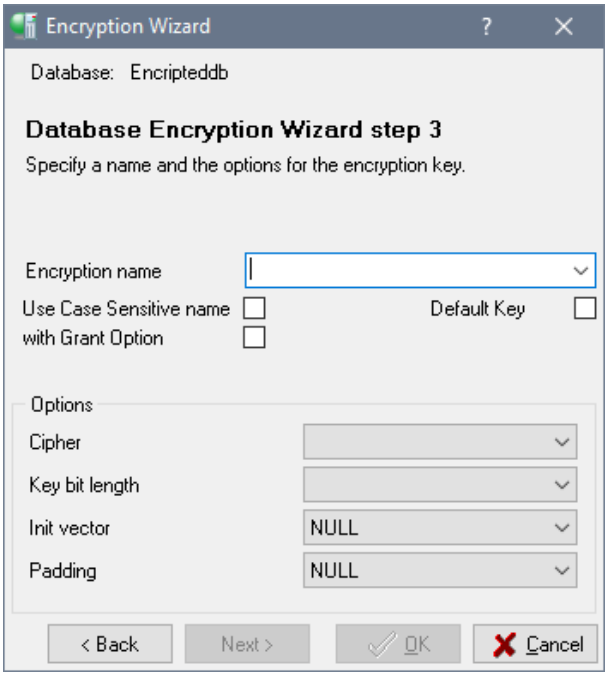
6. After reading the page, click **Next**.
7. In Step 1, type the SYSDSO **Password** and click **Next**.

8. In Step 2, create and confirm a System Encryption Password. Then click **Next**.

NOTE

For more information about **External**, see Using the External Option when Creating a SEP.

9. In Step 3, type a name for the Encryption Key in the **Encryption Name** field. Change the fields in the **Options** section as desired.



- You can specify any name for the encryption key.
- The encryption key used to encrypt a backup needs to be password protected. You can create a key without a password, but that will cause problems if that key is used at the time of backup.
- It is recommend that you select the **Default Key** check box. However it is not required.
- In this example **DES** is used as Cipher because it is free.

10. In Step 4, create a password protected key and then click **OK**.

The dialog closes and a message opens informing you that the encryption of database is complete.

NOTE

For more information about Embedded User Authentication, see the InterBase Operations Guide.

Decrypting the Database

Database-level decryption causes all pages to be decrypted and rewritten in plain text. To **decrypt the database** using IBConsole:

1. In the left pane, right-click on the database name, and select **Decrypt Database**. This decrypts the database.
2. A dialog appears asking you to confirm your decision.

Database-level decryption causes all pages to be decrypted and rewritten in plaintext.

NOTE

Notice that the command in the right pane has been changed to **Encrypt Database.**

Performing Column-level Encryption Using IBConsole

With column-level encryption, only the data of the selected columns is encrypted. Also, with column-level encryption the database/table owner can specify ***GRANT/REVOKE*** access to certain roles, users, stored procedures, and triggers.

After enabling EUA, login as SYSDSO and take the following steps in IBConsole:

1. Right-click on the database that contains the columns you want to encrypt, and choose **Set SEP**.
2. On **Set System Encryption Password**, enter and confirm a password, then choose **OK**.
3. Select the **Encryptions** node, then right-click in the right pane and select **Create**.

4. In **Encryption Editor** complete the fields as desired and choose **OK**.

   - Name: Enter your name of the encryption key. You have an option to make this case sensitive or not.
   - Description: Enter a description to define your encryption key.
   - Cipher: Select from the drop-down list: None, AES, DES.
   - Key Length: Specifies the bit length of the encryption key. For DES it is always 56. For AES you can select from 128, 192, or 256, with 128 as the default value. The higher the bit length, the stronger the encryption.
   - Init Vector: Select NULL (the default) or RANDOM. This specifies the initialization vector.
   - Pad: Select NULL (the default) or RANDOM. Random provides stronger encryption.
   - Password and Confirm Password: Enter and confirm the password for this encryption key.
   - Grant Owner: This is selected by default. If this is not selected the key cannot be used until you GRANT permission to someone using the Grant Editor.

5. Disconnect from the database as SYSDSO, and reconnect as SYSDBA.
6. Select the **Tables** node, right-click the table that contains the columns that you want to encrypt, and select **Properties**.
7. Click and the Table Editor opens.
8. Select the name of the column to encrypt, and click **Edit Field**.
9. In **Field Property Editor**, complete the following information, and click **OK**.

   - Name: Where the name of the column is entered or changed.
   - Field Kind: Select the kind of column to create. Once you make a selection three options are available:
   - Domain: You have two options: (1) Select Domain where you select an existing domain to be used with the column; or (2) New Domain which opens the Domain Editor to create a new domain to be used with the column.
   - Data Type: You can open the Data Type Editor to create or alter the Data Type definition for the column.
   - Computed By: This field is only visible when created a Computed By column. Computed By columns cannot be altered.
     - Domain: Displays the domain name if the column definition is based on a domain.
     - Data Type: The data type definition is displayed, whether it is based on a domain or created manually.
     - Default: This field is only enabled when creating a column that is not Computed By.
     - Not Null: When checked the column cannot have null values. This is only enabled when creating a column definition.
     - Encryption: The column is encrypted with the selected encryption key. If “none” is selected the column will decrypt.
     - Decrypt Default: This value is displayed to users who are not granted permission to see or alter the data in the column. If no “Decrypt default: is specified, IBConsole hides the column when a user without rights displays the data in the table.

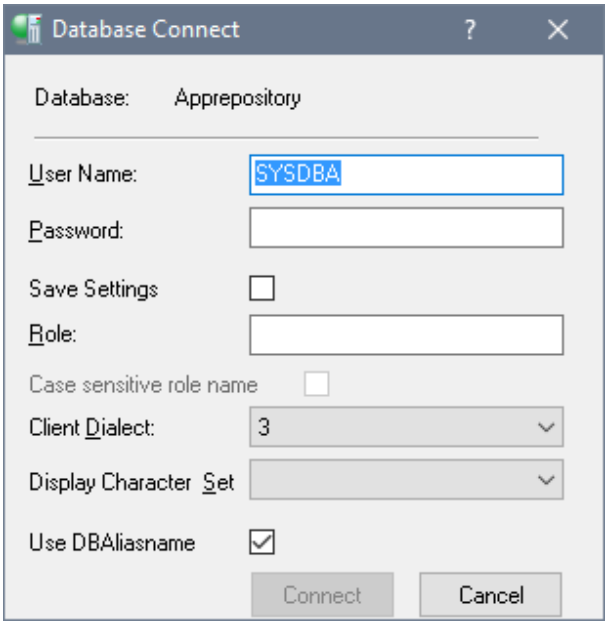
10. Choose **OK** again on the Table Editor. The column you selected is encrypted using the encryption level you specified.

Backup and Restore an Encrypted Database

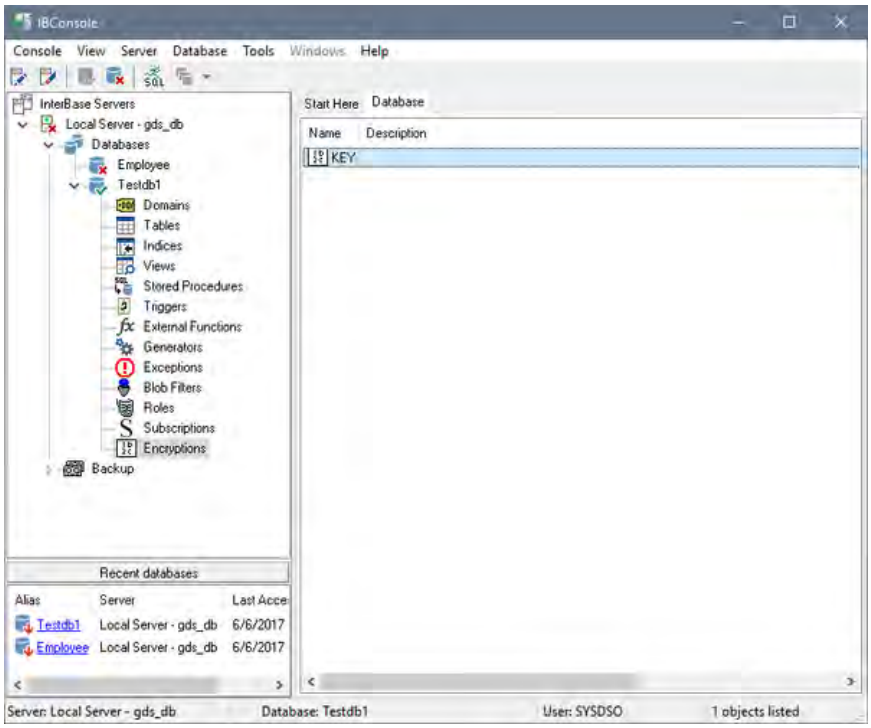
To backup and restore an encrypted database the database must have a special password-protected encryption key. Any password-protected encryption key will work, but it is a good practice to create a special key for this purpose. Only SYSDSO can create encryption keys.

Create an Encrypted Key

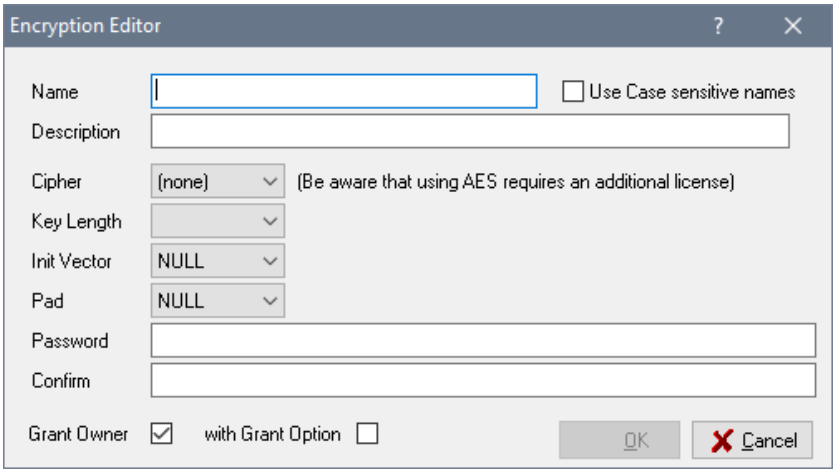
1. If you are connected to the database then you must first disconnect from it.
2. In the right pane double-click **Connect As**, or select **Database > Connect As** from the menu. The **Connect As** dialog will be displayed.



3. Specify SYSDSO as the user and supply the SYSDSO password that you created in the Encryption Wizard.
4. Click Connect and the database is connected. Notice that in the node, Encryptions is now visible.



5. Select this node, then right-click in the right pane and select **Create** from the pop-up menu. The **Encryption Editor** opens.



6. Specify a name for the encryption key, select a Cipher, and enter and confirm a password.
7. Click **OK** and the backup key is created. Note that the key you just created is now listed in the right panel.
8. Now disconnect from the database.

Backup an Encrypted Database

1. Connect to the database as owner of the database.
2. Double-click on **Database Backup** in the right pane or select **Database > Maintenance > Backup/Restore > Backup** from the menu. The Database Backup dialog is displayed.
3. IBConsole supplies a default filename for the backup file. To override this click the browse button to the right of the **File Name** input box. Here you can select another backup file or create a new file name.

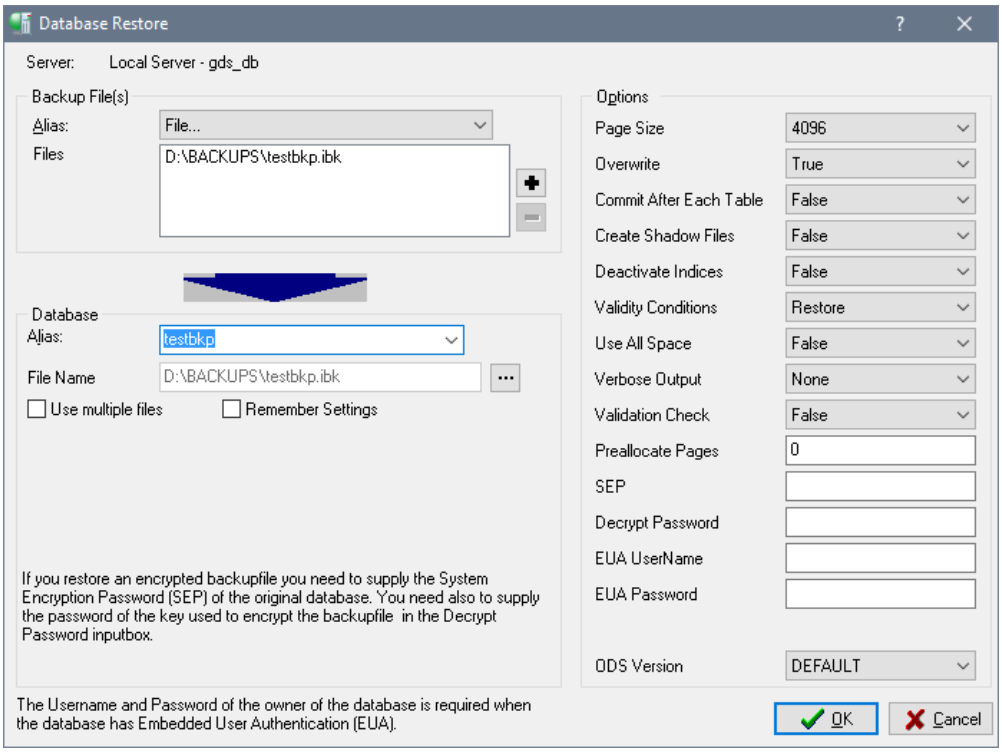
4. In the Options group box select the **Encryption Key** from the dropdown box. IBConsole lists all available password protected encryption keys.
5. IBConsole already has the SEP so you do not need to specify this.
6. Click **OK** and a the cursor changes to a "processing" cursor. Once the backup is completed, an information dialog tells you the database backup is complete.

Restore an Encrypted Database

To restore a database logon as SYSDBA or as the owner of the database, make sure you are not connected to the database, then do the following steps:

1. Right-click the **Backup** node in the left pane and select **Restore** or select **Database > Maintenance > Backup/Restore > Restore** from the menu. The **Database Restore** dialog opens.
2. First, specify the backup file to be restored. In the Backup File(s) option, select **File...** from the Alias field.
3. In the File Open dialog browser to your backup file and click **Open**.
4. In the Database option, specify the database to restore to. Select a database alias from the **Alias** drop down or specify a file.
5. You need to set the following options:

- If you are restoring to the original database file name, you need to set **Overwrite** to **True**.
- In **SEP** you must specify the SEP of the backup file. At this point IBConsole does not make any assumption about the SEP of a backup file. So you must specify the SEP you used when you encrypted the database.
- In the Decrypt Password field, supply the password of the encryption key used to create the backup file.
- Specify the name of the database owner and password in the **EUA User Name** and **EUA Password** fields.



6. Click **OK** to start the restore. Once the restore is complete, you will get a completion message.

Encrypting Backup Files

Because backup files are often sent off-site for disaster recovery or long-term archival purposes, it is important that encrypted databases have their backup files encrypted as well.

A database encrypted at the database or column level must be backed up as encrypted. There is no override or "backdoor" to back up an encrypted database in unencrypted form. To allow a regular, unencrypted database backup, the user would have to manually make a file copy of the database and alter the database copy to decrypt all pages and columns and drop all encryptions. The GBAK utility provides three additional switches to facilitate encrypt and decrypt operations on database backups.

The GBAK utility uses the ***-encrypt*** and ***-decrypt*** switches to provide the information required to encrypt and decrypt a database backup. The **-sep** switch is used to pass the system encryption password of the database that is being backed up and restored. If the ***-sep*** switch is not provided, InterBase automatically provides the value associated with the ISC_SYSTEM_ENCRYPT_PASSWORD environment variable (when the variable has been defined).

IMPORTANT

Starting with InterBase XE, all encrypted databases (AES or DES) can only be backed up or restored using the "-se service" option. Only service-side backups are allowed for encrypted databases thus making data visibility more secure; this inhibits any potential process-space view of unencrypted data on a normal backup client.

Avoiding Embedded Spaces in GBAK Encrypt/Decrypt and Sep Statements

When using the -sep (for creating the System Encryption Password), ***-encrypt***, ***-decrypt*** arguments, it is recommended that you avoid using delimited identifiers and password arguments with embedded spaces in the argument, if possible.

The ***-sep***, ***-encrypt***, ***-decrypt*** arguments require quotations if they contain embedded spaces. If quotations are required, the quotation nesting level depends on whether GBAK is invoked with the -service switch. If the -service switch is not given, then one level of quotation is satisfactory. If the -service switch is given, then two levels of quotation are required.

Encrypting a Database Backup File

The **-encrypt** switch requires the name of the encryption defined in the database that is being backed up.

The following example shows how to use the -encrypt and -sep switches to encrypt a sample database backup file:

```
gbak -b c:\embarcadero\interbase\examples\database\employee.ib employee.ibak -sep 'sep password'
-encrypt my_backup_key
```

Decrypting a Database Backup File During a Restore

The ***-decrypt*** switch is used during the database restore process to provide the password of the encryption that was used to originally backup the database.

The following example shows how to use the ***-decrypt*** and ***-sep*** switches to decrypt a sample database file during a database restore**:**

```
gbak -r employee.ibak c:\embarcadero\interbase\examples\database\employee.ib -sep "'sep
password"' -decrypt 'my password'
```

For more information about using the **GBAK -b** and **-r** options to perform database backups and restores, see the InterBase Operations Guide.

Additional Guidelines for Encrypting and Decrypting Database Backup Files

When preparing to encrypt or decrypt database backup files, keep the following information in mind:

- The encryption chosen for a database backup must be custom password-protected and at least as strong, in terms of encryption key size, as the strongest encryption defined in the database.
- An encrypted database backup file will be almost the same size as an unencrypted database backup. However, the time to encrypt and decrypt a backup file may be longer than a backup which is not encrypted.

- GBAK retrieves all encrypted column data in plaintext form, so Over-the-Wire (OTW) encryption should be used if backing up and restoring over the network. Alternatively, the -se service manager switch can be used to backup and restore on the server to avoid network transmission. For more information about OTW, see the InterBase Operations Guide.
- It is the user's responsibility to remember the encryption password and system encryption password necessary to decrypt a set of database backup files as there is no means for InterBase to do so automatically.
- Databases with AES encryption keys allow backup/restore activities only as a service. It was designed to facilitate discovering a "strong encryption" license mandate in the engine. Databases with DES encryption keys are allowed to be backed up (and restored) as a pure client. However, the encrypted data, now decrypted with proper authentication, could be visible in transit in the process space of GBAK (or other backup applications).

Backup/restore operations can be restricted on any encrypted database, whether DES or AES strength. An error message now displays the following when GBAK is not run as a service on encrypted databases:

```
gbak: ERROR: encrypted database: use -service switch
```

Character Sets and Collation Orders

This chapter discusses the following topics:

- Available character sets and their corresponding collation orders
- Character set storage requirements
- Specifying default character set for an entire database
- Specifying an alternative character set for a particular column in a table
- Specifying a client application character set that the server should use when translating data between itself and the client
- Specifying the collation order for a column
- Specifying the collation order for a value in a comparison operation
- Specifying the collation order in an ***ORDER BY*** clause
- Specifying the collation order in a ***GROUP BY*** clause

About 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. InterBase supports four different types of collation order: Windows, dBASE, Paradox, and ISO. The ISO collation sequence is recommended in preference to the other three.

Character Set Storage Requirements

It is important to know the storage requirements of a particular character set because InterBase restricts the maximum amount of storage in each field of a CHAR column to 32,767 bytes. VARCHAR columns are restricted to 32,765 bytes.

For character sets that require only a single byte of storage per character, the maximum number of characters that can be stored in a single field corresponds to the number of bytes. For character sets that require multiple bytes per character, determine the maximum number of symbols that can be safely stored in a field by dividing 32,767 or 32,765 by the number of bytes required for each character.

For example, for a CHAR column defined 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;
```

InterBase Character Sets

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 implicit collation of the set, the one that is used if no ***COLLATE*** clause specifies an alternative order. The implicit collation order cannot be specified in the ***COLLATE*** clause.

Collation names of the form WINxxxx are defined by Microsoft, those of the form DB_xxx are dBASE, and those that start with PDOX or PXW are Paradox. Collation names of the form AA-BB are ISO collations: AA is the language, BB is the country.

Character sets and collation orders				
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
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

Character sets and collation orders				
Character set	**Character set ID**	**Maximum character size**	**Minimum character size**	**Collation orders**
EUCJ_0208	6	2 bytes	1 byte	EUJC_0208
GB_2312	57	2 bytes	1 byte	GB_2312
ISO8859_1	21	1 byte	1 byte	ISO8859_1 CC_PTBRLAT1 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
ISO8859_15	39	1 byte	1 byte	ISO8859_15 CC_PTBRLAT9 DA_DA9 DE_DE9 DU_NL9 EN_UK9 EN_US9 ES_ES9 FI_FI9 FR_CA9 FR_FR9 IS_IS9

Character sets and collation orders				
Character set	**Character set ID**	**Maximum character size**	**Minimum character size**	**Collation orders**
				IT_IT9 NO_NO9 PT_PT9 SV_SV9
KOI8R	58	1 byte	1 byte	KOI8R RU_RU
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
UNICODE_BE UCS2BE	8	2 bytes	2 bytes	N/A at this time
UNICODE_LE UCS2LE	64	2 byte	2 bytes	N/A
WIN1250	51	1 byte	1 byte	WIN1250 PXW_CSY PXW_HUNDC PXW_PLK PXW_SLOV
WIN1251	52	1 byte	1 byte	WIN1251 PXW_CYRL
WIN1252	53	1 byte	1 byte	WIN1252 CC_PTBRWIN 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

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 sets corresponding to DOS code pages	
Character set	**DOS code page**
DOS437	437
DOS850	850
DOS852	852
DOS857	857
DOS860	860
DOS861	861
DOS863	863
DOS865	865

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.

UNICODE BE and UNICODE LE Character Sets

InterBase now supports 16-bit UNICODE_BE and UNICODE_LE as server character sets. These character sets cannot be used as client character sets. If your client needs full UNICODE character support, please use UTF8 instead of UNICODE_LE and UNICODE_BE for the client character set (a.k.a LC_CSET). A client can use the UTF8 (or other native) client character set to connect with a UNICODE database.

A database schema is declared to use the new character set in the ***CREATE DATABASE*** statement, as follows:

```
CREATE DATABASE <filespec> <...> DEFAULT CHARACTER SET UNICODE;
```

Note that InterBase uses "big endian" ordering by default.

The attributes for the UNICODE_BE and UNICODE_LE character sets are shown in InterBase Character Sets.

NOTE

InterBase 2008 does not support UNICODE collations in this release. The default collation is binary sort order for UNICODE.

Support for the UTF-8 Character Set

The UTF-8 character set is an alternative coded representation form for all of the characters of the ISO/IEC 10646 standard. To use the UTF-8 character set, you would declare a database schema to use the character set, in the ***CREATE DATABASE*** SQL statement, as shown below:

```
CREATE DATABASE <filespec> <...> DEFAULT CHARACTER SET UTF8;
```

Additionally, you may use the alias UTF_8.

The attributes for the UTF-8 character set are shown in InterBase Character Sets.

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, comparisons, ***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.ib' 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 Language Reference Guide.

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 data type 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.ib' 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 Language Reference Guide.

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 in Specifying a Character Set for a Column in a Table:

```
SET NAMES DOS437;
CONNECT 'europe.ib' USER 'JAMES' PASSWORD 'U4EEAH';
```

For the complete syntax of ***SET NAMES*** and ***CONNECT***, see Language Reference Guide.

Specifying Collation Orders

This section describes how to use the ***COLLATE*** clause to specify collation order in columns, comparison operations, ***ORDER BY*** clauses, and ***GROUP BY*** clauses.

Specifying Collation Order for a Column

Use the ***COLLATE*** clause with either ***CREATE TABLE*** or ***ALTER TABLE*** to specify the collation order for a CHAR or VARCHAR column. The ***COLLATE*** clause 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 Language Reference Guide.

Specifying Collation Order in a Comparison Operation

When CHAR or VARCHAR values are compared in a ***WHERE*** clause, it is 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 Language Reference Guide.

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 Guide.

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 Language Reference Guide.