Language Reference

Version 4.0

Borland®
InterBase®

Borland International, Inc., 100 Borland Way
P.O. Box 660001, Scotts Valley, CA 95067-0001

Printed in the U.S.A.

INT1340WW21773 1E0R1295
9596979899-9 8 7 6 5 4 3 2 1
H1

Table of Contents

Tables and Figures

Preface

This preface describes the documentation set, the printing conventions used to display information in text and in code examples, and the conventions a user should employ when specifying database objects and files by name in applications.

The InterBase Documentation Set

The InterBase documentation set is an integrated package designed for all levels of users. The InterBase server documentation consists of a five-book core set and a platform-specific installation guide. Information on the InterBase Client for Windows is provided in a single book.

The InterBase core documentation set consists of the following books:

Table 1: InterBase Core Documentation

Book	Description
Getting Started	Provides a basic introduction to InterBase and roadmap for using the documentation and a tutorial for learning basic SQL through **isql**. Introduces more advanced topics such as creating stored procedures and triggers.
Data Definition Guide	Explains how to create, alter, and delete database objects through **isql**.
Language Reference	Describes SQL and DSQL syntax and usage.
Programmer's Guide	Describes how to write embedded SQL and DSQL database applications in a host language, precompiled through **gpre**.
API Guide	Explains how to write database applications using the InterBase API.
Installing and Running on . . .	Platform-specific information on installing and running InterBase.

Additional documentation includes the following book:

Table 2: InterBase Client Documentation

Book	Description
InterBase Windows Client User's Guide	Installing and using the InterBase PC client. Using Windows **isql** and the InterBase Server Manager.

Printing Conventions

The InterBase documentation set uses different fonts to distinguish various kinds of text and syntax.

Text Conventions

The following table describes font conventions used in text, and provides examples of their use:

Table 3: Text Conventions

Convention	Purpose	Example
UPPERCASE	SQL keywords, names of all database objects such as tables, columns, indexes, stored procedures, and SQL functions.	The following SELECT statement retrieves data from the CITY column in the CITIES table.
italic	Introduces new terms, and emphasizes words. Also used for file names and host-language variables.	The *isc4.gdb* security database is *not* accessible without a valid *username* and *password*.
bold	Utility names, user-defined and host-language function names. Function names are always followed by parentheses to distinguish them from utility names.	To back up and restore a database, use **gbak** or the server manager. The **datediff()** function can be used to calculate the number of days between two dates.

Syntax Conventions

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

Table 4: Syntax Conventions

Convention	Purpose	Example
UPPERCASE	Keywords that must be typed exactly as they appear when used.	SET TERM !!;
italic	Parameters that *cannot* be broken into smaller units. For example, a table name cannot be subdivided.	CREATE TABLE *name* (*<col>* [**,** *<col>* ...]);
<italic>	Parameters in angle brackets that *can* be broken into smaller syntactic units.	CREATE TABLE *name* (*<col>* [**,** *<col>* ...])**;**
	For example, column definitions (*<col>*) can be subdivided into a name, data type and constraint definition.	*<col> = name <datatype>* [CONSTRAINT *name <type>*]
[ ]	Square brackets enclose optional syntax.	*<col>* [**,** *<col>* ...]
...	Closely spaced ellipses indicate that a clause within brackets can be repeated as many times as necessary.	(*<col>* [**,** *<col>* ...]);
|	The pipe symbol indicates that either of two syntax clauses that it separates may be used, but not both.	SET TRANSACTION {SNAPSHOT [TABLE STABILITY] | READ COMMITTED};
	Inside curly braces, the pipe symbol separates multiple choices, one of which *must* be used.	
{ }	Curly braces indicate that one of the enclosed options *must* be included in actual statement use.	SET TRANSACTION {SNAPSHOT [TABLE STABILITY] | READ COMMITTED};

Database Object-naming Conventions

InterBase database objects, such as tables, views, and column names, appear in text and code in uppercase in the InterBase documentation set because this is the way such information is stored in a database's system tables.

When an applications programmer or end user creates a database object or refers to it by name, case is unimportant. The following limitations on naming database objects must be observed:

- Start each name with an alphabetic character (A-Z or a-z).
- Restrict object names to 31 characters, including dollar signs ($), underscores (_), 0 to 9, A to Z, and a to z. Some objects, such as constraint names, are restricted to 27 bytes in length.
- Keep object names unique. In all cases, objects of the same type, for example, tables and views, *must* be unique. In most cases, object names must also be unique within the database.

File-naming Conventions

InterBase is available on a wide variety of platforms. In most cases users in a heterogenous networking environment can access their InterBase database files regardless of platform differences between client and server machines if they know the target platform's file naming conventions.

Because file-naming conventions differ widely from platform to platform, and because the core InterBase documentation set is the same for each of these platforms, all file names in text and in examples are restricted to a base name with a maximum of eight characters, with a maximum extension length of three characters. For example, the example database on all servers is referred to as *employee.gdb*.

Generally, InterBase fully supports each platform's file-naming conventions, including the use of node and path names. InterBase, however, recognizes two categories of file specification in commands and statements that accept more than one file name. The first file specification is called the *primary file specification*. Subsequent file specifications are called *secondary file specifications*. Some commands and statements place restrictions on using node names with secondary file specifications.

In syntax, file specification is denoted as follows:

```
"<filespec>"
```

Primary File Specifications

InterBase syntax always supports a full file specification, including optional node name and full path, for primary file specifications. For example, the syntax notation for CREATE DATABASE appears as follows:

```
CREATE {DATABASE | SCHEMA} "<filespec>"
   [USER "username" [PASSWORD "password"]]
   [PAGE_SIZE [=] int]
   [LENGTH [=] int [PAGE[S]]]
   [DEFAULT CHARACTER SET charset]
   . . .
```

In this syntax, the <*filespec*> that follows CREATE DATABASE supports a node name and path specification, including a platform-specific drive or volume specification.

Secondary File Specifications

For InterBase syntax that supports multiple file specification, such as CREATE DATABASE, all file specifications after the first are secondary. Secondary file specifications generally cannot include a node name, but may specify a full path name. For example, the syntax notation for CREATE DATABASE appears as follows:

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

<secondary_file> = FILE "<filespec>" [<fileinfo>] [<secondary_file>]

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

In the secondary file specification, <*filespec*> does not support specification of a node name.

CHAPTER 1

Introduction

The InterBase *Language Reference* details the syntax and usage of SQL and Dynamic SQL (DSQL) statements for embedded applications programming, and for the InterBase interactive SQL utility, **isql**. It describes additional language and syntax specific to InterBase stored procedures and triggers.

Who Should Use This Book

The *Language Reference* assumes a general familiarity with SQL, data definition, data manipulation, and programming practice. It is a syntax and usage resource for:

- Programmers writing embedded SQL and DSQL database applications.
- Programmers writing directly to the InterBase applications programming interface (API), who need to know supported SQL syntax.
- Database designers who create and maintain databases and tables with **isql**.
- Users who perform queries and data manipulation operations through **isql**.

Topics Covered in This Book

The following table lists the chapters in the *Language Reference*, and provides a brief description of them:

Table 1-1: *Language Reference* Chapters

Chapter	Description
Chapter 1: Introduction	Introduces the book, and describes its intended audience.
Chapter 2: SQL Statement and Function Reference	Provides syntax and usage information for SQL and DSQL statements.
Chapter 3: Procedure and Trigger Language Reference	Describes syntax and usage information for stored procedure and trigger language.
Appendix A: Reserved Words	Lists keywords, symbols, and punctuation, that have special meaning to InterBase.
Appendix B: Error Codes and Messages	Summarizes InterBase error messages and error codes.
Appendix C: System Tables and Views	Describes InterBase system tables and views that track meta-data.
Appendix D: Character Sets and Collation Orders	Explains all about character sets and corresponding collation orders for a variety of environments and uses.

CHAPTER 2

SQL Statement and Function Reference

This chapter provides the syntax and usage for each InterBase SQL statement. It includes the following topics:

- Statement List
- Function List
- Data Types
- Error Handling
- Using Statement and Function Definitions

Statement List

The following table lists the SQL statements described in this chapter:

Table 2-1: SQL Statements

ALTER DATABASE	ALTER DOMAIN	ALTER EXCEPTION
ALTER INDEX	ALTER PROCEDURE	ALTER TABLE
ALTER TRIGGER	BASED ON	BEGIN DECLARE SECTION
CLOSE	CLOSE (BLOB)	COMMIT
CONNECT	CREATE DATABASE	CREATE DOMAIN
CREATE EXCEPTION	CREATE GENERATOR	CREATE INDEX
CREATE PROCEDURE	CREATE SHADOW	CREATE TABLE
CREATE TRIGGER	CREATE VIEW	DECLARE CURSOR
DECLARE CURSOR (BLOB)	DECLARE EXTERNAL FUNCTION	DECLARE FILTER
DECLARE STATEMENT	DECLARE TABLE	DELETE

Table 2-1: SQL Statements (Continued)

DESCRIBE	DISCONNECT	DROP DATABASE
DROP DOMAIN	DROP EXCEPTION	DROP EXTERNAL FUNCTION
DROP FILTER	DROP INDEX	DROP PROCEDURE
DROP SHADOW	DROP TABLE	DROP TRIGGER
DROP VIEW	END DECLARE SECTION	EVENT INIT
EVENT WAIT	EXECUTE	EXECUTE IMMEDIATE
EXECUTE PROCEDURE	FETCH	FETCH (BLOB)
GRANT	INSERT	INSERT CURSOR (BLOB)
OPEN	OPEN (BLOB)	PREPARE
REVOKE	ROLLBACK	SELECT
SET DATABASE	SET GENERATOR	SET NAMES
SET STATISTICS	SET TRANSACTION	UPDATE
WHENEVER		

Function List

The following table lists the SQL functions described in this chapter:

Table 2-2: SQL Functions

Function	Type	Purpose
AVG()	Aggregate	Calculates the average of a set of values.
CAST()	Conversion	Converts a column from one data type to another.
COUNT()	Aggregate	Returns the number of rows that satisfy a query's search condition.
GEN_ID()	Numeric	Returns a system-generated value.
MAX()	Aggregate	Retrieves the maximum value from a set of values.
MIN()	Aggregate	Retrieves the minimum value from a set of values.
SUM()	Aggregate	Totals the values in a set of numeric values.
UPPER()	Conversion	Converts a string to all uppercase.

Aggregate functions perform calculations over a series of values, such as the columns retrieved with a SELECT statement.

Conversion functions transform data types, either converting them from one type to another, or by converting CHARACTER data types to all uppercase.

The numeric function, GEN_ID(), produces a system-generated number that can be inserted into a column requiring a numeric data type.

Data Types

InterBase supports most SQL data types, but does not directly support the SQL DATE, TIME, and TIMESTAMP data types. In addition to standard SQL data types, InterBase also supports binary large object (BLOB) data types, and arrays of data types (except for BLOB data).

The following table lists the data types available to SQL statements in InterBase:

Table 2-3: Data Types Supported by InterBase 4.0

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

Table 2-3: Data Types Supported by InterBase 4.0 (Continued)

Name	Size	Range/Precision	Description
NUMERIC (*precision*, *scale*)	variable	*precision* = 1 to 15. Specifies exactly *precision* digits of precision to store. *scale* = 1 to 15. Specifies number of decimal places for storage. Must be less than or equal to *precision*.	Number with a decimal point *scale* digits from the right. For example, NUMERIC(10,3) holds numbers accurately in the following format: ppppppp.sss
SMALLINT	16 bits	-32768 to 32767.	Signed short (word).
VARCHAR(*n*)	*n* characters	1 to 32765 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.

‡ **Actual size of DOUBLE is platform dependent. Most platforms support the 64 bit size.**

Error Handling

Every time an executable SQL statement is executed, the SQLCODE variable is set to indicate its success or failure. SQLCODE is not generated for declarative statements that are not executed, such as DECLARE CURSOR, DECLARE TABLE, and DECLARE STATEMENT.

The following table lists values that are returned to SQLCODE:

Table 2-4: SQLCODE and Message Summary

SQLCODE	Message	Meaning
< 0	SQLERROR	Error occurred. Statement did not execute.
0	SUCCESS	Successful execution.
+1-99	SQLWARNING	System warning or informational message.
+100	NOT FOUND	No qualifying rows found, or end of current active set of rows reached.

In **isql** when an error occurs, an error message is displayed.

In embedded applications, the programmer must provide error handling by checking the value of SQLCODE.

To check SQLCODE, use one or a combination of the following approaches:

- Test for SQLCODE values with the WHENEVER statement.

- Check SQLCODE directly.
- Use the **isc_print_sqlerror()** routine to display specific error messages.

For more information about error handling, see the *Programmer's Guide*.

Using Statement and Function Definitions

Each statement and function definition includes the following elements:

Table 2-5: Statement and Function Format

Element	Description
Title	Statement name.
Definition	The statement's main purpose and availability.
Syntax	Diagram of the statement and its parameters.
Argument	Parameters available for use with the statement.
Description	Information about using the statement.
Examples	Examples of using the statement in a program and in **isql**.
See Also	Where to find more information about the statement or others related to it.

Most statements can be used in SQL, DSQL, and **isql**. In many cases, the syntax is nearly identical, except that embedded SQL statements must always be preceded by the EXEC SQL keywords. EXEC SQL is omitted from syntax statements for clarity.

In other cases there are small, but significant differences among SQL, DSQL, and **isql** syntax. In these cases, separate syntax statements appear under the statement heading.

ALTER DATABASE

Adds secondary files to the current database. Available in SQL, DSQL, and **isql**.

Syntax

```
ALTER {DATABASE | SCHEMA}
    ADD <add_clause>;
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in **isql**, the semicolon is a terminating symbol for the statement, so it must be included.

```
<add_clause> =
    FILE "<filespec>" [<fileinfo>] [<add_clause>]

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

Argument	Description
SCHEMA	Alternative keyword for DATABASE.
ADD FILE "*<filespec>*"	Adds one or more secondary files to receive database pages after the primary file is filled. For a remote database, associate secondary files with the same node.
LENGTH [=] *int* [PAGE[S]]	Specifies the range of pages for a secondary file by providing the number of pages in each file.
STARTING [AT [PAGE]] *int*	Specifies a range of pages for a secondary file by providing the starting page number.

Description

ALTER DATABASE adds 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.

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.

Example

The following **isql** statement adds secondary files to an existing database:

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

See Also

CREATE DATABASE, DROP DATABASE

For more information about multi-file databases, see the *Data Definition Guide*.

For more information about exclusive database access, see the *Windows Client User's Guide*.

ALTER DOMAIN

Changes a domain definition. Available in SQL, DSQL, and **isql**.

Syntax

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

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in **isql**, the semicolon is a terminating symbol for the statement, so it must be included.

```
<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> = {= | < | > | <= | >= | !< | !> | <> | !=}
```

Argument	Description
name	Name of an existing domain.
SET DEFAULT	Specifies a default column value that is entered when no other entry is made. Values: • *literal*—Inserts a specified string, numeric value, or date value. • NULL—Enters a NULL value. • USER—Enters the user name of the current user. Column must be of compatible text type to use the default. Defaults set at column level override defaults set at the domain level.
DROP DEFAULT	Drops existing default.

Argument	Description
ADD [CONSTRAINT] CHECK *<dom_search_condition>*	Adds a CHECK constraint to the domain definition. A domain definition can include only one CHECK constraint.
DROP CONSTRAINT	Drops CHECK constraint from the domain definition.

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

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

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

Example The following **isql** statements create a domain that must have a value > 1,000, then alter it by setting a default of 9,999:

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

See Also CREATE DOMAIN, CREATE TABLE, DROP DOMAIN

For a complete discussion of creating domains, and using them to create column definitions, see the *Data Definition Guide*.

ALTER EXCEPTION

Changes the message associated with an existing exception. Available in DSQL and **isql**.

Syntax `ALTER EXCEPTION` *`name`* `"`*`<message>`*`"`

Argument	Description
name	Name of an existing exception message.
"*<message>*"	Quoted string containing ASCII values.

Description ALTER EXCEPTION changes the text of an exception error message.

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

Example This **isql** statement alters the message of an exception:

```
ALTER EXCEPTION CUSTOMER_CHECK "Hold shipment for customer
   remittance.";
```

See Also ALTER PROCEDURE, ALTER TRIGGER, CREATE EXCEPTION, CREATE PROCEDURE, CREATE TRIGGER, DROP EXCEPTION

For more information on creating, raising, and handling exceptions, see the *Data Definition Guide*.

ALTER INDEX

Activates or deactivates an index. Available in SQL, DSQL, and **isql**.

Syntax

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

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in **isql**, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
name	Name of an existing index.
ACTIVE	Changes an INACTIVE index to an ACTIVE one.
INACTIVE	Changes an ACTIVE index to an INACTIVE one.

Description ALTER INDEX makes an inactive index available for use, or disables the use of an active index. Deactivating and reactivating an index is useful when changes in the distribution of indexed data cause the index to become unbalanced.

Before inserting or updating a large number of rows, deactivate a table's indexes to avoid altering the index incrementally. When finished, reactivate the index. Reactivating a deactivated index rebuilds and rebalances an index.

If an index is in use, ALTER INDEX does not take effect until the index is no longer in use.

ALTER INDEX fails and returns an error if the index is defined for a UNIQUE, PRIMARY KEY, or FOREIGN KEY constraint. To alter such an index, use DROP INDEX to delete the index, then recreate it with CREATE INDEX.

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

Note To add or drop index columns or keys, use DROP INDEX to delete the index, then recreate it with CREATE INDEX.

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

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

See Also ALTER TABLE, CREATE INDEX, DROP INDEX, SET STATISTICS

ALTER PROCEDURE

Changes the definition of an existing stored procedure. Available in DSQL and **isql**.

Syntax

```
ALTER PROCEDURE name
    [(param <datatype> [, param <datatype> ...])]
    [RETURNS (param <datatype> [, param <datatype> ...])]
    AS <procedure_body> [terminator]
```

Argument	Description
name	Name of an existing procedure.
param <datatype>	Input parameters used by the procedure. Legal data types are listed under CREATE PROCEDURE.
RETURNS *param <datatype>*	Output parameters used by the procedure. Legal data types are listed under CREATE PROCEDURE.
<procedure_body>	The procedure body. Includes: • Local variable declarations • A block of statements in procedure and trigger language See CREATE PROCEDURE for a complete description.
terminator	Terminator defined by the **isql** SET TERM command to signify the end of the procedure body. Required by **isql**.

Description ALTER PROCEDURE changes an existing stored procedure without affecting its dependencies. It can modify a procedure's input parameters, output parameters, and body.

The complete procedure header and body must be included in the ALTER PROCEDURE statement. The syntax is exactly the same as CREATE PROCEDURE, except CREATE is replaced by ALTER.

Caution Be careful about changing the type, number, and order of input and output parameters to a procedure, since existing application code may assume the

procedure has its original format.

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

Procedures in use are not altered until they are no longer in use.

ALTER PROCEDURE changes take effect when they are committed. Changes are then reflected in all applications that use the procedure without recompiling or relinking.

Example The following **isql** statements alter the GET_EMP_PROJ procedure, changing the return parameter to have a data type of VARCHAR(20):

```
SET TERM !! ;
ALTER PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT)
RETURNS (PROJ_ID VARCHAR(20)) AS
   BEGIN
      FOR SELECT PROJ_ID
      FROM EMPLOYEE_PROJECT
      WHERE EMP_NO = :emp_no
      INTO :proj_id
      DO
         SUSPEND;
   END !!
SET TERM ; !!
```

See Also CREATE PROCEDURE, DROP PROCEDURE, EXECUTE PROCEDURE

For more information on creating and using procedures, see the *Data Definition Guide*.

For a complete description of the statements in procedure and trigger language, see Chapter 3: “Procedure and Trigger Language Reference.”

ALTER TABLE

Changes a table by adding or dropping columns or integrity constraints. Available in SQL, DSQL, and **isql**.

Syntax `ALTER TABLE` *`table <operation>`* `[,` *`<operation>`* `...];`

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in **isql**, the semicolon is a terminating symbol for the statement, so it must be included.

```
<operation> = {ADD <col_def> | ADD <table_constraint> | DROP col
    | DROP CONSTRAINT constraint}

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

Note The COLLATE clause cannot be specified for BLOB columns.

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

<constraint_def> = {PRIMARY KEY | UNIQUE
    | CHECK (<search_condition>)
    | REFERENCES other_table [(other_col [, other_col ...])]}

<datatype> = {
    {SMALLINT | INTEGER | FLOAT | DOUBLE PRECISION} [<array_dim>]
    | {DECIMAL | NUMERIC} [(precision [, scale])] [<array_dim>]
    | DATE [<array_dim>]
    | {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR}
    [(1...32767)] [<array_dim>] [CHARACTER SET charname]
    | {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
    [VARYING] [(1...32767)] [<array_dim>]
    | BLOB [SUB_TYPE {int | subtype_name}] [SEGMENT SIZE n]
    [CHARACTER SET charname]
    | BLOB [(seglen [, subtype])]
    }

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

Note Outermost brackets, in bold, must be included when declaring arrays.

```
<table_constraint> = [CONSTRAINT constraint] <tconstraint_opt>
    [<table_constraint>]
<tconstraint_opt> = {
    {PRIMARY KEY | UNIQUE} (col [, col ...])
    | FOREIGN KEY (col [, col ...]) REFERENCES other_table
    | CHECK (<search_condition>)
    }
```

Note For the full syntax of <*search_condition*>, see CREATE TABLE.

Argument	Description
table	Name of an existing table to modify.
<*operation*>	Action to perform on the table. Valid options are: • ADD a new column or table constraint to a table. • DROP an existing column or constraint from a table.

Argument	Description
<col_def>	Description of a new column to add. Must include a column name and data type. Can include default values, column constraints, and a specific collation order.
<table_constraint>	Description of a new table constraint to add. Only one table constraint can be added to a table.
col	Name of column to add or drop. Column name must be unique within the table.
<constraint>	Name of constraint to add or drop. Constraint name must be unique within the table.
COLLATE *collation*	Adds a collation order to the specified table.
<datatype>	Data type of column to add.
domain	Name of a domain upon which a column definition should be based.
COMPUTED [BY] *<expr>*	Specifies a column calculated from *<expr>* and which is not therefore allocated storage space in the database. *<expr>* can be any arithmetic expression valid for the data types in the expression. Other columns referenced in *<expr>* must exist before they can be used. BLOB columns cannot be referenced. *<expr>* must return a single value, and cannot return an array.
NOT NULL	Specifies that a column cannot contain a NULL value. If a table already has rows, a new column cannot be NOT NULL. NOT NULL is only a column attribute.
DEFAULT	Specifies a default column value that is entered when no other entry is made. Values: • *literal*: Inserts a specified string, numeric value, or date value. • NULL: Enters a NULL value. • USER: Enters the user name of the current user. Column must be of compatible text type to use the default. • Defaults set at column level override defaults set at the domain level.
<constraint_def>	Column constraint definition.
CONSTRAINT	Adds a named constraint to a column.
DROP CONSTRAINT	Drops the specified table constraint.

Description

ALTER TABLE enables the structure of an existing table to be modified. A single ALTER TABLE can perform multiple adds and drops.

Creating PRIMARY KEY and FOREIGN KEY constraints requires exclusive access to the database.

Naming column constraints is optional. If a name is not specified, InterBase assigns a system-generated name. Assigning a descriptive name can make a constraint easier to find for changing or dropping, and easier to find when its name appears in a constraint violation error message.

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

ALTER TABLE fails if current data in a table violates a PRIMARY KEY or UNIQUE constraint definition added to the table. It also fails if a column to be dropped is:

- Part of a UNIQUE, PRIMARY, or FOREIGN KEY constraint, or is used in a CHECK constraint.
- Used in the value expression of a computed column.

Drop the constraint or computed column before dropping the table column. PRIMARY KEY and UNIQUE constraints cannot be dropped if referenced by FOREIGN KEY constraints. In these cases, drop the FOREIGN KEY constraint before dropping the PRIMARY KEY or UNIQUE key it references.

When altering a column based on a domain, an additional CHECK constraint can be supplied for the column. Changes to tables that contain CHECK constraints with subqueries may cause constraint violations.

Caution When a column is altered or dropped any data stored in it is lost.

Examples The following **isql** statement adds a column to a table and drops a column:

```
ALTER TABLE COUNTRY
   ADD CAPITAL VARCHAR(25),
   DROP CURRENCY;
```

Note This statement results in the loss of any data in the dropped column.

The next **isql** statement adds two columns to a table and defines a UNIQUE constraint on one of them:

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

See Also ALTER DOMAIN, CREATE DOMAIN, CREATE TABLE

For more information about preserving data in columns before altering them, see the *Programmer's Guide*.

ALTER TRIGGER

Changes an existing trigger. Available in DSQL and **isql**.

Syntax

```
ALTER TRIGGER name
    [ACTIVE | INACTIVE]
    [{BEFORE | AFTER} {DELETE | INSERT | UPDATE}]
    [POSITION number]
    [AS <trigger_body>] [terminator]
```

Argument	Description
name	Name of an existing trigger.
ACTIVE	Specifies that a trigger action takes effect when fired (default).
INACTIVE	Specifies that a trigger action does *not* take effect.
BEFORE	Specifies the trigger fires before the associated operation takes place.
AFTER	Specifies the trigger fires after the associated operation takes place.
DELETE | INSERT | UPDATE	Specifies the table operation that causes the trigger to fire.
POSITION *number*	Specifies order of firing 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. Triggers for a table need not be consecutive. Triggers on the same action with the same position number will fire in random order.
<trigger_body>	Body of the trigger, a block of statements in procedure and trigger language. See CREATE TRIGGER for a complete description.
terminator	Terminator defined by the **isql** SET TERM command to signify the end of the trigger body. Not needed when altering only the trigger header.

Description

ALTER TRIGGER changes the definition of an existing trigger. If any of the arguments to ALTER TRIGGER are omitted, then they default to their current values, that is the value specified by CREATE TRIGGER, or the last ALTER TRIGGER.

ALTER TRIGGER can change:

- Header information only, 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.
- Body information only, the trigger statements that follow the AS clause.
- Header and trigger body information. In this case, the new trigger definition replaces the old trigger definition.

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

Note To alter a trigger defined automatically by a CHECK constraint on a table, use ALTER TABLE to change the constraint definition.

Examples The following **isql** statement modifies the trigger, SET_CUST_NO, to be inactive:

```
ALTER TRIGGER SET_CUST_NO INACTIVE;
```

The next **isql** statement modifies the trigger, SET_CUST_NO, to insert a row into the table, NEW_CUSTOMERS, for each new customer.

```
SET TERM !! ;
ALTER TRIGGER SET_CUST_NO FOR CUSTOMER
BEFORE INSERT AS
   BEGIN
      NEW.CUST_NO = GEN_ID(CUST_NO_GEN, 1);
      INSERT INTO NEW_CUSTOMERS(NEW.CUST_NO, TODAY)
   END !!
SET TERM ; !!
```

See Also CREATE TRIGGER, DROP TRIGGER

For a complete description of the statements in procedure and trigger language, see Chapter 3: “Procedure and Trigger Language Reference.”

For more information about triggers, see the *Data Definition Guide*.

AVG( )

Calculates the average of numeric values in a specified column or expression. Available in SQL, DSQL, and **isql**.

Syntax `AVG ([ALL] <val> | DISTINCT <val>)`

Argument	Description
ALL	Returns the average of all values.
DISTINCT	Eliminates duplicate values before calculating the average.
<val>	A column or expression that evaluates to a numeric data type.

Description AVG() is an aggregate function that returns the average of the values in a specified column or expression. Only numeric data types are allowed as input to AVG().

If a field value involved in a calculation is NULL or unknown, it is automatically excluded from the calculation. Automatic exclusion prevents averages from being skewed by meaningless data.

AVG() computes its value over a range of selected rows. If the number of rows returned by a SELECT is zero, AVG() returns a NULL value.

Examples The following embedded SQL statement returns the average of all rows in a table:

```
EXEC SQL
   SELECT AVG (BUDGET) FROM DEPARTMENT INTO :avg_budget;
```

The next embedded SQL statement demonstrates the use of SUM(), AVG(), MIN(), and MAX() over a subset of rows in a table:

```
EXEC SQL
   SELECT SUM (BUDGET), AVG (BUDGET), MIN (BUDGET), MAX (BUDGET)
   FROM DEPARTMENT
   WHERE HEAD_DEPT = :head_dept
   INTO :tot_budget, :avg_budget, :min_budget, :max_budget;
```

See Also COUNT(), MAX(), MIN(), SUM()

BASED ON

Declares a host-language variable based on a column. Available in SQL.

Syntax `BASED [ON] [`*`dbhandle`*`.]`*`table.col`*`[.SEGMENT] `*`variable`*`;`

Argument	Description
dbhandle.	Handle for the database in which a table resides in a multi-database program. *dbhandle* must be previously declared in a SET DATABASE statement.
table.col	Name of table and name of column on which the variable is based.
.SEGMENT	Bases the local variable size on the segment length of the BLOB column during BLOB FETCH operations. Use only when *table.col* refers to a column of BLOB data type.
variable	Name of the host-language variable that inherits the characteristics of a database column.

Description BASED ON is a preprocessor directive that creates a host-language variable based on a column definition. The host variable inherits the attributes described for the column and any characteristics that make the variable type consistent with the programming language in use. For example, in C, BASED ON adds one byte to CHAR and VARCHAR variables to accommodate the NULL character terminator.

Use BASED ON in a program's variable declaration section.

Note BASED ON does not require the EXEC SQL keywords.

To declare a host-language variable large enough to hold a BLOB segment during FETCH operations, use the SEGMENT option of the BASED ON clause. The variable's size is derived from the segment length of a BLOB column. If the segment length for the BLOB column is changed in the database, recompile the program to adjust the size of host variables created with BASED ON.

Examples The following embedded statements declare a host variable based on a column:

```
EXEC SQL
   BEGIN DECLARE SECTION
      BASED_ON EMPLOYEE.SALARY salary;
EXEC SQL
   END DECLARE SECTION;
```

See Also BEGIN DECLARE SECTION, CREATE TABLE, END DECLARE SECTION

BEGIN DECLARE SECTION

Identifies the start of a host-language variable declaration section. Available in SQL.

Syntax

```
BEGIN DECLARE SECTION;
```

Description

BEGIN DECLARE SECTION is used in embedded SQL applications to identify the start of host-language variable declarations for variables that will be used in subsequent SQL statements. BEGIN DECLARE SECTION is also a preprocessor directive that instructs **gpre** to declare SQLCODE automatically for the applications programmer.

Important

BEGIN DECLARE SECTION must always appear within a module's global variable declaration section.

Example

The following embedded SQL statements declare a section and a host-language variable:

```
EXEC SQL
   BEGIN DECLARE SECTION;
      BASED ON EMPLOYEE.SALARY salary;
EXEC SQL
   END DECLARE SECTION;
```

See Also

BASED ON, END DECLARE SECTION

CAST( )

Converts a column from one data type to another. Available in SQL, DSQL, and **isql**.

Syntax

CAST (*<val>* AS *<datatype>*)

Argument	Description
<val>	A column, constant, or expression. In SQL, *<val>* can also be a host-language variable, function, or UDF.
<datatype>	Data type to which to convert.

Description

CAST() allows mixing of numerics and characters in a single expression by converting *<val>* to a specified data type.

Normally, only similar data types can be compared in search conditions. CAST() can be used in search conditions to translate one data type into another for comparison purposes.

Data types can be converted as shown in the following table:

Table 2-6: Compatible Data Types for CAST()

From Data Type Class	To Data Type Class
numeric	character, varying character, date
character, varying character	numeric, date
date	character, varying character, date
BLOB, arrays	—

An error results if a given data type cannot be converted into the data type specified in CAST().

Example In the following WHERE clause, CAST() is used to translate a CHARACTER data type, INTERVIEW_DATE, to a DATE data type to compare against a DATE data type, HIRE_DATE:

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

See Also UPPER()

CLOSE

Closes an open cursor. Available in SQL.

Syntax

```
CLOSE cursor;
```

Argument	Description
cursor	Name of an open cursor.

Description CLOSE terminates the specified cursor, releasing the rows in its active set and any associated system resources. A cursor is a one-way pointer into the ordered set of rows retrieved by the select expression in the DECLARE CURSOR statement. A cursor enables sequential access to retrieved rows in turn and update in place.

There are four related cursor statements:

Stage	Statement	Purpose
1	DECLARE CURSOR	Declares the cursor. The SELECT statement determines rows retrieved for the cursor.
2	OPEN	Retrieves the rows specified for retrieval with DECLARE CURSOR. The resulting rows become the cursor's *active set*.
3	FETCH	Retrieves the current row from the active set, starting with the first row. Subsequent FETCH statements advance the cursor through the set.
4	CLOSE	Closes the cursor and releases system resources.

FETCH statements cannot be issued against a closed cursor. Until a cursor is closed and reopened, InterBase does not reevaluate values passed to the search conditions. Another user can commit changes to the database while a cursor is open, making the active set different the next time that cursor is reopened.

Note In addition to CLOSE, COMMIT and ROLLBACK automatically close all cursors in a transaction.

Example The following embedded SQL statement closes a cursor:

```
EXEC SQL
   CLOSE BC;
```

See Also CLOSE (BLOB), COMMIT, DECLARE CURSOR, FETCH, OPEN, ROLLBACK

CLOSE (BLOB)

Terminates a specified BLOB cursor and releases associated system resources. Available in SQL.

Syntax `CLOSE` *`blob_cursor`*`;`

Argument	Description
blob_cursor	Name of an open BLOB cursor.

Description CLOSE closes an opened read or insert BLOB cursor. Generally a BLOB cursor should only be closed after:

- Fetching all the BLOB segments for BLOB READ operations.
- Inserting all the segments for BLOB INSERT operations.

Example The following embedded SQL statement closes a BLOB cursor:

```
EXEC SQL
   CLOSE BC;
```

See Also DECLARE CURSOR (BLOB), FETCH (BLOB), INSERT CURSOR (BLOB), OPEN (BLOB)

COMMIT

Makes a transaction's changes to the database permanent, and ends the transaction. Available in SQL, DSQL, and **isql**.

Syntax `COMMIT [WORK] [TRANSACTION` *name*`] [RELEASE] [RETAIN [SNAPSHOT]];`

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in **isql**, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
WORK	An optional word used for compatibility with other relational databases that require it.
TRANSACTION *name*	Commits transaction *name* to database. Without this option, COMMIT affects the default transaction.
RELEASE	Available for compatibility with earlier versions of InterBase.
RETAIN [SNAPSHOT]	Commits changes and retains current transaction context.

Description COMMIT is used to end a transaction and:

- Write all updates to the database.
- Make the transaction's changes visible to subsequent SNAPSHOT transactions or READ COMMITTED transactions.
- Close open cursors, unless the RETAIN argument is used.

A transaction ending with COMMIT is considered a successful termination. Always use COMMIT or ROLLBACK to end the default transaction.

Tip After read-only transactions, which make no database changes, use COMMIT rather than ROLLBACK. The effect is the same, but the performance of subsequent transactions is better and the system resources used by them are reduced.

Important The RELEASE argument is only available for compatibility with previous versions of InterBase. To detach from a database use DISCONNECT.

Examples The following **isql** statement makes permanent the changes to the database made by the default transaction:

```
COMMIT;
```

The next embedded SQL statement commits a named transaction:

```
EXEC SQL
   COMMIT TR1;
```

The following embedded SQL statement uses COMMIT RETAIN to commit changes while maintaining the current transaction context:

```
EXEC SQL
   COMMIT RETAIN;
```

See Also DISCONNECT, ROLLBACK

For more information about handling transactions, see the *Programmer's Guide*.

CONNECT

Attaches to one or more databases. Available in SQL. A subset of CONNECT options is available in **isql**.

Syntax SQL form:

```
CONNECT [TO] {{ALL | DEFAULT} [<config_opts>] | <db_specs>};

<db_specs> = {{"<filespec>" | :variable} AS dbhandle | dbhandle}
    [<config_opts>] [, <db_specs>]

<config_opts> = USER {"username" | :variable}
    | PASSWORD {"password" | :variable}
    | CACHE int [BUFFERS] [<config_opts>]
```

isql form:

```
CONNECT ["]<filespec>["] [USER "username"[PASSWORD "password"]]
```

Argument	Description
{ALL | DEFAULT}	Connects to all databases specified with SET DATABASE. Any options specified with CONNECT TO ALL affect all databases.
"*<filespec>*"	Database file name. Can include path specification and node.

Argument	Description
dbhandle	Database handle declared in a previous SET DATABASE statement.
:*variable*	Host-language variable specifying a database, user name, or password.
AS	Attaches to a database and assigns a previously declared handle to it.
USER "*username*" | :*variable*	String or host-language variable that specifies a user name for use when attaching to the database. The server checks the user name against the security database. User names are case insensitive on the server.
PASSWORD "*password*" | :*variable*	String or host-language variable, up to 8 characters in size, that specifies password for use when attaching to the database. The server checks the user name and password against the security database. Case sensitivity is retained for the comparison.
CACHE *int* [BUFFERS]	Sets the number of cache buffers for a database, which determines the number of database pages a program can use at the same time. Values for *int*: • Default: 75. • Minimum value: 45. • Maximum value: System-dependent. Do not use "*<filespec>*" form of database name with cache assignments.

Description

The CONNECT statement:

- Initializes database data structures.
- Determines if the database is on the originating node (a *local database*) or on another node (a *remote database*). Databases used by PC clients are always located on remote servers. An error message occurs if InterBase cannot locate the database.
- Optionally specifies a user name and password for use when attaching to the database. PC clients must always send a valid user name and password. Passwords are restricted to 8 characters in length.
- Attaches to the database and verifies the header page. The database file must contain a valid database, and the on-disk structure (ODS) version number of the database must be the one recognized by the installed version of InterBase on the server, or InterBase returns an error.
- Optionally establishes a database handle declared in a SET DATABASE statement.
- Specifies a cache buffer for the process attaching to a database.

In SQL programs before a database can be opened with CONNECT, it must be declared with the SET DATABASE statement. **isql** does not use SET DATABASE.

In SQL programs while the same CONNECT statement can open more than one database, use separate statements to keep code easy to read.

When CONNECT attaches to a database, it uses the default character set (NONE), or one specified in a previous SET NAMES statement.

In SQL programs the CACHE option changes the database cache size count (the total number of available buffers) from the default of 75. This option can be used to:

- Set a new default size for all databases listed in the CONNECT statement that do not already have a specific cache size.
- Specify a cache for a program that uses a single database.
- Change the cache for one database without changing the default for all databases used by the program.

The size of the cache persists as long as the attachment is active. If a database is already attached through a multi-client server, an increase in cache size due to a new attachment persists until all the attachments end. A decrease in cache size does not affect databases that are already attached through a server.

A subset of CONNECT features is available in **isql**: database file name, USER, and PASSWORD. **isql** can only be connected to one database at a time. Each time CONNECT is used to attach to a database, previous attachments are disconnected.

Examples

The following statement opens a database for use in **isql**. It uses all the CONNECT options available to **isql**:

```
CONNECT "employee.gdb" USER "ACCT_REC" PASSWORD "peanuts";
```

The next statement, from an embedded application, attaches to a database file stored in the host-language variable and assigns a previously declared database handle to it:

```
EXEC SQL
   SET DATABASE DB1 = "employee.gdb";
EXEC SQL
   CONNECT :db_file AS DB1;
```

The following embedded SQL statement attaches to *employee.gdb* and allocates 150 cache buffers:

```
EXEC SQL
   CONNECT "accounts.gdb" CACHE 150;
```

The next embedded SQL statement connects the user to all databases specified with previous SET DATABASE statements:

```
EXEC SQL
   CONNECT ALL USER "ACCT_REC" PASSWORD "peanuts"
   CACHE 50;
```

The following embedded SQL statement attaches to the database, *employee.gdb*, with 80 buffers and database *employee2.gdb* with the default of 75 buffers:

```
EXEC SQL
   CONNECT "employee.gdb" CACHE 80, "employee2.gdb";
```

The next embedded SQL statement attaches to all databases and allocates 50 buffers:

```
EXEC SQL
   CONNECT ALL CACHE 50;
```

The following embedded SQL statement connects to EMP1 and EMP2, setting the number of buffers for each to 80:

```
EXEC SQL
   CONNECT EMP1 CACHE 80, EMP2 CACHE 80;
```

The next embedded SQL statement connects to two databases identified by variable names, setting different user names and passwords for each:

```
EXEC SQL
   CONNECT
   :orderdb AS DB1 USER "ACCT_REC" PASSWORD "peanuts",
   :salesdb AS DB2 USER "ACCT_PAY" PASSWORD "payout";
```

See Also DISCONNECT, SET DATABASE, SET NAMES

For more information about cache buffers, see the *Data Definition Guide*.

For more information about database security, see the *Windows Client User's Guide*.

For more information about **isql**, see *Getting Started*.

COUNT( )

Calculates the number of rows that satisfy a query's search condition. Available in SQL, DSQL, and **isql**.

Syntax `COUNT ( * | [ALL] <val> | DISTINCT <val>)`

Argument	Description
*	Retrieves the number of rows in a specified table, including NULL values.
ALL	Counts all non-NULL values in a column.
DISTINCT	Returns the number of unique, non-NULL values for the column.
<val>	A column or expression.

Description COUNT() is an aggregate function that returns the number of rows that satisfy a query's search condition. It may be used in views and joins as well as tables.

Example The following embedded SQL statement returns the number of unique currency values it encounters in the COUNTRY table:

```
EXEC SQL
   SELECT COUNT (DISTINCT CURRENCY) INTO :cnt FROM COUNTRY;
```

See Also AVG(), MAX(), MIN(), SUM()

CREATE DATABASE

Creates a new database. Available in SQL, DSQL, and **isql**.

Syntax

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

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in **isql**, the semicolon is a terminating symbol for the statement, so it must be included.

```
<secondary_file> = FILE "<filespec>" [<fileinfo>] [<secondary_file>]

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

Argument	Description
"*<filespec>*"	A new database file specification. File naming conventions are platform-specific.
USER "*username*"	If provided, the user name is checked against valid user name and password combinations in the security database on the server where the database will reside. Windows client applications must provide a user name on attachment to a server. Any client application attaching to a database on NT or NetWare must provide a user name on attachment.
PASSWORD "*password*"	Up to 8 characters. The password is checked against valid user name and password combinations in the security database on the server where the database will reside. Windows client applications must provide a user name and password on attachment to a server. Any client application attaching to a database on NT or NetWare must provide a password on attachment.
PAGE_SIZE [=] *int*	Size, in bytes, for database pages. *int* can be 1024 (default), 2048, 4096, or 8192.
DEFAULT CHARACTER SET *charset*	Sets default character set for a database. *charset* is the name of a character set. If omitted, character set defaults to NONE.
FILE "*<filespec>*"	Names one or more secondary files to hold database pages after the primary file is filled. For databases created on remote servers, secondary file specifications cannot include a node name.
STARTING [AT [PAGE]] *int*	Specifies the starting page number for a secondary file.
LENGTH [=] *int* [PAGE[S]]	Specifies the length of a primary or secondary database file. Use for primary file only if defining a secondary file in the same statement.

Description

CREATE DATABASE creates a database and establishes the following characteristics for it:

- The name of the primary file that identifies the database for users. By default, databases are contained in single files.
- The name of any secondary files in which the database is stored. A database can reside in more than one disk file if additional file names are specified as secondary files. If a database is created on a remote server, secondary file specifications cannot include a node name.
- The size of database pages. Increasing page size can improve performance for the following 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.
- 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 may be appropriate, since less data needs to be passed back and forth and less memory is used by the disk cache.

- The number of pages in each database file.
- The character set used by the database. For a list of the character sets recognized by InterBase, see Appendix D: "Character Sets and Collation Orders."

  Choice of DEFAULT CHARACTER SET limits possible collation orders to a subset of all available collation orders. Given a specific character set, a specific collation order can be specified when data is selected, inserted, or updated in a column.

  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. In that case, no transliteration is performed between the source and destination character sets, and transliteration errors may occur during assignment.
- System tables that describe the structure of the database.

After creating the database, a user may define its tables, views, indexes, and system views.

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

Examples The following **isql** statement creates a database in the current directory using **isql**:

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

The next embedded SQL statement creates a database with a page size of 2048 bytes rather than the default of 1024:

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

The following embedded SQL statement creates a database stored in two files and specifies its default character set:

```
EXEC SQL
   CREATE DATABASE "employee.gdb"
      DEFAULT CHARACTER SET ISO8859_1
      FILE "employee.gd1" STARTING AT PAGE 10001 LENGTH 10000 PAGES;
```

See Also ALTER DATABASE, DROP DATABASE

For more information about secondary files, character set specification, and collation order, see the *Data Definition Guide*.

For more information about page size, see the *Windows Client User's Guide*.

CREATE DOMAIN

Creates a column definition that is global to the database. Available in SQL, DSQL, and **isql**.

Syntax

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

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in **isql**, the semicolon is a terminating symbol for the statement, so it must be included.

Note The COLLATE clause cannot be specified for BLOB columns.

```
<datatype> = {
    {SMALLINT | INTEGER | FLOAT | DOUBLE PRECISION} [<array_dim>]
    | {DECIMAL | NUMERIC} [(precision [, scale])] [<array_dim>]
    | DATE [<array_dim>]
    | {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR}
    [(1...32767)] [<array_dim>] [CHARACTER SET charname]
    | {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
    [VARYING] [(1...32767)] [<array_dim>]
    | BLOB [SUB_TYPE {int | subtype_name}] [SEGMENT SIZE int]
    [CHARACTER SET charname]
    | BLOB [(seglen [, subtype])]
    }

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

Note Outermost brackets, in bold, must be included when declaring arrays.

```
<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> = {= | < | > | <= | >= | !< | !> | <> | !=}
```

Argument	Description
domain	Unique name for the domain.
<datatype>	SQL data type.
DEFAULT	Specifies a default column value that is entered when no other entry is made. Values: • *literal*—Inserts a specified string, numeric value, or date value. • NULL—Enters a NULL value. • USER—Enters the user name of the current user. Column must be of compatible character type to use the default.
NOT NULL	Specifies that the values entered in a column cannot be NULL.
CHECK (*<dom_ search_ condition>*)	Creates a single CHECK constraint for the domain.
VALUE	Placeholder for the name of a column eventually based on the domain.
COLLATE *collation*	Specifies a collation sequence for the domain.

Description

CREATE DOMAIN builds an inheritable column definition that acts as a template for columns defined with CREATE TABLE or ALTER TABLE. The domain definition contains a set of characteristics, which include:

- Data type
- An optional default value
- Optional disallowing of NULL values
- An optional CHECK constraint
- An optional collation clause

The CHECK constraint in a domain definition sets a <*dom_search_condition*> that must be true for data entered into columns based on the domain. The CHECK constraint cannot reference any domain or column.

Note Be careful when creating domains. It is possible to create a domain with contradictory constraints, such as declaring a domain NOT NULL, and assigning it a DEFAULT value of NULL.

The data type specification for a CHAR, VARCHAR, or BLOB text domain definition can include a CHARACTER SET clause to specify a character set for the domain. Otherwise, the domain uses the default database character set. For a complete list of character sets recognized by InterBase, see Appendix D: "Character Sets and Collation Orders."

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. In these cases, no transliteration is performed between the source and destination character sets, so errors can occur during assignment.

The COLLATE clause enables specification of a particular collation order for CHAR, VARCHAR, and BLOB text data types. Choice of collation order is restricted to those supported for the domain's given character set, which is either the default character set for the entire database, or a different set defined in the CHARACTER SET clause as part of the data type definition. For a complete list of collation orders recognized by InterBase, see Appendix D: "Character Sets and Collation Orders."

Columns based on a domain definition inherit all characteristics of the domain. The domain default, collation clause, and NOT NULL setting may be overridden when defining a column based on a domain. A column based on a domain can add additional CHECK constraints to the domain CHECK constraint.

Examples The following **isql** statement creates a domain that must have a positive value greater than 1,000, with a default value of 9,999. The keyword VALUE substitutes for the name of a column based on this domain.

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

The next **isql** 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"));
```

The following **isql** statement creates a domain that defines an array of CHAR data type:

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

In the following **isql** example, the first statement creates a domain with USER 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
```

The next **isql** 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;
```

See Also ALTER DOMAIN, ALTER TABLE, CREATE TABLE, DROP DOMAIN

For more information about character set specification and collation orders, see the *Data Definition Guide*.

CREATE EXCEPTION

Creates a used-defined error and associated message for use in stored procedures and triggers. Available in DSQL and **isql**.

Syntax `CREATE EXCEPTION` *name* "*<message>*";

Important In SQL statements passed to DSQL, omit the terminating semicolon. In **isql**, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
name	Name associated with the exception message. Must be unique among exception names in the database.
"*<message>*"	Quoted string containing alphanumeric characters and punctuation. Maximum length: 78 characters.

Description

CREATE EXCEPTION creates an exception, a user-defined error with an associated message. Exceptions may be raised in triggers and stored procedures.

Exceptions are global to the database. The same message or set of messages is available to all stored procedures and triggers in an application. For example, a database can have English and French versions of the same exception messages and use the appropriate set as needed.

When raised by a trigger or a stored procedure, an exception:

- Terminates the trigger or procedure in which it was raised and undoes any actions performed (directly or indirectly) by it.
- Returns an error message to the calling application. In **isql**, the error message appears on the screen, unless output is redirected.

Exceptions may be trapped and handled with a WHEN statement in a stored procedure or trigger.

Examples

This **isql** statement creates the exception, UNKNOWN_EMP_ID:

```
CREATE EXCEPTION UNKNOWN_EMP_ID "Invalid employee number or project
   id.";
```

The following statement from a stored procedure raises the previously created exception when SQLCODE -530 is set, which is a violation of a FOREIGN KEY constraint:

```
. . .
   WHEN SQLCODE -530 DO
      EXCEPTION UNKNOWN_EMP_ID;
. . .
```

See Also

ALTER EXCEPTION, ALTER PROCEDURE, ALTER TRIGGER, CREATE PROCEDURE, CREATE TRIGGER, DROP EXCEPTION

For more information on creating, raising, and handling exceptions, see the *Data Definition Guide*.

CREATE GENERATOR

Declares a generator to the database. Available in SQL, DSQL, and **isql**.

Syntax

```
CREATE GENERATOR name;
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in **isql**, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
name	Name for the generator.

Description

CREATE GENERATOR declares a generator to the database and sets its starting value to zero. A generator is a sequential number that can be automatically inserted in a column with the GEN_ID() function. A generator is often used to ensure a unique value in a PRIMARY KEY, such as an invoice number, that must uniquely identify the associated row.

A database can contain any number of generators. Generators are global to the database, and can be used and updated in any transaction. InterBase does not assign duplicate generator values across transactions.

After a generator is created, SET GENERATOR can set or change its current value. The generator can be used by writing a trigger, procedure, or SQL statement that calls GEN_ID().

Example

The following **isql** script fragment creates the generator, EMPNO_GEN, and the trigger, CREATE_EMPNO. The trigger uses the generator to produce sequential numeric keys, incremented by 1, for the NEW.EMPNO column:

```
CREATE GENERATOR EMPNO_GEN;
COMMIT;
SET TERM !! ;
CREATE TRIGGER CREATE_EMPNO FOR EMPLOYEES
   BEFORE INSERT
   POSITION 0
   AS BEGIN
      NEW.EMPNO = GEN_ID(EMPNO_GEN, 1);
   END
SET TERM ; !!
```

Important Because each statement in a stored procedure body must be terminated by a semicolon, you must define a different symbol to terminate the CREATE TRIGGER in **isql**. Use SET TERM before CREATE TRIGGER to specify a

terminator other than a semicolon. After CREATE TRIGGER, include another SET TERM to change the terminator back to a semicolon.

See Also GEN_ID(), SET GENERATOR

CREATE INDEX

Creates an index on one or more columns in a table. Available in SQL, DSQL, and **isql**.

Syntax

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

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in **isql**, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
UNIQUE	Prevents insertion or updating of duplicate values into indexed columns.
ASC[ENDING]	Sorts columns in ascending order, the default order if none is specified.
DESC[ENDING]	Sorts columns in descending order.
index	Unique name for the index.
table	Name of the table on which the index is defined.
col	Column in *table* to index.

Description Creates an index on one or more columns in a table. Use CREATE INDEX to improve speed of data access. Using an index for columns that appear in a WHERE clause may improve search performance.

Important BLOB columns and arrays cannot be indexed.

A UNIQUE index cannot be created on a column or set of columns that already contains duplicate or NULL values.

ASC and DESC specify the order in which an index is sorted. For faster response to queries that require sorted values, use the index order that matches the query's ORDER BY clause. Both an ASC and a DESC index can be created on the same column or set of columns to access data in different orders.

Tip 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.

Examples The following **isql** statement creates a unique index:

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

The next **isql** statement creates a descending index:

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

The following **isql** statement creates a two-column index:

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

See Also ALTER INDEX, DROP INDEX, SELECT, SET STATISTICS

CREATE PROCEDURE

Creates a stored procedure, its input and output parameters, and its actions. Available in DSQL, and **isql**.

Syntax

```
CREATE PROCEDURE name
    [(param <datatype> [, param <datatype> ...])]
    [RETURNS <datatype> [, param <datatype> ...])]
    AS <procedure_body> [terminator]

<procedure_body> =
    [<variable_declaration_list>]
    <block>

<variable_declaration_list> =
DECLARE VARIABLE var <datatype>;
[DECLARE VARIABLE var <datatype>; ...]

<block> =
BEGIN
    <compound_statement>
    [<compound_statement> ...]
END

<compound_statement> = {<block> | statement;}
```

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

Argument	Description
name	Name of the procedure. Must be unique among procedure, table, and view names in the database.
param <datatype>	Input parameters that the calling program uses to pass values to the procedure: • *param*—Name of the input parameter, unique for variables in the procedure. • *<datatype>*—An InterBase data type.
RETURNS *param <datatype>*	Output parameters that the procedure uses to return values to the calling program: • *param*—Name of the output parameter, unique for variables within the procedure. • *<datatype>*—An InterBase 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 <datatype>*	Declares local variables used only in the procedure. Each declaration must be preceded by DECLARE VARIABLE and followed by a semicolon (;). *var* is the name of the local variable, unique for variables in the procedure.
statement	Any single statement in InterBase procedure and trigger language. Each statement (except BEGIN and END) must be followed by a semicolon (;).
terminator	Terminator defined by SET TERM which signifies the end of the procedure body. Used in **isql** only.

Description

CREATE PROCEDURE defines a new stored procedure to a database. A stored procedure is a self-contained program written in InterBase procedure and trigger language, and stored as part of a database's metadata. Stored procedures can receive input parameters from and return values to applications.

InterBase procedure and trigger language includes all SQL data manipulation statements and some powerful extensions, including IF . . . THEN . . . ELSE, WHILE . . . DO, FOR SELECT . . . DO, exceptions, and error handling.

There are two types of 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, or an error will result.
- *Executable* procedures that an application can call directly, with the EXECUTE PROCEDURE statement. An executable procedure need not return values to the calling program.

A stored procedure is composed of a *header* and a *body*.

The procedure 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 that a procedure receives from the calling program.
- RETURNS followed by a list of *output parameters* and their data types if the procedure returns values to the calling program.

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 may be many levels of nesting.

Important Because each statement in a stored procedure body must be terminated by a semicolon, you must define a different symbol to terminate the CREATE PROCEDURE statement in **isql**. Use SET TERM before CREATE PROCEDURE to specify a terminator other than a semicolon. After the CREATE PROCEDURE statement, include another SET TERM to change the terminator back to a semicolon.

InterBase does not allow database changes that affect the behavior of an existing stored procedure (e.g., DROP TABLE, DROP EXCEPTION). To see all procedures defined for the current database or the text and parameters of a named procedure, use the **isql** internal commands SHOW PROCEDURES or SHOW PROCEDURE *procedure*.

InterBase procedure and trigger language is a complete programming language for stored procedures and triggers. It includes:

- SQL data manipulation statements: INSERT, UPDATE, DELETE, and singleton SELECT.

- SQL operators and expressions, including UDFs linked with the database and generators.
- Powerful extensions to SQL, including assignment statements, control-flow statements, context variables (for triggers), event-posting statements, exceptions, and error-handling statements.

The following table summarizes language extensions for stored procedures. For a complete description of each statement, see Chapter 3: "Procedure and Trigger Language Reference."

Table 2-7: 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* */	Programmer's comment, where *comment_text* can be any number of lines of text.
EXCEPTION *exception_name*	Raises the named exception. An exception is a user-defined error, which returns an error message to the calling application unless handled by a WHEN statement.
EXECUTE PROCEDURE *proc_name* [*var* [, *var* ...]] [RETURNING_VALUES *var* [, *var* ...]]	Executes stored procedure, *proc_name,* with the listed input arguments, returning values in the listed output arguments following RETURNING_VALUES. Input and output arguments must be local variables.
EXIT	Jumps to the final END statement in the procedure.
FOR <*select_statement*> DO <*compound_statement*>	Repeats the statement or block following DO for every qualifying row retrieved by <*select_statement*>. <*select_statement*>: a normal SELECT statement, except the INTO clause is required and must come last.
<*compound_statement*>	Either a single statement in procedure and trigger language or a block of statements bracketed by BEGIN and END.

Table 2-7: Procedure and Trigger Language Extensions (Continued)

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.
NEW.*column*	New context variable that indicates a new column value in an INSERT or UPDATE operation.
OLD.*column*	Old context variable that indicates a column value before an UPDATE or DELETE operation.
POST_EVENT *event_name* | *col*	Posts the event, *event_name*, or uses the value in *col* as an 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.

Examples

The following procedure, SUB_TOT_BUDGET, takes a department number as its input parameter, and returns the total, average, minimum, and maximum budgets of departments with the specified HEAD_DEPT.

```
/* Compute total, average, smallest, and largest department budget.
*Parameters:
* department id
*
*Returns:
* total budget
* average budget
```

```
* min budget
* max budget */

SET TERM !! ;
CREATE PROCEDURE SUB_TOT_BUDGET (HEAD_DEPT CHAR(3))
RETURNS (tot_budget DECIMAL(12, 2), avg_budget DECIMAL(12, 2),
   min_budget DECIMAL(12, 2), max_budget DECIMAL(12, 2))
AS
BEGIN
   SELECT SUM(BUDGET), AVG(BUDGET), MIN(BUDGET), MAX(BUDGET)
   FROM DEPARTMENT
   WHERE HEAD_DEPT = :head_dept
   INTO :tot_budget, :avg_budget, :min_budget, :max_budget;
   EXIT;
END !!
SET TERM ; !!
```

The following select procedure, ORG_CHART, displays an organizational chart:

```
/* Display an org-chart.
*
*  Parameters:
*     --
*  Returns:
*     parent department
*     department name
*     department manager
*     manager's job title
*     number of employees in the department */

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 !!
```

When ORG_CHART is invoked, for example in the following **isql** statement:

```
SELECT * FROM ORG_CHART
```

It displays the department name for each department, which department it is in, 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, the first time it encounters the SUSPEND statement, it terminates, returning the information for Corporate Headquarters only.

See Also ALTER EXCEPTION, ALTER PROCEDURE, CREATE EXCEPTION, DROP EXCEPTION, DROP PROCEDURE, EXECUTE PROCEDURE, SELECT

For more information on creating and using procedures, see the *Data Definition Guide*.

For a complete description of the statements in procedure and trigger language, see Chapter 3: "Procedure and Trigger Language Reference."

CREATE SHADOW

Creates one or more duplicate, in-sync copies of a database. Available in SQL, DSQL, and **isql**.

Syntax

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

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in **isql**, the semicolon is a terminating symbol for the statement, so it must be included.

```
<secondary_file> = FILE "<filespec>" [<fileinfo>] [<secondary_file>]

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

Argument	Description
set_num	Positive integer that designates a shadow set to which all subsequent files listed in the statement belong.
AUTO	Specifies the default access behavior for databases in the event of shadow unavailability. All attachments and accesses succeed; references to the shadow are deleted and the shadow file is detached.
MANUAL	Specifies that database attachments and accesses fail until a shadow becomes available, *or* all references to the shadow are removed from the database.
CONDITIONAL	Creates a new shadow, allowing shadowing to continue if the primary shadow becomes unavailable or if the shadow replaces the database due to disk failure.
"*<filespec>*"	Explicit path name and file name for the shadow file. Shadow file specifications cannot include a node name.
LENGTH [=] *int* [PAGE[S]]	Length in database pages of an additional shadow file. Page size is determined by page size of the database itself.
<secondary_file>	Specifies the length of a primary or secondary shadow file. Use for primary file only if defining a secondary file in the same statement.
STARTING [AT [PAGE]] *int*	Starting page number at which a secondary shadow file begins.

Description

CREATE SHADOW is used to guard against loss of access to a database by establishing one or more copies of the database on secondary storage devices.

Each copy of the database consists of one or more shadow files, referred to as a *shadow set*. Each shadow set is designated by a unique positive integer.

Disk shadowing has three components:

- A database to shadow.
- The RDB$FILES system table, which lists shadow files and other information about the database.
- A shadow set, consisting of one or more shadow files.

When CREATE SHADOW is issued, a shadow is established for the database most recently attached by an application. A shadow set can consist of one or multiple files. In case of disk failure, the database administrator (DBA) activates the disk shadow so that it can take the place of the database. If CONDITIONAL is specified, then when the DBA activates the disk shadow to replace an actual database, a new shadow is established for the database.

If a database is larger than the space available for a shadow on one disk, use the <*secondary_file*> option to define multiple shadow files. Multiple shadow files can be spread over several disks.

Tip To add a secondary file to an existing disk shadow, drop the shadow with DROP SHADOW and use CREATE SHADOW to recreate it with the desired number of files.

Examples The following **isql** statement creates a single, automatic shadow file for *employee.gdb*:

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

The next **isql** statement creates a conditional, single, automatic shadow file for *employee.gdb*:

```
CREATE SHADOW 2 CONDITIONAL "employee.shd" LENGTH 1000;
```

The following **isql** statements create a multiple-file shadow set for the *employee.gdb* database. The first statement specifies starting pages for the shadow files; the second statement specifies the number of pages for the shadow files.

```
CREATE SHADOW 3 AUTO
      "employee.sh1"
   FILE "employee.sh2"
         STARTING AT PAGE 1000
   FILE "employee.sh3"
         STARTING AT PAGE 2000;

CREATE SHADOW 4 MANUAL "employee.sdw"
   LENGTH 1000
   FILE "employee.sh1"
```

```
        LENGTH 1000
    FILE "employee.sh2";
```

See Also DROP SHADOW

For more information about using shadows, see the *Windows Client User's Guide*.

CREATE TABLE

Creates a new table in an existing database. Available in SQL, DSQL, and **isql**.

Syntax

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

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in **isql**, the semicolon is a terminating symbol for the statement, so it must be included.

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

Note The COLLATE clause cannot be specified for BLOB columns.

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

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

Note Outermost brackets, in bold, must be included when declaring arrays.

```
<expr> = A valid SQL expression that results in a single value.

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

<constraint_def> = {UNIQUE | PRIMARY KEY
    | CHECK (<search_condition>)
```

```
    | REFERENCES other_table [(other_col [, other_col ...])]}

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

<tconstraint_def> = {{PRIMARY KEY | UNIQUE} (col [, col ...])
    | FOREIGN KEY (col [, col ...]) REFERENCES other_table
    | 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>}

<val> = {
    col [<array_dim>] | :variable
    | <constant> | <expr> | <function>
    | udf ([<val> [, <val> ...]])
    | NULL | USER | RDB$DB_KEY | ?
    } [COLLATE collation]
```

Note In SQL and **isql**, <*val*> cannot be a parameter placeholder (**?**). In DSQL and **isql**, <*val*> cannot be a variable. <*expr*> is any complex SQL statement or equation that produces a single value.

```
<constant> = num | "string" | charsetname "string"

<function> = {
    COUNT (* | [ALL] <val> | DISTINCT <val>)
    | SUM ([ALL] <val> | DISTINCT <val>)
    | AVG ([ALL] <val> | DISTINCT <val>)
    | MAX ([ALL] <val> | DISTINCT <val>)
    | MIN ([ALL] <val> | DISTINCT <val>)
    | CAST (<val> AS <datatype>)
    | UPPER (<val>)
    | GEN_ID (generator, <val>)
    }
```

```
<operator> = {= | < | > | <= | >= | !< | !> | <> | !=}

<select_one> = SELECT on a single column that returns exactly one
    value.

<select_list> = SELECT on a single column that returns zero or more
    values.

<select_expr> = SELECT on a list of values that returns zero or more
    values.
```

Argument	Description
table	Name for the table. Table name must be unique among table and procedure names in the database.
EXTERNAL [FILE] "*<filespec>*"	Declares that data for the table to create resides in a table or file outside the database. *<filespec>* is the complete file specification of the external file or table.
col	Name for the table column; unique among column names in the table.
<datatype>	SQL data type for a column.
COMPUTED [BY] (*<expr>*)	Bases a column definition on an expression. The expression must return a single value, and cannot be an array or return an array. *<expr>* is any arithmetic expression that is valid for the data types in the columns.
domain	Name of an existing domain.
COLLATE *collation*	Specifies the collation order for the column or value. Collation order set at column level overrides domain level collation order.
DEFAULT	Specifies a default column value that is entered when no other entry is made. Values: • *literal*—Inserts a specified string, numeric value, or date value. • NULL—Enters a NULL value. • USER—Enters the user name of the current user. Column must be of compatible text type to use the default. Defaults set at column level override defaults set at the domain level.
CONSTRAINT *constraint*	Places a named constraint on a table or column. A constraint is a rule applied to a table's structure or contents. If this clause is omitted, InterBase generates a system name for the constraint.

Description

CREATE TABLE establishes a new table, its columns, and integrity constraints in an existing database. The user who creates a table is the table's owner and has all privileges for it, including the ability to GRANT privileges to other users, triggers, and stored procedures.

CREATE TABLE supports several options for defining columns:

- Local columns specify the name and data type for data entered into the column.
- Computed columns are based on an expression. Column values are computed each time the table is accessed. If the data type is not specified, InterBase calculates an appropriate one. Columns referenced in the expression must exist before the column can be defined.
- Domain-based columns inherit all the characteristics of a domain, but the column definition can include a new default value, a NOT NULL attribute, additional CHECK constraints, or a collation clause that overrides the domain definition. It can also include additional column constraints.
- 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 the single column. Otherwise, the column uses the default database character set. If the database character set is changed, all columns subsequently defined have the new character set, but existing columns are not affected. For a complete list of character sets recognized by InterBase, see Appendix D: “Character Sets and Collation Orders.”
- 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. In this case, no transliteration is performed between the source and destination character sets, and errors may occur during assignment.
- The COLLATE clause enables specification of a particular collation order for CHAR, VARCHAR, and BLOB text data types. Choice of collation order is restricted to those supported for the column’s given character set, which is either the default character set for the entire database, or a different set defined in the CHARACTER SET clause as part of the data type definition. For a complete list of collation orders recognized by InterBase, see Appendix D: “Character Sets and Collation Orders.”

NOT NULL is an attribute that prevents the entry of NULL or unknown values in column. NOT NULL affects all INSERT and UPDATE operations on a column.

Integrity constraints can be defined for a table when it is created. Integrity constraints are rules that control the database and its contents, enforcing column-to-table and table-to-table relationships, and validating data entries. They span all

transactions that access the database and are automatically maintained by the system. CREATE TABLE can create the following types of integrity constraints:

- PRIMARY KEY constraints are unique identifiers for each row in a table. Values in this column or ordered set of columns cannot occur in more than one row. A PRIMARY KEY column must also define the NOT NULL attribute. A table can have only one PRIMARY KEY that can be defined on one or more columns.
- UNIQUE keys ensure that no two rows have the same value for a specified column or ordered set of columns. A unique column must also define the NOT NULL attribute. A table can have one or more UNIQUE keys. A UNIQUE key can be referenced by a FOREIGN KEY in another table.
- Referential constraints ensure that values in a set of columns that define a FOREIGN KEY are the same as values in UNIQUE or PRIMARY KEY columns in a referenced table. Before the REFERENCES constraint can be added, the UNIQUE or PRIMARY KEY columns it references must already be defined in the referenced table.
- CHECK constraints enforce a <*search_condition*> that must be true for inserts or updates to the specified table. <*search_condition*> can require a certain combination or range of values or compare the value entered with data in other columns.

Creating PRIMARY KEY and FOREIGN KEY constraints requires exclusive access to the database.

For unnamed constraints, the system assigns a unique constraint name stored in the system table RDB$RELATION_CONSTRAINTS.

Note Constraints are not enforced on expressions.

Important A DECLARE TABLE must precede CREATE TABLE in embedded applications if the same SQL program both creates a table and inserts data in the table.

The EXTERNAL FILE option creates a table whose data resides in an external file, rather than in the InterBase database. Use this option to:

- Define an InterBase table composed of data from an external source, such as data in files managed by other operating systems or in non-database applications.
- Transfer data to an existing InterBase table from an external file.

Examples The following **isql** statement creates a simple table with a PRIMARY KEY:

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

The next **isql** statement creates both a column-level and a table-level UNIQUE constraint:

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

The following **isql** statement illustrates table-level PRIMARY KEY, FOREIGN KEY, and CHECK constraints. The PRIMARY KEY constraint is based on three columns. This example also illustrates creating an array column of VARCHAR.

```
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),
   CHECK (MIN_SALARY < MAX_SALARY)
);
```

The next **isql** statement creates a table with a calculated column:

```
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)
);
```

In the following **isql** statement the first column retains the default collating order for the database's default character set. The second column has a different collating order, and the third column definition includes a character set and a collating order.

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

See Also CREATE DOMAIN, DECLARE TABLE, GRANT, REVOKE

For more information on creating metadata, using integrity constraints, external tables, data types, collation order, and character sets, see the *Data Definition Guide*.

CREATE TRIGGER

Creates a trigger, including when it fires, and what actions it performs. Available in DSQL, and **isql**.

Syntax

```
CREATE TRIGGER name FOR table
    [ACTIVE | INACTIVE]
    {BEFORE | AFTER}
    {DELETE | INSERT | UPDATE}
    [POSITION number]
    AS <trigger_body> terminator

<trigger_body> =
    [<variable_declaration_list>] <block>

<variable_declaration_list> =
    DECLARE VARIABLE variable <datatype>;
    [DECLARE VARIABLE variable <datatype>; ...]

<block> =
BEGIN
    <compound_statement>
    [<compound_statement> ...]
END

<compound_statement> = {<block> | statement;}

<datatype> = {
    {SMALLINT | INTEGER | FLOAT | DOUBLE PRECISION}
    | {DECIMAL | NUMERIC} [(precision [, scale])]
    | DATE | {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR}
    [(int)] [CHARACTER SET charname]
```

```
| {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
[VARYING] [(int)]}
```

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 random order.
DECLARE VARIABLE *var <datatype>*	Declares local variables used only in the trigger. Each declaration must be preceded by DECLARE VARIABLE and followed by a semicolon (;). • *var*: Local variable name, unique in the trigger. • *<datatype>*: The 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. Used in **isql** only.

Description

CREATE TRIGGER defines a new trigger to a database. A trigger is a self-contained program 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 automatically execute, or *fire*. Triggers defined for UPDATE on non-updatable views fire even if no update occurs.

A trigger is composed of a *header* and a *body*.

The trigger header contains:

- A *trigger name*, unique within the database, that distinguishes the trigger from all others.
- 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.

Important Because each statement in the trigger body must be terminated by a semicolon, you must define a different symbol to terminate the trigger body itself. In **isql**, include a SET TERM statement before CREATE TRIGGER to specify a terminator other than a semicolon. After the body of the trigger, include another SET TERM to change the terminator back to a semicolon.

A trigger is associated with a table. The table owner and any user granted privileges to the table automatically have rights to execute associated triggers.

Triggers can be granted privileges on tables, just as users or procedures can be granted privileges. Use the GRANT statement, but instead of using TO *username*, use TO TRIGGER *trigger_name*. Triggers' privileges can be revoked similarly using REVOKE.

When a user performs an action that fires a trigger, the trigger will have privileges to perform its actions if one of the following conditions is true:

- The trigger has privileges for the action.
- The user has privileges for the action.

InterBase procedure and trigger language is a complete programming language for stored procedures and triggers. It includes:

- SQL data manipulation statements: INSERT, UPDATE, DELETE, and singleton SELECT.

- SQL operators and expressions, including UDFs linked with the calling application and generators.
- Powerful extensions to SQL, including assignment statements, control-flow statements, context variables, event-posting statements, exceptions, and error-handling statements.

The following table summarizes language extensions for triggers. For a complete description of each statement, see Chapter 3: "Procedure and Trigger Language Reference."

Table 2-8: 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* */	Programmer's comment, where *comment_text* can be any number of lines of text.
EXCEPTION *exception_name*	Raises the named exception. An exception is a user-defined error, which returns an error message to the calling application unless handled by a WHEN statement.
EXECUTE PROCEDURE *proc_name* [*var* [, *var* ...]] [RETURNING_VALUES *var* [, *var* ...]]	Executes stored procedure, *proc_name,* with the listed input arguments, returning values in the listed output arguments following RETURNING_VALUES. Input and output arguments must be local variables.
EXIT	Jumps to the final END statement in the procedure.
FOR *<select_statement>* DO *<compound_statement>*	Repeats the statement or block following DO for every qualifying row retrieved by *<select_statement>*. *<select_statement>*: a normal SELECT statement, except the INTO clause is required and must come last.
<compound_statement>	Either a single statement in procedure and trigger language or a block of statements bracketed by BEGIN and END.

Table 2-8: Procedure and Trigger Language Extensions (Continued)

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.
NEW.*column*	New context variable that indicates a new column value in an INSERT or UPDATE operation.
OLD.*column*	Old context variable that indicates a column value before an UPDATE or DELETE operation.
POST_EVENT *event_name* | *col*	Posts the event, *event_name*, or uses the value in *col* as an event name.
WHILE (*<condition>*) DO *<compound_statement>*	While *<condition>* is TRUE, keep performing *<compound_statement>*. First *<condition>* is tested, and if it is TRUE, then *<compound_statement>* is performed. This sequence is repeated until *<condition>* is no longer TRUE.
WHEN {*<error>* [, *<error>* ...] | ANY} DO *<compound_statement>*	Error-handling statement. When one of the specified errors occurs, performs *<compound_statement>*. WHEN statements, if present, must come at the end of a block, just before END. • *<error>*: EXCEPTION *exception_name,* SQLCODE *errcode* or GDSCODE *number.* • ANY: Handles any errors.

Examples

The following trigger, SAVE_SALARY_CHANGE, makes correlated updates to the SALARY_HISTORY table when a change is made to an employee's salary in the EMPLOYEE table:

```
SET TERM !! ;
CREATE TRIGGER SAVE_SALARY_CHANGE FOR EMPLOYEE
AFTER UPDATE AS
BEGIN
   IF (OLD.SALARY <> NEW.SALARY) THEN
   INSERT INTO SALARY_HISTORY
      (EMP_NO, CHANGE_DATE, UPDATER_ID, OLD_SALARY, PERCENT_CHANGE)
      VALUES (OLD.EMP_NO, "now", USER,OLD.SALARY,
      (NEW.SALARY - OLD.SALARY) * 100 / OLD.SALARY);
END !!
SET TERM ; !!
```

The following trigger, SET_CUST_NO, uses a generator to create unique customer numbers when a new customer record is inserted in the CUSTOMER table:

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

The following trigger, POST_NEW_ORDER, posts an event named “new_order” whenever a new record is inserted in the SALES table:

```
SET TERM !! ;
CREATE TRIGGER POST_NEW_ORDER FOR SALES
AFTER INSERT AS
BEGIN
   POST_EVENT "new_order";
END !!
SET TERM ; !!
```

The following four fragments of trigger headers demonstrate how the POSITION option determines trigger firing order:

```
CREATE TRIGGER A FOR accounts
   BEFORE UPDATE
   POSITION 5 . . . /*Trigger body follows*/

CREATE TRIGGER B FOR accounts
   BEFORE UPDATE
   POSITION 0 . . . /*Trigger body follows*/

CREATE TRIGGER C FOR accounts
   AFTER UPDATE
   POSITION 5 . . . /*Trigger body follows*/

CREATE TRIGGER D FOR accounts
   AFTER UPDATE
   POSITION 3 . . . /*Trigger body follows*/
```

When this update takes place:

```
UPDATE accounts SET account_status = "on_hold"
   WHERE account_balance <0;
```

The triggers fire in this order:

1. Trigger B fires.
2. Trigger A fires.
3. The update occurs.

4. Trigger D fires.
5. Trigger C fires.

See Also ALTER EXCEPTION, ALTER TRIGGER, CREATE EXCEPTION, CREATE PROCEDURE, DROP EXCEPTION, DROP TRIGGER, EXECUTE PROCEDURE

For more information on creating and using triggers, see the *Data Definition Guide*.

For a complete description of the statements in procedure and trigger language, see Chapter 3: "Procedure and Trigger Language Reference."

CREATE VIEW

Creates a new view of data from one or more tables. Available in SQL, DSQL, and **isql**.

Syntax

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

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in **isql**, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
name	Name for the view. Must be unique among all view, table, and procedure names in the database.
view_col	Names the columns for the view. Column names must be unique among all column names in the view. Required if the view includes columns based on expressions; otherwise optional. Default: Column name from the underlying table.
<select>	Specifies the selection criteria for rows to be included in the view.
WITH CHECK OPTION	Prevents INSERT or UPDATE operations on an updatable view if the INSERT or UPDATE violates the search condition specified in the WHERE clause of the view's *<select>*.

Description CREATE VIEW describes a view of data based on one or more underlying tables in the database. The rows to return are defined by a SELECT statement that lists columns from the source tables. Only the view definition is stored in the database; a view does not directly represent physically stored data. It is possible to perform select, project, join, and union operations on views 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 may have privileges to a view without having access to its base tables. When creating views:

- A read-only view requires SELECT privileges for any underlying tables.
- An updatable view requires ALL privileges to the underlying tables.

The *view_col* option ensures that the view always contains the same columns and that the columns always have the same view-defined names.

View column names correspond in order and number to the columns listed in the <*select*>, so specify *all* view column names or *none*.

A *view_col* definition can contain one or more columns based on an expression that combines the outcome of two columns. The expression must return a single value, and cannot return an array or array element. If the view includes an expression, the *view-column* option is required.

Note Any columns used in the value expression must exist before the expression can be defined.

A SELECT statement clause cannot include the ORDER BY clause.

When SELECT * is used rather than a column list, order of display is based on the order in which columns are stored in the base table.

WITH CHECK OPTION enables InterBase to verify that a row added to or updated in a view is able to be seen through the view before allowing the operation to succeed. Do not use WITH CHECK OPTION for read-only views.

Note DSQL does not support view definitions containing UNION clauses. To create such a view, use embedded SQL.

A view is updatable if:

- It is a subset of a single table or another updatable view.
- All base table columns excluded from the view definition allow NULL values.
- The view's SELECT statement does not contain subqueries, a DISTINCT predicate, a HAVING clause, aggregate functions, joined tables, user-defined functions, or stored procedures.

If the view definition does not meet 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.

Examples The following **isql** statement creates an updatable view:

```
CREATE VIEW SNOW_LINE (CITY, STATE, SNOW_ALTITUDE) AS
   SELECT CITY, STATE, ALTITUDE
   FROM CITIES
   WHERE ALTITUDE > 5000;
```

The next **isql** statement uses a nested query to create a view:

```
CREATE VIEW RECENT_CITIES AS
   SELECT STATE, CITY, POPULATION FROM CITIES WHERE STATE IN
   (SELECT STATE FROM STATES WHERE STATEHOOD > "1-JAN-1850");
```

In an updatable view, the WITH CHECK OPTION prevents any inserts or updates through the view that do not satisfy the WHERE clause of the CREATE VIEW SELECT statement:

```
CREATE VIEW HALF_MILE_CITIES AS
   SELECT CITY, STATE, ALTITUDE FROM CITIES
         WHERE ALTITUDE > 2500
   WITH CHECK OPTION;
```

The WITH CHECK OPTION clause in the view would prevent the following insertion:

```
INSERT INTO HALF_MILE_CITIES (CITY, STATE, ALTITUDE)
   VALUES ("Chicago", "Illinois", 250);
```

On the other hand, the following UPDATE would be permitted:

```
INSERT INTO HALF_MILE_CITIES (CITY, STATE, ALTITUDE)
   VALUES ("Truckee", "California", 2736);
```

The WITH CHECK OPTION clause does not allow updates through the view which change the value of a row so that the view cannot retrieve it. For example, the WITH CHECK OPTION in the HALF_MILE_CITIES view prevents the following update:

```
UPDATE HALF_MILE_CITIES
   SET ALTITUDE = 2000
   WHERE STATE = "NY";
```

The next **isql** statement creates a view that joins two tables, and so is 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;
```

See Also CREATE TABLE, DROP VIEW, GRANT, INSERT, REVOKE, SELECT, UPDATE

For a complete discussion of views, see the *Data Definition Guide*.

DECLARE CURSOR

Defines a cursor for a table by associating a name with the set of rows specified in a SELECT statement. Available in SQL and DSQL.

Syntax

SQL form:

```
DECLARE cursor CURSOR FOR <select> [FOR UPDATE OF <col> [, <col>...]];
```

DSQL form:

```
DECLARE cursor CURSOR FOR <statement_id>
```

BLOB form:

See DECLARE CURSOR (BLOB)

Argument	Description
cursor	Name for the cursor.
<select>	Determines which rows to retrieve. SQL only.
FOR UPDATE OF *<col>* [, *<col>* ...]	Enables UPDATE and DELETE of specified column for retrieved rows.
<statement_id>	SQL statement name of a previously prepared statement, which, in this case, must be a SELECT statement. DSQL only.

Description

DECLARE CURSOR defines the set of rows that can be retrieved using the cursor it names. It is the first member of a group of table cursor statements that must be used in sequence.

<select> specifies a SELECT statement that determines which rows to retrieve. The SELECT statement cannot include INTO or ORDER BY clauses.

The FOR UPDATE OF clause is necessary for updating or deleting rows using the WHERE CURRENT OF clause with UPDATE and DELETE.

A cursor is a one-way pointer into the ordered set of rows retrieved by the select expression in the DECLARE CURSOR statement. It enables sequential access to retrieved rows in turn. There are four related cursor statements:

Stage	Statement	Purpose
1	DECLARE CURSOR	Declares the cursor. The SELECT statement determines rows retrieved for the cursor.
2	OPEN	Retrieves the rows specified for retrieval with DECLARE CURSOR. The resulting rows become the cursor's *active set*.

Stage	Statement	Purpose
3	FETCH	Retrieves the current row from the active set, starting with the first row. Subsequent FETCH statements advance the cursor through the set.
4	CLOSE	Closes the cursor and releases system resources.

Examples

The following embedded SQL statement declares a cursor with a search condition:

```
EXEC SQL
   DECLARE C CURSOR FOR
   SELECT CUST_NO, ORDER_STATUS
   FROM SALES
   WHERE ORDER_STATUS IN ("open", "shipping");
```

The next DSQL statement declares a cursor for a previously prepared statement, QUERY1:

```
DECLARE Q CURSOR FOR QUERY1
```

See Also CLOSE, DECLARE CURSOR (BLOB), FETCH, OPEN, PREPARE, SELECT

DECLARE CURSOR (BLOB)

Declares a BLOB cursor for read or insert. Available in SQL.

Syntax

```
DECLARE cursor CURSOR FOR
    {READ BLOB column FROM table
    | INSERT BLOB column INTO table}
    [FILTER [FROM subtype] TO subtype]
    [MAXIMUM_SEGMENT length];
```

Argument	Description
cursor	Name for the BLOB cursor.
column	Name of the BLOB column.
table	Table name.
READ BLOB	Declares a read operation on the BLOB.
INSERT BLOB	Declares a write operation on the BLOB.
[FILTER [FROM *subtype*] TO *subtype*]	Specifies optional BLOB filters used to translate a BLOB from one user-specified format to another. *subtype* determines which filters are used for translation.
MAXIMUM_ SEGMENT *length*	Length of the local variable to receive the BLOB data after a FETCH operation.

Description Declares a cursor for reading or inserting BLOB data. A BLOB cursor can be associated with only one BLOB column.

To read partial BLOB segments when a host-language variable is smaller than the segment length of a BLOB, declare the BLOB cursor with the MAXIMUM_SEGMENT clause. If *length* is less than the BLOB segment, FETCH returns *length* bytes. If the same or greater, it returns a full segment (the default).

Examples The following embedded SQL statement declares a READ BLOB cursor and uses the MAXIMUM_SEGMENT option:

```
EXEC SQL
   DECLARE BC CURSOR FOR
   READ BLOB JOB_REQUIREMENT FROM JOB MAXIMUM_SEGMENT 40;
```

The next embedded SQL statement declares an INSERT BLOB cursor:

```
EXEC SQL
   DECLARE BC CURSOR FOR
   INSERT BLOB JOB_REQUIREMENt INTO JOB;
```

See Also CLOSE (BLOB), FETCH (BLOB), INSERT CURSOR(BLOB), OPEN (BLOB)

DECLARE EXTERNAL FUNCTION

Declares an existing user-defined function (UDF) to the database. Available in SQL, DSQL, and **isql**.

Syntax

```
DECLARE EXTERNAL FUNCTION name [<datatype> | CSTRING (int)
    [, <datatype> | CSTRING (int) ...]]
    RETURNS {<datatype> [BY VALUE] | CSTRING (int)}
    ENTRY_POINT "<entryname>"
    MODULE_NAME "<modulename>";
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in **isql**, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
name	Name of the UDF.
<datatype>	Data type of an input or return parameter. All input parameters are passed to a UDF by reference. Return parameters can be passed by value. Cannot be an array element.
RETURNS	Specifies the return value of a function.

Argument	Description
BY VALUE	Specifies that a return value should be passed by value, rather than by reference.
CSTRING (*int*)	Specifies a UDF that returns a null-terminated string *int* bytes in length.
"*<entryname>*"	Quoted string specifying the name of the UDF as stored in the UDF library.
"*<modulename>*"	Quoted file specification identifying the object module in which the UDF resides.

Description DECLARE EXTERNAL FUNCTION provides information about a UDF to the database: where to find it, its name, the input parameters it requires, and the single value it returns. A UDF exists in a library outside the database.

"*<entryname>*" is the actual name of the function as stored in the UDF library. It does not have to match the name of the UDF as stored in the database.

Important Do not use DECLARE EXTERNAL FUNCTION when creating a database on a NetWare server. UDF libraries cannot be created or used on NetWare servers.

Example The following **isql** statement declares the function, TOPS(), to a database:

```
DECLARE EXTERNAL FUNCTION TOPS
   CHAR(256), INTEGER, BLOB
   RETURNS INTEGER BY VALUE
   ENTRY_POINT "te1" MODULE_NAME "tm1";
```

See Also DROP EXTERNAL FUNCTION

For more information about writing UDFs, see the *Data Definition Guide*.

For more information about using UDFs in embedded applications, see the *Programmer's Guide*.

DECLARE FILTER

Declares an existing BLOB filter to a database. Available in SQL, DSQL, and **isql**.

Syntax

```
DECLARE FILTER filter
    INPUT_TYPE subtype OUTPUT_TYPE subtype
    ENTRY_POINT "<entryname>" MODULE_NAME "<modulename>";
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in **isql**, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
filter	Name of the filter. Must be unique among filter names in the database.
INPUT_TYPE *subtype*	Specifies the BLOB subtype from which data is to be converted.
OUTPUT_TYPE *subtype*	Specifies the BLOB subtype into which data is to be converted.
"*<entryname>*"	Quoted string specifying the name of the BLOB filter as stored in a linked library.
"*<modulename>*"	Quoted file specification identifying the object module in which the filter is stored.

Description

DECLARE FILTER provides information about an existing BLOB filter to the database: where to find it, its name, and the BLOB subtypes it works with. A BLOB filter is a user-written program that converts data stored in BLOB columns from one subtype to another.

INPUT_TYPE and OUTPUT_TYPE together determine the behavior of the BLOB filter. Each filter declared to the database should have a unique combination of INPUT_TYPE and OUTPUT_TYPE integer values. InterBase provides a built-in type of 1, for handling text. User-defined types must be expressed as negative values.

"*<entryname>*" is the name of the BLOB filter stored in the library. When an application uses a BLOB filter, it calls the filter function with this name.

Important Do not use DECLARE FILTER when creating a database on a NetWare server. BLOB filters cannot be created or used on NetWare servers.

Example

The following **isql** statement declares a BLOB filter:

```
DECLARE FILTER DESC_FILTER
INPUT_TYPE 1
OUTPUT_TYPE -4
ENTRY_POINT "desc_filter"
MODULE_NAME "FILTERLIB";
```

See Also

DROP FILTER

For instructions on writing BLOB filters, see the *Programmer's Guide*.

For more information about BLOB subtypes, see the *Data Definition Guide*.

DECLARE STATEMENT

Identifies dynamic SQL statements before they are prepared and executed in an embedded program. Available in SQL.

Syntax

```
DECLARE <statement> STATEMENT;
```

Argument	Description
<statement>	Name of an SQL variable for a user-supplied SQL statement to prepare and execute at run time.

Description DECLARE STATEMENT names an SQL variable for a user-supplied SQL statement to prepare and execute at run time. DECLARE STATEMENT is not executed, so it does not produce run-time errors. The statement provides internal documentation.

Example The following embedded SQL statement declares Q1 to be the name of a string for preparation and execution.

```
EXEC SQL
   DECLARE Q1 STATEMENT;
```

See Also EXECUTE, EXECUTE IMMEDIATE, PREPARE

DECLARE TABLE

Describes the structure of a table to the preprocessor, **gpre**, before it is created with CREATE TABLE. Available in SQL.

Syntax

```
DECLARE table TABLE (<table_def>);
```

Argument	Description
table	Name of the table that will be created. Table names must be unique within the database.
<table_def>	Definition of the table. For complete table definition syntax, see CREATE TABLE.

Description DECLARE TABLE causes **gpre** to store a table description. A table declaration is required if a table is both created and populated with data in the same program. If the declared table already exists in the database or if the declaration contains syntax errors, **gpre** returns an error.

When a table is referenced at run time, the column descriptions and data types are checked against the description stored in the database. If the table description is not in the database and the table is not declared, or if column descriptions and data types do not match, the application returns an error.

DECLARE TABLE can include an existing domain in a column definition, but must give the complete column description if the domain is not defined at compile time.

DECLARE TABLE cannot include integrity constraints and column attributes, even if they are present in a subsequent CREATE TABLE statement.

Important DECLARE TABLE cannot appear in a program that accesses multiple databases.

Examples The following embedded SQL statements declare and create a table:

```
EXEC SQL
   DECLARE STOCK TABLE
   (MODEL SMALLINT,
   MODELNAME CHAR(10),
   ITEMID INTEGER);

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

See Also CREATE DOMAIN, CREATE TABLE

DELETE

Removes rows in a table or in the active set of a cursor. Available in SQL, DSQL, and **isql**.

Syntax SQL and DSQL form:

```
DELETE [TRANSACTION transaction] FROM table
    {[WHERE <search_condition>] | WHERE CURRENT OF cursor};
```

Important Omit the terminating semicolon for DSQL.

<search_condition> = Search condition as specified in SELECT.

isql form:

```
DELETE FROM TABLE [WHERE <search_condition>];
```

Argument	Description
TRANSACTION *transaction*	Name of the transaction under control of which the statement is executed. SQL only.
table	Name of the table from which to delete rows.
WHERE *<search_condition>*	Search condition that specifies the rows to delete. Without this clause, DELETE affects all rows in the specified table or view.
WHERE CURRENT OF *cursor*	Specifies that the current row in the active set of *cursor* is to be deleted.

Description

DELETE specifies one or more rows to delete from a table or updatable view. DELETE is one of the database privileges controlled by the GRANT and REVOKE statements.

The TRANSACTION clause can be used in multiple transaction SQL applications to specify which transaction controls the DELETE operation. The TRANSACTION clause is not available in DSQL or **isql**.

For searched deletions, the optional WHERE clause can be used to restrict deletions to a subset of rows in the table.

Caution Without a WHERE clause, a searched delete removes all rows from a table.

When performing a positioned delete with a cursor, the WHERE CURRENT OF clause must be specified to delete one row at a time from the active set.

Examples

The following **isql** statement deletes all rows in a table:

```
DELETE FROM EMPLOYEE_PROJECT;
```

The next embedded SQL statement is a searched delete in an embedded application. It deletes all rows where a host-language variable equals a column value.

```
EXEC SQL
   DELETE FROM SALARY_HISTORY
   WHERE EMP_NO = :emp_num;
```

The following embedded SQL statements use a cursor and the WHERE CURRENT OF option to delete rows from CITIES with a population less than the host variable, *min_pop*. They declare and open a cursor that finds qualifying cities, fetch rows into the cursor, and delete the current row pointed to by the cursor.

```
EXEC SQL
   DECLARE SMALL_CITIES CURSOR FOR
   SELECT CITY, STATE
   FROM CITIES
```

```
   WHERE POPULATION < :min_pop;
EXEC SQL
   OPEN SMALL_CITIES;
EXEC SQL
   FETCH SMALL_CITIES INTO :cityname, :statecode;
WHILE (!SQLCODE)
{
EXEC SQL
   DELETE FROM CITIES
   WHERE CURRENT OF SMALL_CITIES;
EXEC SQL
   FETCH SMALL_CITIES INTO :cityname, :statecode;
}
EXEC SQL
   CLOSE SMALL_CITIES;
```

See Also DECLARE CURSOR, FETCH, GRANT, OPEN, REVOKE, SELECT

For more information about using cursors, see the *Programmer's Guide*.

DESCRIBE

Provides information about columns that are retrieved by a dynamic SQL (DSQL) statement, or information about dynamic parameters that statement passes. Available in SQL.

Syntax

```
DESCRIBE [OUTPUT | INPUT] statement
    {INTO | USING} SQL DESCRIPTOR xsqlda;
```

Argument	Description
OUTPUT	Indicates that column information should be returned in the XSQLDA (default).
INPUT	Indicates that dynamic parameter information should be stored in the XSQLDA.
statement	A previously defined alias for the statement to DESCRIBE. Aliases are defined with PREPARE.
{INTO | USING} SQL DESCRIPTOR *xsqlda*	Specifies the XSQLDA to use for the DESCRIBE statement.

Description DESCRIBE has two uses:

- As a *describe output statement*, DESCRIBE stores into an XSQLDA a description of the columns that make up the select list of a previously prepared statement. If the PREPARE statement included an INTO clause, it is unnecessary to use DESCRIBE as an output statement.

- As a *describe input statement*, DESCRIBE stores into an XSQLDA a description of the dynamic parameters that are in a previously prepared statement.

DESCRIBE is one of a group of statements that process DSQL statements.

Statement	Purpose
PREPARE	Readies a DSQL statement for execution.
DESCRIBE	Fills in the XSQLDA with information about the statement.
EXECUTE	Executes a previously prepared statement.
EXECUTE IMMEDIATE	Prepares a DSQL statement, executes it once, and discards it.

Separate DESCRIBE statements must be issued for input and output operations. The INPUT keyword must be used to store dynamic parameter information.

Important When using DESCRIBE for output, if the value returned in the *sqld* field in the XSQLDA is larger than the *sqln* field, you must:

- Allocate more storage space for XSQLVAR structures.
- Reissue the DESCRIBE statement.

Note The same XSQLDA structure can be used for input and output if desired.

Example The following embedded SQL statement retrieves information about the output of a SELECT statement:

```
EXEC SQL
   DESCRIBE Q INTO xsqlda
```

The next embedded SQL statement stores information about the dynamic parameters passed with a statement to be executed:

```
EXEC SQL
   DESCRIBE INPUT Q2 USING SQL DESCRIPTOR xsqlda;
```

See Also EXECUTE, EXECUTE IMMEDIATE, PREPARE

For more information about DSQL programming and the XSQLDA, see the *Programmer's Guide*.

DISCONNECT

Detaches an application from a database. Available in SQL.

Syntax

```
DISCONNECT {{ALL | DEFAULT} | dbhandle [, dbhandle] ...]};
```

Argument	Description
ALL | DEFAULT	Either keyword detaches all open databases.
dbhandle	Previously declared database handle specifying a database to detach.

Description

DISCONNECT closes a specific database identified by a database handle or all databases, releases resources used by the attached database, zeroes database handles, commits the default transaction if the **gpre -manual** option is not in effect, and returns an error if any non-default transaction is not committed.

Before using DISCONNECT, commit or roll back the transactions affecting the database to be detached.

To reattach to a database closed with DISCONNECT, reopen it with a CONNECT statement.

Examples

The following embedded SQL statements close all databases:

```
EXEC SQL
   DISCONNECT DEFAULT;

EXEC SQL
   DISCONNECT ALL;
```

The next embedded SQL statements close the databases identified by their handles:

```
EXEC SQL
   DISCONNECT DB1;

EXEC SQL
   DISCONNECT DB1, DB2;
```

See Also

COMMIT, CONNECT, ROLLBACK, SET DATABASE

DROP DATABASE

Deletes the currently attached database. Available in **isql**.

Syntax

```
DROP DATABASE;
```

Description

DROP DATABASE deletes the currently attached database, including any associated secondary, 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.

Example

The following **isql** statement deletes the current database:

```
DROP DATABASE;
```

See Also

ALTER DATABASE, CREATE DATABASE

DROP DOMAIN

Deletes a domain from a database. Available in SQL, DSQL, and **isql**.

Syntax

```
DROP DOMAIN name;
```

Important

In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in **isql**, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
name	Name of an existing domain.

Description

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, use ALTER TABLE to delete the columns based on the domain before executing DROP DOMAIN.

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

Example The following **isql** statement deletes a domain:

```
DROP DOMAIN COUNTRYNAME;
```

See Also ALTER DOMAIN, ALTER TABLE, CREATE DOMAIN

DROP EXCEPTION

Deletes an exception from a database. Available in DSQL and **isql**.

Syntax `DROP EXCEPTION` *name*

Argument	Description
name	Name of an existing exception message.

Description DROP EXCEPTION removes an exception from a database.

Exceptions used in existing procedures and triggers cannot be dropped.

Tip In **isql**, SHOW EXCEPTION displays a list of exceptions' *dependencies*, the procedures and triggers that use the exceptions.

An exception can be dropped by its creator, the SYSDBA user, and any user with operating system root privileges.

Example This **isql** statement drops an exception:

```
DROP EXCEPTION UNKNOWN_EMP_ID;
```

See Also ALTER EXCEPTION, ALTER PROCEDURE, ALTER TRIGGER, CREATE EXCEPTION, CREATE PROCEDURE, CREATE TRIGGER

DROP EXTERNAL FUNCTION

Removes a user-defined function (UDF) declaration from a database. Available in SQL, DSQL, and **isql**.

Syntax `DROP EXTERNAL FUNCTION` *name*;

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in **isql**, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
name	Name of an existing UDF.

Description DROP EXTERNAL FUNCTION deletes a UDF declaration from a database. Dropping a UDF declaration from a database does *not* remove it from the corresponding UDF library, but it does make the UDF inaccessible from the database. Once the definition is dropped, any applications that depend on the UDF will return run-time errors.

A UDF can be dropped by its declarer, the SYSDBA user, or any users with operating system root privileges.

Important UDFs are not available for databases on NetWare servers. If a UDF is accidentally declared for a database on a NetWare server, DROP EXTERNAL FUNCTION should be used to remove the declaration.

Example This **isql** statement drops a UDF:

```
DROP EXTERNAL FUNCTION TOPS;
```

See Also DECLARE EXTERNAL FUNCTION

DROP FILTER

Removes a BLOB filter declaration from a database. Available in SQL, DSQL, and **isql**.

Syntax `DROP FILTER` *`name`*`;`

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in **isql**, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
name	Name of an existing BLOB filter.

Description DROP FILTER removes a BLOB filter declaration from a database. Dropping a BLOB filter declaration from a database does *not* remove it from the corresponding BLOB filter library, but it does make the filter inaccessible from the database. Once the definition is dropped, any applications that depend on the filter will return run-time errors.

DROP FILTER fails and returns an error if any processes are using the filter.

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

Important BLOB filters are not available for databases on NetWare servers. If a BLOB filter is accidentally declared for a database on a NetWare server, DROP FILTER should be used to remove the declaration.

Example This **isql** statement drops a BLOB filter:

```
DROP FILTER DESC_FILTER;
```

See Also DECLARE FILTER

DROP INDEX

Removes an index from a database. Available in SQL, DSQL, and **isql**.

Syntax `DROP INDEX` *`name`*`;`

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in **isql**, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
name	Name of an existing index.

Description DROP INDEX removes a user-defined index from a database.

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

Important System-defined indexes, such as those for UNIQUE, PRIMARY KEY, and FOREIGN KEY integrity constraints, cannot be dropped.

An index in use is not dropped until it is no longer in use.

Example The following **isql** statement deletes an index:

```
DROP INDEX MINSALX;
```

See Also ALTER INDEX, CREATE INDEX

For more information about integrity constraints and system-defined indexes, see the *Data Definition Guide*.

DROP PROCEDURE

Deletes an existing stored procedure from a database. Available in DSQL, and **isql**.

Syntax

```
DROP PROCEDURE name
```

Argument	Description
name	Name of an existing stored procedure.

Description

DROP PROCEDURE removes an existing stored procedure definition from a database.

Procedures used by other procedures, triggers, or views cannot be dropped. Procedures currently in use cannot be dropped.

Tip In **isql**, SHOW PROCEDURE displays a list of procedures' *dependencies*, the procedures, triggers, exceptions, and tables that use the procedures.

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

Example

The following **isql** statement deletes a procedure:

```
DROP PROCEDURE GET_EMP_PROJ;
```

See Also

ALTER PROCEDURE, CREATE PROCEDURE, EXECUTE PROCEDURE

DROP SHADOW

Deletes a shadow from a database. Available in SQL, DSQL, and **isql**.

Syntax

```
DROP SHADOW set_num;
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in **isql**, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
set_num	Positive integer to identify an existing shadow set.

Description DROP SHADOW deletes a shadow set and detaches from the shadowing process. The **isql** SHOW DATABASE command can be used to see shadow set numbers for a database.

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

Example The following **isql** statement deletes a shadow set from its database:

```
DROP SHADOW 1;
```

See Also CREATE SHADOW

DROP TABLE

Removes a table from a database. Available in SQL, DSQL, and **isql**.

Syntax `DROP TABLE` *`name`*`;`

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in **isql**, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
name	Name of an existing table.

Description DROP TABLE removes a table's data, metadata, and indexes from a database. It also drops any triggers that reference the table.

A table referenced in an SQL expression, a view, integrity constraint, or stored procedure cannot be dropped. A table used by an active transaction is not dropped until it is free.

Note When used to drop an external table, DROP TABLE only removes the table definition from the database. The external file is not deleted.

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

Example The following embedded SQL statement drops a table:

```
EXEC SQL
   DROP TABLE COUNTRY;
```

See Also ALTER TABLE, CREATE TABLE

DROP TRIGGER

Deletes an existing user-defined trigger from a database. Available in DSQL and **isql**.

Syntax

```
DROP TRIGGER name
```

Argument	Description
name	Name of an existing trigger.

Description

DROP TRIGGER removes a user-defined trigger definition from the database. System-defined triggers, such as those created for CHECK constraints, cannot be dropped. Use ALTER TABLE to drop the CHECK clause that defines the trigger.

Triggers used by an active transaction cannot be dropped until the transaction is terminated.

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

Tip To inactivate a trigger temporarily, use ALTER TRIGGER and specify INACTIVE in the header.

Example

The following **isql** statement drops a trigger:

```
DROP TRIGGER POST_NEW_ORDER;
```

See Also

ALTER TRIGGER, CREATE TRIGGER

DROP VIEW

Removes a view definition from the database. Available in SQL, DSQL, and **isql**.

Syntax

```
DROP VIEW name;
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in **isql**, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
name	Name of an existing view definition to drop.

Description DROP VIEW enables a view's creator to remove a view definition from the database if the view is not used in another view, stored procedure, or CHECK constraint definition.

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

Example The following **isql** statement removes a view definition:

```
DROP VIEW PHONE_LIST;
```

See Also CREATE VIEW

END DECLARE SECTION

Identifies the end of a host-language variable declaration section. Available in SQL.

Syntax

```
END DECLARE SECTION;
```

Description END DECLARE SECTION is used in embedded SQL applications to identify the end of host-language variable declarations for variables that will be used in subsequent SQL statements.

Example The following embedded SQL statements declare a section, and single host-language variable:

```
EXEC SQL
   BEGIN DECLARE SECTION;
      BASED_ON EMPLOYEE.SALARY salary;
EXEC SQL
   END DECLARE SECTION;
```

See Also BASED ON, BEGIN DECLARE SECTION

EVENT INIT

Registers interest in one or more events with the InterBase event manager. Available in SQL.

Syntax

```
EVENT INIT request_name [<dbhandle>]
    [("<string>" | :<variable> [, "<string>" | :<variable> ...]);
```

Argument	Description
request_name	Application event handle.
<dbhandle>	Specifies the database to examine for occurrences of the events. If omitted, *<dbhandle>* defaults to the database named in the most recent SET DATABASE statement.
"*<string>*"	Unique name identifying an event associated with *event_name*.
:*<variable>*	Host-language character array containing a list of event names to associate with.

Description EVENT INIT is the first step in the InterBase two-part synchronous event mechanism:

1. EVENT INIT registers an application's interest in an event.
2. EVENT WAIT causes the application to wait until notified of the event's occurrence.

EVENT INIT registers an application's interest in a list of events in parentheses. The list should correspond to events posted by stored procedures or triggers in the database. If an application registers interest in multiple events with a single EVENT INIT, then when one of those events occurs, the application must determine which event occurred.

Events are posted by a POST_EVENT call within a stored procedure or trigger.

The event manager keeps track of events of interest. At commit time, when an event occurs, the event manager notifies interested applications.

Example The following embedded SQL statement registers interest in an event:

```
EXEC SQL
   EVENT INIT ORDER_WAIT EMPDB ("new_order");
```

See Also CREATE PROCEDURE, CREATE TRIGGER, EVENT WAIT, SET DATABASE

For more information about events, see the *Programmer's Guide*.

EVENT WAIT

Causes an application to wait until notified of an event's occurrence. Available in SQL.

Syntax `EVENT WAIT` *`request_name`*`;`

Argument	Description
request_name	Application event handle declared in a previous EVENT INIT statement.

Description EVENT WAIT is the second step in the InterBase two-part synchronous event mechanism. After a program registers interest in an event, EVENT WAIT causes the process running the application to sleep until the event of interest occurs.

Examples The following embedded SQL statements register an application event name and indicate the program is ready to receive notification when the event occurs:

```
EXEC SQL
   EVENT INIT ORDER_WAIT EMPDB ("new_order");
EXEC SQL
   EVENT WAIT ORDER_WAIT;
```

See Also EVENT INIT

For more information about events, see the *Programmer's Guide*.

EXECUTE

Executes a previously prepared dynamic SQL (DSQL) statement. Available in SQL.

Syntax

```
EXECUTE [TRANSACTION transaction] statement
    [USING SQL DESCRIPTOR xsqlda] [INTO SQL DESCRIPTOR xsqlda];
```

Argument	Description
TRANSACTION *transaction*	Specifies the transaction under which execution occurs.
statement	Alias of a previously prepared statement to execute.
USING SQL DESCRIPTOR	Specifies that values corresponding to the prepared statement's parameters should be taken from the specified XSQLDA.
INTO SQL DESCRIPTOR	Specifies that return values from the executed statement should be stored in the specified XSQLDA.
xsqlda	XSQLDA host-language variable.

Description EXECUTE carries out a previously prepared DSQL statement. It is one of a group of statements that process DSQL statements.

Statement	Purpose
PREPARE	Readies a DSQL statement for execution.
DESCRIBE	Fills in the XSQLDA with information about the statement.
EXECUTE	Executes a previously prepared statement.
EXECUTE IMMEDIATE	Prepares a DSQL statement, executes it once, and discards it.

Before a *statement* can be executed, it must be prepared using the PREPARE statement. The statement can be any SQL data definition, manipulation, or transaction management statement. Once it is prepared, a statement can be executed any number of times.

The TRANSACTION clause can be used in SQL applications running multiple, simultaneous transactions to specify which transaction controls the EXECUTE operation.

USING DESCRIPTOR enables EXECUTE to extract a statement's parameters from an XSQLDA structure previously loaded with values by the application. It need only be used for statements that have dynamic parameters.

INTO DESCRIPTOR enables EXECUTE to store return values from statement execution in a specified XSQLDA structure for application retrieval. It need only be used for DSQL statements that return values.

Note If an EXECUTE statement provides both a USING DESCRIPTOR clause and an INTO DESCRIPTOR clause, then two XSQLDA structures must be provided.

Example The following embedded SQL statement executes a previously prepared DSQL statement:

```
EXEC SQL
   EXECUTE DOUBLE_SMALL_BUDGET;
```

The next embedded SQL statement executes a previously prepared statement with parameters stored in an XSQLDA:

```
EXEC SQL
   EXECUTE Q USING DESCRIPTOR xsqlda;
```

The following embedded SQL statement executes a previously prepared statement with parameters in one XSQLDA, and produces results stored in a second XSQLDA:

```
EXEC SQL
   EXECUTE Q USING DESCRIPTOR xsqlda_1 INTO DESCRIPTOR xsqlda_2;
```

See Also DESCRIBE, EXECUTE IMMEDIATE, PREPARE

For more information about DSQL programming and the XSQLDA, see the *Programmer's Guide*.

EXECUTE IMMEDIATE

Prepares a dynamic SQL (DSQL) statement, executes it once, and discards it. Available in SQL.

Syntax

```
EXECUTE IMMEDIATE [TRANSACTION transaction]
    {:<variable> | "string"} [USING SQL DESCRIPTOR xsqlda];
```

Argument	Description
TRANSACTION *transaction*	Specifies the transaction under which execution occurs.
:*<variable>*	Host variable containing the SQL statement to execute.
"*string*"	A string literal containing the SQL statement to execute.
USING SQL DESCRIPTOR	Specifies that values corresponding to the statement's parameters should be taken from the specified XSQLDA.
xsqlda	XSQLDA host-language variable.

Description EXECUTE IMMEDIATE prepares a DSQL statement stored in a host-language variable or in a literal string, executes it once, and discards it. To prepare and execute a DSQL statement for repeated use, use PREPARE and EXECUTE instead of EXECUTE IMMEDIATE.

The TRANSACTION clause can be used in SQL applications running multiple, simultaneous transactions to specify which transaction controls the EXECUTE IMMEDIATE operation.

The SQL statement to execute must be stored in a host variable or be a string literal. It can contain any SQL data definition statement or data manipulation statement that does not return output.

USING DESCRIPTOR enables EXECUTE IMMEDIATE to extract the values of a a statement's parameters from an XSQLDA structure previously loaded with appropriate values.

Example The following embedded SQL statement prepares and executes a statement in a host variable:

```
EXEC SQL
   EXECUTE IMMEDIATE :insert_date;
```

See Also DESCRIBE, EXECUTE IMMEDIATE, PREPARE

For more information about DSQL programming and the XSQLDA, see the *Programmer's Guide*.

EXECUTE PROCEDURE

Calls a stored procedure. Available in SQL, DSQL, and **isql**.

Syntax SQL form:

```
EXECUTE PROCEDURE [TRANSACTION transaction]
    name [:param [[INDICATOR]:indicator]]
    [, :param [[INDICATOR]:indicator] ...]
    [RETURNING_VALUES :param [[INDICATOR]:indicator]
    [, :param [[INDICATOR]:indicator] ...]];
```

DSQL form:

```
EXECUTE PROCEDURE name [param [, param ...]]
    [RETURNING_VALUES param [, param ...]]
```

isql form:

```
EXECUTE PROCEDURE name [param [, param ...]]
```

Argument	Description
TRANSACTION *transaction*	Specifies the transaction under which execution occurs.
name	Name of an existing stored procedure in the database.
param	Input or output parameter. Can be a host variable or a constant.
RETURNING_VALUES : *param*	Host variable which takes the values of an output parameter.
[INDICATOR] :*indicator*	Host variable for indicating NULL or unknown values.

Description EXECUTE PROCEDURE calls the specified stored procedure. If the procedure requires input parameters, they are passed as host-language variables or as constants. If a procedure returns output parameters to an SQL program, host variables must be supplied in the RETURNING_VALUES clause to hold the values returned.

In **isql**, do not use the RETURN clause or specify output parameters. **isql** will automatically display return values.

Note In DSQL, an EXECUTE PROCEDURE statement requires an input descriptor area if it has input parameters and an output descriptor area if it has output parameters.

In embedded SQL, input parameters and return values may have associated indicator variables for tracking NULL values. Indicator variables are integer values that indicate unknown or NULL values of return values.

An indicator variable that is less than zero indicates that the parameter is unknown or NULL. An indicator variable that is zero or greater indicates that the associated parameter is known and not NULL.

Examples The following embedded SQL statement demonstrates how the executable procedure, DEPT_BUDGET, is called from embedded SQL with literal parameters:

```
EXEC SQL
   EXECUTE PROCEDURE DEPT_BUDGET 100 RETURNING_VALUES :sumb;
```

The next embedded SQL statement calls the same procedure using a host variable instead of a literal as the input parameter:

```
EXEC SQL
   EXECUTE PROCEDURE DEPT_BUDGET :rdno RETURNING_VALUES :sumb;
```

See Also ALTER PROCEDURE, CREATE PROCEDURE, DROP PROCEDURE

For more information about indicator variables, see the *Programmer's Guide*.

FETCH

Retrieves the next available row from the active set of an opened cursor. Available in SQL and DSQL.

Syntax SQL form:

```
FETCH cursor
    [INTO :hostvar [[INDICATOR] :indvar]
    [, :hostvar [[INDICATOR] :indvar] ...]];
```

DSQL form:

```
FETCH cursor {INTO | USING} SQL DESCRIPTOR xsqlda
```

Blob form:

```
See FETCH (BLOB).
```

Argument	Description
cursor	Name of the opened cursor from which to fetch rows.
:*hostvar*	A host-language variable for holding values retrieved with the FETCH. • Optional if FETCH gets rows for DELETE or UPDATE. • Required if row is displayed before DELETE or UPDATE.
:*indvar*	Indicator variable for reporting that a column contains an unknown or NULL value.
[INTO | USING] SQL DESCRIPTOR	Specifies that values should be returned in the specified XSQLDA.
xsqlda	XSQLDA host-language variable.

Description

FETCH retrieves one row at a time into a program from the active set of a cursor. The first FETCH operates on the first row of the active set. Subsequent FETCH statements advance the cursor sequentially through the active set one row at a time until no more rows are found and SQLCODE is set to 100.

A cursor is a one-way pointer into the ordered set of rows retrieved by the select expression in the DECLARE CURSOR statement. A cursor enables sequential access to retrieved rows. There are four related cursor statements:

Stage	Statement	Purpose
1	DECLARE CURSOR	Declare the cursor. The SELECT statement determines rows retrieved for the cursor.
2	OPEN	Retrieve the rows specified for retrieval with DECLARE CURSOR. The resulting rows become the cursor's *active set*.
3	FETCH	Retrieve the current row from the active set, starting with the first row. Subsequent FETCH statements advance the cursor through the set.
4	CLOSE	Close the cursor and release system resources.

The number, size, data type, and order of columns in a FETCH must be the same as those listed in the query expression of its matching DECLARE CURSOR statement. If they are not, the wrong values can be assigned.

Examples

The following embedded SQL statement fetches a column from the active set of a cursor:

```
EXEC SQL
   FETCH PROJ_CNT INTO :department, :hcnt;
```

See Also CLOSE, DECLARE CURSOR, DELETE, FETCH (BLOB), OPEN

For more information about cursors and XSQLDA, see the *Programmer's Guide*.

FETCH (BLOB)

Retrieves the next available segment of a BLOB column and places it in the specified local buffer. Available in SQL.

Syntax

```
FETCH cursor INTO
    [:<buffer> [[INDICATOR] :segment_length];
```

Argument	Description
cursor	Name of an open BLOB cursor from which to retrieve segments.
:*<buffer>*	Host-language variable for holding segments fetched from the BLOB column. User must declare the buffer before fetching segments into it.
INDICATOR	Optional keyword indicating that a host-language variable for indicating the number of bytes returned by the FETCH follows.
:*segment_length*	Host-language variable used to indicate he number of bytes returned by the FETCH.

Description FETCH retrieves the next segment from a BLOB and places it into the specified buffer.

The host variable, *segment_length*, indicates the number of bytes fetched. This is useful when the number of bytes fetched is smaller than the host variable, for example, when fetching the last portion of a BLOB.

FETCH can return two SQLCODE values:

- SQLCODE = 100 indicates that there are no more BLOB segments to retrieve.
- SQLCODE = 101 indicates that a partial segment was retrieved and placed in the local buffer variable.

Note To ensure that a host variable buffer is large enough to hold a BLOB segment buffer during FETCH operations, use the SEGMENT option of the BASED ON statement.

Example The following code, from an embedded SQL application, performs a BLOB FETCH:

```
while (SQLCODE != 100)
{
   EXEC SQL
      OPEN BLOB_CUR USING :blob_id;
   EXEC SQL
      FETCH BLOB_CUR INTO :blob_segment :blob_seg_len;

   while (SQLCODE !=100 || SQLCODE == 101)
   {
      blob_segment{blob_seg_len + 1] = '\0';
      printf("%*.*s",blob_seg_len,blob_seg_len,blob_segment);
         blob_segment{blob_seg_len + 1] = ' ';
      EXEC SQL
         FETCH BLOB_CUR INTO :blob_segment :blob_seg_len;
   }
   . . .
}
```

See Also BASED ON, CLOSE (BLOB), DECLARE CURSOR (BLOB), INSERT CURSOR (BLOB), OPEN (BLOB)

GEN_ID( )

Produces a system-generated integer value. Available in SQL, DSQL, and **isql**.

Syntax `GEN_ID (`*`generator, step`*`)`

Argument	Description
generator	Name of an existing generator.
step	Integer or expression specifying the increment for increasing or decreasing the current generator value. Values can range from $-(2^{31})$ to $2^{31}-1$.

Description The GEN_ID() function:

1. Increments the current value of the specified generator by *step*.
2. Returns the new value of the specified generator.

GEN_ID() is useful for automatically producing unique values that can be inserted into a UNIQUE or PRIMARY KEY column. To insert a generated number in a column, write a trigger, procedure, or SQL statement that calls GEN_ID().

Note A generator is initially created with CREATE GENERATOR. By default, the value of a generator begins at zero. It can be set to a different value with SET GENERATOR.

Examples The following **isql** trigger definition includes a call to GEN_ID():

```
SET TERM !! ;
CREATE TRIGGER CREATE_EMPNO FOR EMPLOYEES
   BEFORE INSERT
   POSITION 0
   AS BEGIN
      NEW.EMPNO = GEN_ID (EMPNO_GEN, 1);
   END
SET TERM ; !!
```

The first time the trigger fires, NEW.EMPNO is set to 1. The next time, it is set to 2, and so on.

See Also CREATE GENERATOR, SET GENERATOR

GRANT

Assigns privileges to users for specified database objects. Available in SQL, DSQL, and **isql**.

Syntax

```
GRANT{
    {ALL [PRIVILEGES] | SELECT | DELETE | INSERT
    | UPDATE [(col [, col ...])]}
    ON [TABLE] {tablename | viewname}
    TO {<object> | <userlist>}
    | EXECUTE ON PROCEDURE procname
    TO {<object> | <userlist>}
    };
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in **isql**, the semicolon is a terminating symbol for the statement, so it must be included.

```
<object> = PROCEDURE procname | TRIGGER trigname | VIEW viewname
    | [USER] username | PUBLIC [, <object>]

<userlist> = [USER] username [, [USER] username ...]
    [WITH GRANT OPTION]
```

Argument	Description
col	Column to which the granted privileges apply.
tablename	Name of an existing table for which granted privileges apply.
viewname	Name of an existing view for which granted privileges apply.
<object>	Name of a user or an existing database object to which privileges are to be granted.
<userlist>	A list of users to whom privileges are to be granted.
WITH GRANT OPTION	Passes GRANT authority for privileges listed in the GRANT statement to *<userlist>*.

Description

GRANT assigns privileges for database objects to users or other database objects. When an object is first created, only its creator has privileges to it, and only its creator can GRANT privileges for it to other users or objects.

To access a table or view, a user or object needs SELECT, INSERT, UPDATE, or DELETE privileges for that table or view. SELECT, INSERT, UPDATE, and DELETE privileges may be assigned as a unit with ALL.

To call a stored procedure in an application, a user or object needs EXECUTE privilege for it.

Users can be given permission to grant privileges to other users by providing a *<userlist>* that includes the WITH GRANT OPTION. Users can only grant to others the privileges that they, themselves, are assigned.

Privileges may be assigned to all users by specifying PUBLIC in place of a list of user names. Specifying PUBLIC grants privileges only to users, not to database objects.

The following table summarizes available privileges:

Table 2-9: SQL Privileges

Privilege	Enables users to . . .
ALL	Perform SELECT, DELETE, INSERT, UPDATE, and EXECUTE.
SELECT	Retrieve rows from a table or view.
DELETE	Eliminate rows from a table or view.
INSERT	Store new rows in a table or view.
UPDATE	Change the current value of one or more columns in a table or view. Can be restricted to a specified subset of columns.
EXECUTE	Execute a stored procedure.

Privileges can only be removed by the user who assigned them by using REVOKE. If ALL privileges are assigned, then ALL privileges must be revoked. If privileges are granted to PUBLIC, they may only be removed for PUBLIC.

Examples

The following **isql** statement grants SELECT and DELETE privileges to a user. The WITH GRANT OPTION gives the user GRANT authority.

```
GRANT SELECT, DELETE ON COUNTRY TO CHLOE WITH GRANT OPTION;
```

The next embedded SQL statement, from an embedded program, grants SELECT and UPDATE privileges to a procedure for a table:

```
EXEC SQL
   GRANT SELECT, UPDATE ON JOB TO PROCEDURE GET_EMP_PROJ;
```

This embedded SQL statement grants EXECUTE privileges for a procedure to another procedure, and to a user:

```
EXEC SQL
   GRANT EXECUTE ON PROCEDURE GET_EMP_PROJ
   TO PROCEDURE ADD_EMP_PROJ, LUIS;
```

See Also

REVOKE

For more information about privileges, see the *Data Definition Guide*.

INSERT

Adds one or more new rows to a specified table. Available in SQL, DSQL, and **isql**.

Syntax

```
INSERT [TRANSACTION transaction] INTO <object> [(col [, col ...])]
    {VALUES (<val> [, <val> ...]) | <select_expr>};
```

Important

In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in **isql**, the semicolon is a terminating symbol for the statement, so it must be included.

```
<object> = tablename | viewname

<val> = {
    :variable | <constant> | <expr>
    | <function> | udf ([<val> [, <val> ...]])
    | NULL | USER | RDB$DB_KEY | ?
    } [COLLATE collation]
```

Note In SQL and **isql**, <*val*> cannot be a parameter placeholder (**?**). In DSQL and **isql**, <*val*> cannot be a variable. The COLLATE clause cannot be used with BLOB values.

```
<constant> = num | "string" | charsetname "string"

<expr> = A valid SQL expression that results in a single column value.
<function> = {
    CAST (<val> AS <datatype>)
    | UPPER (<val>)
    | GEN_ID (generator, <val>)
    }

<select_expr> = A SELECT returning zero or more rows and where the
    number of columns in each row is the same as the number of items
    to be inserted.
```

Argument	Description
TRANSACTION *transaction*	Name of the transaction that controls the execution of the INSERT.
INTO *<object>*	Name of an existing table or view into which to insert data.
col	Name of an existing column in a table or view into which to insert values.
VALUES (*<val>* [, *<val>* ...])	Lists values to insert into the table or view. Values must be listed in the same order as the target columns.
<select_expr>	Query that returns row values to insert into target columns.

Description

INSERT stores one or more new rows of data in an existing table or view. INSERT is one of the database privileges controlled by the GRANT and REVOKE statements.

Values are inserted into a row in column order unless an optional list of target columns is provided. If the target list of columns is a subset of available columns, default or NULL values are automatically stored in all unlisted columns.

If the optional list of target columns is omitted, the VALUES clause must provide values to insert into all columns in the table.

To insert a single row of data, the VALUES clause should include a specific list of values to insert.

To insert multiple rows of data, specify a <*select_expr*> that retrieves existing data from another table to insert into this one. The selected columns must correspond to the columns listed for insert.

Caution It is legal to select from the same table into which insertions are made, but this practice is not advised because it may result in infinite row insertions.

The TRANSACTION clause can be used in multiple transaction SQL applications to specify which transaction controls the INSERT operation. The TRANSACTION clause is not available in DSQL or **isql**.

Examples

The following statement, from an embedded SQL application, adds a row to a table, assigning values from host-language variables to two columns:

```
EXEC SQL
   INSERT INTO EMPLOYEE_PROJECT (EMP_NO, PROJ_ID) VALUES (:emp_no,
   :proj_id);
```

The next **isql** statement specifies values to insert into a table with a SELECT statement:

```
INSERT INTO PROJECTS
   SELECT * FROM NEW_PROJECTS
   WHERE NEW_PROJECTS.START_DATE > "6-JUN-1994";
```

See Also

GRANT, REVOKE, SET TRANSACTION, UPDATE

INSERT CURSOR (BLOB)

Inserts data into a BLOB cursor in units of a BLOB segment-length or less in size. Available in SQL.

Syntax

```
INSERT CURSOR cursor
    VALUES (:buffer [INDICATOR] :bufferlen);
```

Argument	Description
cursor	Name of the BLOB cursor.
VALUES	Clause containing the name and length of the buffer variable to insert.
:*buffer*	Name of host-variable buffer containing information to insert.
INDICATOR	Indicates that the length of data placed in the buffer follows.
:*bufferlen*	Length, in bytes, of the buffer to insert.

Description

INSERT CURSOR writes BLOB data into a column. Data is written in units equal to or less than the segment size for the BLOB. Before inserting data into a BLOB cursor:

- Declare a local variable, *buffer*, to contain the data to be inserted.
- Declare the length of the variable, *bufferlen*.

- Declare a BLOB cursor for INSERT and open it.

Each INSERT into the BLOB column inserts the current contents of *buffer*. Between statements fill *buffer* with new data. Repeat the INSERT until each existing *buffer* is inserted into the BLOB.

Important INSERT CURSOR requires the INSERT privilege, a table privilege controlled by the GRANT and REVOKE statements.

Example The following embedded SQL statement shows an insert into the BLOB cursor:

```
EXEC SQL
   INSERT CURSOR BC VALUES (:line INDICATOR :len);
```

See Also CLOSE (BLOB), DECLARE CURSOR (BLOB), FETCH (BLOB), OPEN (BLOB)

MAX( )

Retrieves the maximum value in a column. Available in SQL, DSQL, and **isql**.

Syntax `MAX ([ALL] <val> | DISTINCT <val>)`

Argument	Description
ALL	Searches all values in a column.
DISTINCT	Eliminates duplicate values before finding the largest.
<val>	A column, constant, host-language variable, expression, non-aggregate function, or UDF.

Description MAX() is an aggregate function that returns the largest value in a specified column, excluding NULL values. If the number of qualifying rows is zero, MAX() returns a NULL value.

When MAX() is used on a CHAR, VARCHAR, or BLOB text column, the largest value returned varies depending on the character set and collation in use for the column. A default character set can be specified for an entire database with the DEFAULT CHARACTER SET clause in CREATE DATABASE, or specified at the column level with the COLLATE clause in CREATE TABLE.

Example The following embedded SQL statement demonstrates the use of SUM(), AVG(), MIN(), and MAX():

```
EXEC SQL
   SELECT SUM (BUDGET), AVG (BUDGET), MIN (BUDGET), MAX (BUDGET)
   FROM DEPARTMENT
```

```
    WHERE HEAD_DEPT = :head_dept
    INTO :tot_budget, :avg_budget, :min_budget, :max_budget;
```

See Also AVG(), COUNT(), CREATE DATABASE, CREATE TABLE, MIN(), SUM()

MIN( )

Retrieves the minimum value in a column. Available in SQL, DSQL, and **isql**.

Syntax `MIN ([ALL] <val> | DISTINCT <val>)`

Argument	Description
ALL	Searches all values in a column.
DISTINCT	Eliminates duplicate values before finding the smallest.
<val>	A column, constant, host-language variable, expression, non-aggregate function, or UDF.

Description MIN() is an aggregate function that returns the smallest value in a specified column, excluding NULL values. If the number of qualifying rows is zero, MIN() returns a NULL value.

When MIN() is used on a CHAR, VARCHAR, or BLOB text column, the smallest value returned varies depending on the character set and collation in use for the column. A default character set can be specified for an entire database with the DEFAULT CHARACTER SET clause in CREATE DATABASE, or specified at the column level with the COLLATE clause in CREATE TABLE.

Example The following embedded SQL statement demonstrates the use of SUM(), AVG(), MIN(), and MAX():

```
EXEC SQL
   SELECT SUM (BUDGET), AVG (BUDGET), MIN (BUDGET), MAX (BUDGET)
   FROM DEPARTMENT
   WHERE HEAD_DEPT = :head_dept
   INTO :tot_budget, :avg_budget, :min_budget, :max_budget;
```

See Also AVG(), COUNT(), CREATE DATABASE, CREATE TABLE, MAX(), SUM()

OPEN

Retrieve specified rows from a cursor declaration. Available in SQL and DSQL.

Syntax

SQL form:

```
OPEN [TRANSACTION transaction] cursor;
```

DSQL form:

```
OPEN [TRANSACTION transaction] cursor [USING SQL DESCRIPTOR xsqlda]
```

Blob form:

```
See OPEN (BLOB).
```

Argument	Description
TRANSACTION *transaction*	Name of the transaction that controls execution of OPEN.
cursor	Name of a previously declared cursor to open.
USING DESCRIPTOR *xsqlda*	Passes the values corresponding to the prepared statement's parameters through the extended descriptor area (XSQLDA).

Description

OPEN evaluates the search condition specified in a cursor's DECLARE CURSOR statement. The selected rows become the *active set* for the cursor.

A cursor is a one-way pointer into the ordered set of rows retrieved by the SELECT in a DECLARE CURSOR statement. It enables sequential access to retrieved rows in turn. There are four related cursor statements:

Stage	Statement	Purpose
1	DECLARE CURSOR	Declare the cursor. The SELECT statement determines rows retrieved for the cursor.
2	OPEN	Retrieve the rows specified for retrieval with DECLARE CURSOR. The resulting rows become the cursor's *active set*.
3	FETCH	Retrieve the current row from the active set, starting with the first row. Subsequent FETCH statements advance the cursor through the set.
4	CLOSE	Close the cursor and release system resources.

Examples

The following embedded SQL statement opens a cursor:

```
EXEC SQL
   OPEN C;
```

See Also

CLOSE, DECLARE CURSOR, FETCH

OPEN (BLOB)

Opens a previously declared BLOB cursor and prepares it for read or insert. Available in SQL.

Syntax

```
OPEN [TRANSACTION name] cursor
    {INTO | USING} :blob_id;
```

Argument	Description
TRANSACTION *name*	Specifies the transaction under which the cursor is opened. Default: The default transaction.
cursor	Name of the BLOB cursor.
INTO | USING	Depending on BLOB cursor type, use one of these: • INTO: For INSERT BLOB. • USING: For READ BLOB.
blob_id	Identifier for the BLOB column.

Description

OPEN prepares a previously declared cursor for reading or inserting BLOB data. Depending on whether the DECLARE CURSOR statement declares a READ or INSERT BLOB cursor, OPEN obtains the value for BLOB ID differently:

- For a READ BLOB, the *blob_id* comes from the outer TABLE cursor.
- For an INSERT BLOB, the *blob_id* is returned by the system.

Examples

The following embedded SQL statements declare and open a BLOB cursor:

```
EXEC SQL
   DECLARE BC CURSOR FOR
   INSERT BLOB PROJ_DESC INTO PRJOECT;
EXEC SQL
   OPEN BC INTO :blob_id;
```

See Also

CLOSE (BLOB), DECLARE CURSOR (BLOB), FETCH (BLOB), INSERT CURSOR (BLOB)

PREPARE

Prepares a dynamic SQL (DSQL) statement for execution. Available in SQL.

Syntax

```
PREPARE [TRANSACTION transaction] statement
    [INTO SQL DESCRIPTOR xsqlda] FROM {:<variable> | "string"};
```

Argument	Description
TRANSACTION *transaction*	Name of the transaction under control of which the statement is executed.
statement	Establishes an alias for the prepared statement that can be used by subsequent DESCRIBE and EXCUTE statements.
INTO *xsqlda*	Specifies an XSQLDA to be filled in with the description of the select-list columns in the prepared statement.
:*<variable>* | "*string*"	DSQL statement to PREPARE. Can be a host-language variable or a string literal.

Description

PREPARE readies a DSQL statement for repeated execution by:

- Checking the statement for syntax errors.
- Determining data types of optionally specified dynamic parameters.
- Optimizing statement execution.
- Compiling the statement for execution by EXECUTE.

PREPARE is part of a group of statements that prepare DSQL statements for execution.

Statement	Purpose
PREPARE	Readies a DSQL statement for execution.
DESCRIBE	Fills in the XSQLDA with information about the statement.
EXECUTE	Executes a previously prepared statement.
EXECUTE IMMEDIATE	Prepares a DSQL statement, executes it once, and discards it.

After a statement is prepared, it is available for execution as many times as necessary during the current session. To prepare and execute a statement only once, use EXECUTE IMMEDIATE.

statement establishes a symbolic name for the actual DSQL statement to prepare. It is *not* declared as a host-language variable. Except for C programs, **gpre** does not distinguish between uppercase and lowercase in *statement*, treating “B” and “b” as the same character. For C programs, use the **gpre -either_case** switch to activate case sensitivity during preprocessing.

If the optional INTO clause is used, PREPARE also fills in the extended SQL descriptor area (XSQLDA) with information about the data type, length, and

name of select-list columns in the prepared statement. This clause is useful only when the statement to prepare is a SELECT.

Note The DESCRIBE statement can be used instead of the INTO clause to fill in the XSQLDA for a select list.

The FROM clause specifies the actual DSQL statement to PREPARE. It can be a host-language variable, or a quoted-string literal. The DSQL statement to PREPARE can be any SQL data definition, data manipulation, or transaction-control statement.

Examples The following embedded SQL statement prepares a DSQL statement from a host-variable statement. Because it uses the optional INTO clause, the assumption is that the DSQL statement in the host variable is a SELECT.

```
EXEC SQL
   PREPARE Q INTO xsqlda FROM :buf;
```

Note The previous statement could also be prepared and described in the following manner:

```
EXEC SQL
   PREPARE Q FROM :buf;
EXEC SQL
   DESCRIBE Q INTO SQL DESCRIPTOR xsqlda;
```

See Also DESCRIBE, EXECUTE, EXECUTE IMMEDIATE

REVOKE

Withdraws privileges from users for specified database objects. Available in SQL, DSQL, and **isql**.

Syntax

```
REVOKE [GRANT OPTION FOR]{
    {ALL [PRIVILEGES] | SELECT | DELETE | INSERT
    | UPDATE [(col [, col ...])]}
    ON [TABLE] {tablename | viewname}
    FROM {<object> | <userlist>}
    | EXECUTE ON PROCEDURE procname
    FROM {<object> | <userlist>}
    };
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in **isql**, the semicolon is a terminating symbol for the statement, so it must be included.

```
<object> = PROCEDURE procname | TRIGGER trigname | VIEW viewname
    | [USER] username | PUBLIC [, <object>]

<userlist> = [USER] username [, [USER] username ...]
```

Argument	Description
GRANT OPTION FOR	Removes grant authority for privileges listed in the REVOKE statement from *<userlist>*. Cannot be used with *<object>*.
col	Column to which the granted privileges apply.
tablename	Name of an existing table for which granted privileges apply.
viewname	Name of an existing view for which granted privileges apply.
<object>	Name of a user or an existing database object from which privileges are to be revoked.
<userlist>	A list of users from whom privileges are to be revoked.

Description

REVOKE removes privileges to access database objects from users or other database objects. Privileges are operations for which a user has authority. The following table defines SQL privileges:

Table 2-10: SQL Privileges

Privilege	Removes a User's Privilege to . . .
ALL	Perform SELECT, DELETE, INSERT, UPDATE, and EXECUTE.
SELECT	Retrieve rows from a table or view.
DELETE	Remove rows from a table or view.
INSERT	Store new rows in a table or view.
UPDATE	Change the current value of one or more columns in a table or view. Can be restricted to a specified subset of columns.
EXECUTE	Execute a stored procedure.

GRANT OPTION FOR revokes a user's right to GRANT privileges to other users.

The following limitations should be noted for REVOKE:

- Only the user who grants a privilege can revoke that privilege.
- A single user may be assigned the same privileges for a database object by any number of other users. A REVOKE issued by a user only removes privileges previously assigned by that particular user.
- Privileges granted to all users with PUBLIC can only be removed by revoking privileges from PUBLIC.

Examples The following **isql** statement takes the SELECT privilege away from a user for a table:

```
REVOKE SELECT ON COUNTRY FROM MIREILLE;
```

The following **isql** statement withdraws EXECUTE privileges for a procedure from another procedure and a user:

```
REVOKE EXECUTE ON PROCEDURE GET_EMP_PROJ
   FROM PROCEDURE ADD_EMP_PROJ, LUIS;
```

See Also GRANT

For more information about privileges, see the *Data Definition Guide*.

ROLLBACK

Restores the database to its state prior to the start of the current transaction. Available in SQL, DSQL, and **isql**.

Syntax `ROLLBACK [TRANSACTION name] [WORK] [RELEASE];`

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in **isql**, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
TRANSACTION *name*	Specifies the transaction to roll back in a multiple-transaction application. Default: roll back the default transaction.
WORK	Optional word allowed for compatibility.
RELEASE	Detaches from all databases after ending the current transaction. SQL only.

Description ROLLBACK undoes changes made to a database by the current transaction, then ends the transaction. It breaks the program's connection to the database and frees system resources. Use RELEASE in the last ROLLBACK to close all open databases. Wait until a program no longer needs the database to release system resources.

The TRANSACTION clause can be used in multiple-transaction SQL applications to specify which transaction to roll back. If omitted, the default transaction is rolled back. The TRANSACTION clause is not available in DSQL.

Note RELEASE, available only in SQL, detaches from all databases after ending the current transaction. In effect, this option ends database processing.

RELEASE is supported for backward compatibility with older versions of InterBase. The preferred method of detaching is with DISCONNECT.

Examples The following **isql** statement rolls back the default transaction:

```
ROLLBACK;
```

The next embedded SQL statement rolls back a named transaction:

```
EXEC SQL
   ROLLBACK TRANSACTION MYTRANS;
```

See Also COMMIT, DISCONNECT

For more information about controlling transactions, see the *Programmer's Guide*.

SELECT

Retrieves data from one or more tables. Available in SQL, DSQL, and **isql**.

Syntax

```
SELECT [TRANSACTION transaction]
    [DISTINCT | ALL] {* | <val> [, <val> ...]}
    [INTO :var [, :var ...]]
    FROM <tableref> [, <tableref> ...]
    [WHERE <search_condition>]
    [GROUP BY col [COLLATE collation] [, col [COLLATE collation] ...]
    [HAVING <search_condition>]
    [UNION <select_expr>]
    [PLAN <plan_expr>]
    [ORDER BY <order_list>]
    [FOR UPDATE [OF col [, col ...]]];
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in **isql**, the semicolon is a terminating symbol for the statement, so it must be included. In **isql**, the INTO clause cannot be specified.

```
<val> = {
    col [<array_dim>] | :variable
    | <constant> | <expr> | <function>
    | udf ([<val> [, <val> ...]])
    | NULL | USER | RDB$DB_KEY | ?
    } [COLLATE collation] [AS alias]
```

Note In SQL and **isql**, <*val*> cannot be a parameter placeholder (**?**). In DSQL and **isql**, <*val*> cannot be a variable. The COLLATE clause cannot be used with BLOB values.

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

Note Outermost brackets, in bold, must be included when declaring arrays.

```
<constant> = num | "string" | charsetname "string"

<expr> = A valid SQL expression that results in a single value.

<function> = {
    COUNT (* | [ALL] <val> | DISTINCT <val>)
    | SUM ([ALL] <val> | DISTINCT <val>)
    | AVG ([ALL] <val> | DISTINCT <val>)
    | MAX ([ALL] <val> | DISTINCT <val>)
    | MIN ([ALL] <val> | DISTINCT <val>)
    | CAST (<val> AS <datatype>)
    | UPPER (<val>)
    | GEN_ID (generator, <val>)
    }

<tableref> = <joined_table> | table | view | procedure
    [(<val> [, <val> ...])] [alias]

<joined_table> = <tableref> <join_type> JOIN <tableref>
    ON <search_condition> | (<joined_table>)

<join-type> = {[INNER] | {LEFT | RIGHT | FULL } [OUTER]} JOIN

<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>}

<operator> = {= | < | > | <= | >= | !< | !> | <> | !=}

<select_one> = SELECT on a single column that returns exactly one row.

<select_list> = SELECT on a single column that returns zero or more
    rows.
```

```
<select_expr> = SELECT on a list of values that returns zero or more
    rows.

<plan_expr> =
    [JOIN | [SORT] MERGE] (<plan_item> | <plan_expr>
    [, <plan_item> | <plan_expr> ...])

<plan_item> = {table | alias}
    NATURAL | INDEX (<index> [, <index> ...]) | ORDER <index>

<order_list> =
    {col | int} [COLLATE collation] [ASC[ENDING] | DESC[ENDING]]
    [, <order_list>]
```

Argument	Description
TRANSACTION *transaction*	Name of the transaction under control of which the statement is executed. SQL only.
SELECT [DISTINCT | ALL]	Specifies data to retrieve. DISTINCT prevents duplicate values from being returned. ALL, the default, retrieves every value.
{* | *<val>* [, *<val>* ...]	The asterisk (*) retrieves all columns for the specified tables. *<val>* [, *<val>* ...] retrieves a specific list of columns and values.
INTO :*var* [, *var* ...]	Singleton select in embedded SQL only. Specifies a list of host-language variables into which to retrieve values.
FROM *<tableref>* [, *<tableref>* ...]	List of tables, views, and stored procedures from which to retrieve data. List can include joins and joins can be nested.
table	Name of an existing table in a database.
view	Name of an existing view in a database.
procedure	Name of an existing stored procedure that functions like a SELECT statement.
alias	Brief, alternate name for a table, view, or column. After declaration in *<tableref>*, *alias* can stand in for subsequent references to a table or view.
<joined_table>	A table reference consisting of a JOIN.
<join_type>	Type of join to perform. Default: INNER.
WHERE *<search_condition>*	Specifies a condition that limits rows retrieved to a subset of all available rows.
GROUP BY *<col>* [, *<col>* ...]	Partitions the results of a query into groups containing all rows with identical values based on a column list.
COLLATE *collation*	Specifies the collation order for the data retrieved by the query.
HAVING *<search_condition>*	Used with GROUP BY. Specifies a condition that limits grouped rows returned.

Argument	Description
UNION	Combines two or more tables that are fully or partially identical in structure.
PLAN *<plan_expr>*	Specifies the access plan for the InterBase optimizer to use during retrieval.
<plan_item>	Specifies a table and index method for a plan.
ORDER BY *<order_list>*	Specifies the order in which rows are returned.

Description

SELECT retrieves data from tables, views, or stored procedures. Variations of the SELECT statement make it possible to:

- Retrieve a single row, or part of a row, from a table. This operation is referred to as a *singleton select*.

  In embedded applications, all SELECT statements that occur outside the context of a cursor must be singleton selects.

- Retrieve multiple rows, or parts of rows, from a table.

  In embedded applications, multiple row retrieval is accomplished by embedding a SELECT within a DECLARE CURSOR statement.

  In **isql**, SELECT can be used directly to retrieve multiple rows.

- Retrieve related rows, or parts of rows, from a join of two or more tables.

- Retrieve all rows, or parts of rows, from union of two or more tables.

All SELECT statements consist of two required clauses (SELECT, FROM), and possibly others (INTO, WHERE, GROUP BY, HAVING, UNION, PLAN, ORDER BY). The following table explains the purpose of each clause, and when they are required:

Table 2-11: SELECT Statement Clauses

Clause	Purpose	Singleton SELECT	Multi-row SELECT
SELECT	Lists columns to retrieve.	Required	Required
INTO	Lists host variables for storing retrieved columns.	Required	Not allowed
FROM	Identifies the tables to search for values.	Required	Required
WHERE	Specifies the search conditions used to restrict retrieved rows to a subset of all available rows. A WHERE clause may contain its own SELECT statement, referred to as a *subquery*.	Optional	Optional

Table 2-11: SELECT Statement Clauses (Continued)

Clause	Purpose	Singleton SELECT	Multi-row SELECT
GROUP BY	Groups related rows based on common column values. Used in conjunction with HAVING.	Optional	Optional
HAVING	Restricts rows generated by GROUP BY to a subset of those rows.	Optional	Optional
UNION	Combines the results of two or more SELECT statements to produce a single, dynamic table without duplicate rows.	Optional	Optional
ORDER BY	Specifies the sort order of rows returned by a SELECT, either ascending (ASC), the default, or descending (DESC).	Optional	Optional
PLAN	Specifies the query plan that should be used by the query optimizer instead of one it would normally choose.	Optional	Optional
FOR UPDATE	Specifies columns listed after the SELECT clause of a DECLARE CURSOR statement that can be updated using a WHERE CURRENT OF clause.	—	Optional

Because SELECT is such a ubiquitous and complex statement, a meaningful discussion lies outside the scope of this reference. To learn how to use SELECT in **isql**, see *Getting Started.* For a complete explanation of SELECT and its clauses, see the *Programmer's Guide.*

Examples

The following **isql** statement selects columns from a table:

```
SELECT JOB_GRADE, JOB_CODE, JOB_COUNTRY, MAX_SALARY FROM PROJECT;
```

The next **isql** statement uses the * wildcard to select all columns and rows from a table:

```
SELECT * FROM COUNTRIES;
```

The following embedded SQL statement uses an aggregate function to count all rows in a table that satisfy a search condition specified in the WHERE clause:

```
EXEC SQL
   SELECT COUNT (*) INTO :cnt FROM COUNTRY
   WHERE POPULATION > 5000000;
```

The next **isql** statement establishes a table alias in the SELECT clause and uses it to identify a column in the WHERE clause:

```
SELECT C.CITY FROM CITIES C
   WHERE C.POPULATION < 1000000;
```

The following **isql** statement selects two columns and orders the rows retrieved by the second of those columns:

```
SELECT CITY, STATE FROM CITIES
   ORDER BY STATE;
```

The next **isql** statement performs a left join:

```
SELECT CITY, STATE_NAME FROM CITIES C
   LEFT JOIN STATES S ON S.STATE = C.STATE
   WHERE C.CITY STARTING WITH "San";
```

The following **isql** statement specifies a query optimization plan for ordered retrieval, utilizing an index for ordering:

```
SELECT * FROM CITIES
   PLAN (CITIES ORDER CITIES_1);
   ORDER BY CITY
```

The next **isql** statement specifies a query optimization plan based on a three-way join with two indexed column equalities:

```
SELECT * FROM CITIES C, STATES S, MAYORS M
   WHERE C.CITY = M.CITY AND C.STATE = M.STATE
   PLAN JOIN (STATE NATURAL, CITIES INDEX DUPE_CITY,
   MAYORS INDEX MAYORS_1);
```

See Also DECLARE CURSOR, DELETE, INSERT, UPDATE

For an introduction to using SELECT in **isql**, see *Getting Started.*

For a full discussion of data retrieval in embedded programming using DECLARE CURSOR and SELECT, see the *Programmer's Guide.*

SET DATABASE

Declares a database handle for database access. Available in SQL.

Syntax

```
SET {DATABASE | SCHEMA} dbhandle =
    [GLOBAL | STATIC | EXTERN]
    [COMPILETIME] [FILENAME] "<dbname>"
    [USER "<name>" PASSWORD "<string>"]
    [RUNTIME [FILENAME] {"<dbname>" | :var}
    [USER {"<name>" | :var} PASSWORD {"<string>" | :var}]];
```

Argument	Description
dbhandle	An alias for a specified database. The alias must be unique within the program. It is used in subsequent SQL statements that support database handles.
GLOBAL	Default. Makes this database declaration available to all modules.
STATIC	Limits scope of this database declaration to the current module.
EXTERN	References a database declaration in another module, rather than actually declaring a new handle.
COMPILETIME	Identifies the database used to look up column references during preprocessing. If only one database is specified in SET DATABASE, it is used both for run time and compile time.
"*<dbname>*"	Location and path name of the database associated with *dbhandle*. For specific platform file specifications, see that platform's operating system manuals.
RUNTIME	Specifies a different database to use at run time than the one specified to use during preprocessing.
:*<var>*	Host-language variable containing a database specification, user name, or password.
USER "*<name>*"	Required for PC client attachments, optional for all others. A valid user name on the server where the database resides. Used with PASSWORD to gain database access on the server.
PASSWORD "*<string>*"	Required for PC client attachments, optional for all others. A valid password on the server where the database resides. Used with USER to gain database access on the server.

Description

SET DATABASE declares a database handle for a specified database and associates the handle with that database. It enables optional specification of different compile-time and run-time databases. Applications that access multiple databases simultaneously must use SET DATABASE statements to establish separate database handles for each database.

dbhandle is an application-defined name for the database handle. Usually handle names are abbreviations of the actual database name. Once declared, database handles can be used in subsequent CONNECT, COMMIT, and ROLLBACK statements. They can also be used within transactions to differentiate table names when two or more attached databases contain tables with the same names.

"<*dbname*>" is a platform-specific file specification for the database to associate with *dbhandle*. It should follow the file syntax conventions for the server where the database resides.

GLOBAL, STATIC, and EXTERN are optional parameters that determine the scope of a database declaration. The default scope, GLOBAL, means that a database handle is available to all code modules in an application. STATIC limits database handle availability to the code module where the handle is declared. EXTERN references a global database handle in another module.

The optional COMPILETIME and RUNTIME parameters enable a single database handle to refer to one database when an application is preprocessed, and to another database when an application is run by a user. If omitted, or if only a COMPILETIME database is specified, InterBase uses the same database during preprocessing and at run time.

The USER and PASSWORD parameters are required for all PC client applications, but are optional for all other remote attachments. The user name and password are verified by the server in the security database before permitting remote attachments to succeed.

Examples The following embedded SQL statement declares a handle for a database:

```
EXEC SQL
   SET DATABASE DB1 = "employee.gdb";
```

The next embedded SQL statement declares different databases at compile time and run time. It uses a host-language variable to specify the run-time database.

```
EXEC SQL
   SET DATABASE EMDBP = "employee.gdb" RUNTIME :db_name;
```

See Also COMMIT, CONNECT, ROLLBACK, SELECT

For more information on the security database, see the *Windows Client User's Guide*.

SET GENERATOR

Sets a new value for an existing generator. Available in SQL, DSQL, and **isql.**

Syntax `SET GENERATOR` *`name`* `TO` *`int`*`;`

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in **isql**, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
name	Name of an existing generator.
int	Value to which to set the generator, an integer from -2^{31} to $2^{31}-1$.

Description SET GENERATOR initializes a starting value for a newly created generator, or resets the value of an existing generator. A generator provides a unique, sequential numeric value through the GEN_ID() function. If a newly created generator is not initialized with SET GENERATOR, its starting value defaults to zero.

int is the new value for the generator. When the GEN_ID() function inserts or updates a value in a column, that value is *int* plus the increment specified in the GEN_ID() step parameter.

Tip To force a generator's first insertion value to 1, use SET GENERATOR to specify a starting value of 0, and set the step value of the GEN_ID() function to 1.

Important When resetting a generator that supplies values to a column defined with PRIMARY KEY or UNIQUE integrity constraints, be careful that the new value does not enable duplication of existing column values, or all subsequent insertions and updates will fail.

Example The following **isql** statement sets a generator value to 1,000:

```
SET GENERATOR CUST_NO_GEN TO 1000;
```

If GEN_ID() now calls this generator with a step value of 1, the first number it returns is 1,001.

See Also CREATE GENERATOR, CREATE PROCEDURE, CREATE TRIGGER, GEN_ID()

SET NAMES

Specifies an active character set to use for subsequent database attachments. Available in SQL, and **isql**.

Syntax `SET NAMES [`*`charset`* `|` `:`*`var`*`];`

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in **isql**, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
charset	Name of a character set that identifies the active character set for a given process. Default: NONE.
:*var*	Host variable containing string identifying a known character set name. Must be declared as a character set name. SQL only.

Description

SET NAMES specifies the character set to use for subsequent database attachments in an application. It enables the server to translate between the default character set for a database on the server and the character set used by an application on the client.

SET NAMES must appear before the SET DATABASE and CONNECT statements it is to affect.

Tip Use a host-language variable with SET NAMES in an embedded application to specify a character set interactively.

For a complete list of character sets recognized by InterBase, see Appendix D: "Character Sets and Collation Orders." Choice of character sets limits possible collation orders to a subset of all available collation orders. Given a specific character set, a specific collation order can be specified when data is selected, inserted, or updated in a column.

Important 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 is performed between the source and destination character sets, so in most cases, errors occur during assignment.

Example

The following statements demonstrate the use of SET NAMES in an embedded SQL application:

```
EXEC SQL
   SET NAMES ISO8859_1;
EXEC SQL
   SET DATABASE DB1 = "employee.gdb";
EXEC SQL
   CONNECT;
```

The next statements demonstrate the use of SET NAMES in **isql**:

```
SET NAMES LATIN1;
CONNECT "employee.gdb";
```

See Also CONNECT, SET DATABASE

For more information about character sets and collation orders, see the *Data Definition Guide*.

SET STATISTICS

Recomputes the selectivity of a specified index. Available in SQL, DSQL, and **isql**.

Syntax `SET STATISTICS INDEX` *name*`;`

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in **isql**, the semicolon is a terminating symbol for the statement, so it must be included.

Argument	Description
name	Name of an existing index for which to recompute selectivity.

Description SET STATISTICS enables the selectivity of an index to be recomputed. Index selectivity is a calculation, based on the number of distinct rows in a table, that is made by the InterBase optimizer when a table is accessed. It is cached in memory, where the optimizer can access it to calculate the optimal retrieval plan for a given query. For tables where the number of duplicate values in indexed columns radically increases or decreases, periodically recomputing index selectivity can improve performance.

Only the creator of an index can use SET STATISTICS.

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

Example The following embedded SQL statement recomputes the selectivity for an index:

```
EXEC SQL
   SET STATISTICS INDEX MINSALX;
```

See Also ALTER INDEX, CREATE INDEX, DROP INDEX

SET TRANSACTION

Starts a transaction and optionally specifies its behavior. Available in SQL, DSQL, and **isql**.

Syntax

```
SET TRANSACTION [NAME transaction]
    [READ WRITE | READ ONLY]
    [WAIT | NO WAIT]
    [[ISOLATION LEVEL] {SNAPSHOT [TABLE STABILITY]
    | READ COMMITTED [[NO] RECORD_VERSION]}]
    [RESERVING <reserving_clause>
    | USING dbhandle [, dbhandle ...]];
```

Important In SQL statements passed to DSQL, omit the terminating semicolon. In embedded applications written in C and C++, and in **isql**, the semicolon is a terminating symbol for the statement, so it must be included.

```
<reserving_clause> = table [, table ...]
    [FOR [SHARED | PROTECTED] {READ | WRITE}] [, <reserving_clause>]
```

Argument	Description
NAME *transaction*	Specifies the name for this transaction. *transaction* is a previously declared and initialized host-language variable. SQL only.
READ WRITE	Specifies that the transaction can read and write to tables (default).
READ ONLY	Specifies that the transaction can only read tables.
WAIT	Specifies that a transaction wait for access if it encounters a lock conflict with another transaction (default).
NO WAIT	Specifies that a transaction immediately return an error if it encounters a lock conflict.
ISOLATION LEVEL	Specifies the isolation level for this transaction when attempting to access the same tables as other simultaneous transactions. Default: SNAPSHOT.
RESERVING *<reserving_clause>*	Reserves lock for tables at transaction start.
USING *dbhandle* [, *dbhandle* ...]	Limits database access to a subset of available databases. SQL only.

Description

SET TRANSACTION starts a transaction, and optionally specifies its database access, lock conflict behavior, and level of interaction with other concurrent transactions accessing the same data. It can also reserve locks for tables. As an alternative to reserving tables, multiple database SQL applications can restrict a transaction's access to a subset of connected databases.

Important Applications preprocessed with the **gpre -manual** switch must explicitly start each transaction with a SET TRANSACTION statement.

SET TRANSACTION affects the default transaction unless another transaction is specified in the optional NAME clause. Named transactions enable support for multiple, simultaneous transactions in a single application. All transaction names must be declared as host-language variables at compile time. In DSQL, this restriction prevents dynamic specification of transaction names.

By default a transaction has READ WRITE access to a database. If a transaction only needs to read data, specify the READ ONLY parameter.

When simultaneous transactions attempt to update the same data in tables, only the first update succeeds. No other transaction can update or delete that data until the controlling transaction is rolled back or committed. By default, transactions WAIT until the controlling transaction ends, then attempt their own operations. To force a transaction to return immediately and report a lock conflict error without waiting, specify the NO WAIT parameter.

ISOLATION LEVEL determines how a transaction interacts with other simultaneous transactions accessing the same tables. The default ISOLATION LEVEL is SNAPSHOT. It provides a repeatable-read view of the database at the moment the transaction starts. Changes made by other simultaneous transactions are not visible.

SNAPSHOT TABLE STABILITY provides a repeatable read of the database by ensuring that transactions cannot write to tables, though they may still be able to read from them.

READ COMMITTED enables a transaction to see the most recently committed changes made by other simultaneous transactions. It can also update rows as long as no update conflict occurs. Uncommitted changes made by other transactions remain invisible until committed. READ COMMITTED also provides two optional parameters:

- NO RECORD_VERSION, the default, reads only the latest version of a row. If the WAIT lock resolution option is specified, then the transaction waits until the latest version of a row is committed or rolled back, and retries its read.
- RECORD_VERSION reads the latest committed version of a row, even if more recent uncommitted version also resides on disk.

The RESERVING clause enables a transaction to register its desired level of access for specified tables when the transaction starts instead of when the transaction attempts its operations on that table. Reserving tables at transaction start can reduce the possibility of deadlocks.

The USING clause, available only in SQL, can be used to conserve system resources by limiting the number of databases a transaction can access.

Examples The following embedded SQL statement sets up the default transaction with an isolation level of READ COMMITTED. If the transaction encounters an update conflict, it waits to get control until the first (locking) transaction is committed or rolled back.

```
EXEC SQL
   SET TRANSACTION WAIT ISOLATION LEVEL READ COMMITTED;
```

The next embedded SQL statement starts a named transaction:

```
EXEC SQL
   SET TRANSACTION NAME T1 READ COMMITTED;
```

The following embedded SQL statement reserves three tables:

```
EXEC SQL
   SET TRANSACTION NAME TR1
   ISOLATION LEVEL READ COMMITTED
   NO RECORD_VERSION WAIT
   RESERVING TABLE1, TABLE2 FOR SHARED WRITE,
      TABLE3 FOR PROTECTED WRITE;
```

See Also COMMIT, ROLLBACK, SET NAMES

For more information about transactions, see the *Programmer's Guide*.

SUM( )

Totals the numeric values in a specified column. Available in SQL, DSQL, and **isql**.

Syntax `SUM ([ALL] <val> | DISTINCT <val>)`

Argument	Description
ALL	Totals all values in a column.
DISTINCT	Eliminates duplicate values before calculating the total.
<val>	A column, constant, host-language variable, expression, non-aggregate function, or UDF that evaluates to a numeric data type.

Description SUM() is an aggregate function that calculates the sum of numeric values for a column. If the number of qualifying rows is zero, SUM() returns a NULL value.

Example The following embedded SQL statement demonstrates the use of SUM(), AVG(), MIN(), and MAX():

```
EXEC SQL
   SELECT SUM (BUDGET), AVG (BUDGET), MIN (BUDGET), MAX (BUDGET)
   FROM DEPARTMENT
   WHERE HEAD_DEPT = :head_dept
   INTO :tot_budget, :avg_budget, :min_budget, :max_budget;
```

See Also AVG(), COUNT(), MAX(), MIN()

UPDATE

Changes the data in all or part of an existing row in a table, view, or active set of a cursor. Available in SQL, DSQL, and **isql**.

Syntax SQL form:

```
UPDATE [TRANSACTION transaction] {table | view}
    SET col = <val> [, col = <val> ...]
    [WHERE <search_condition> | WHERE CURRENT OF cursor];
```

DSQL and **isql** form:

```
UPDATE {table | view}
    SET col = <val> [, col = <val> ...]
    [WHERE <search_condition>

<val> = {
    col [<array_dim>] | :variable
    | <constant> | <expr> | <function>
    | udf ([<val> [, <val> ...]])
    | NULL | USER | ?}
    [COLLATE collation]
```

Note In SQL and **isql**, <*val*> cannot be a parameter placeholder (**?**). In DSQL and **isql**, <*val*> cannot be a variable. The COLLATE clause cannot be used with BLOB values.

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

Note Outermost brackets, in bold, must be included when declaring arrays.

```
<constant> = num | "string" | charsetname "string"

<expr> = A valid SQL expression that results in a single value.

<function> = {
    CAST (<val> AS <datatype>)
    | UPPER (<val>)
```

```
    | GEN_ID (generator, <val>)
    }

<search_condition> = See CREATE TABLE for a full description.
```

Argument	Description
TRANSACTION *transaction*	Name of the transaction under control of which the statement is executed.
table | *view*	Name of an existing table or view to update.
SET *col* = *<val>*	Specifies the columns to change and the values to assign to those columns.
WHERE *<search_condition>*	Searched update only. Specifies the conditions a row must meet to be modified.
WHERE CURRENT OF *cursor*	Positioned update only. Specifies that the current row of a cursor's active set is to be modified. Not available in DSQL and **isql**.

Description

UPDATE modifies one or more existing rows in a table or view. UPDATE is one of the database privileges controlled by GRANT and REVOKE.

For searched updates, the optional WHERE clause can be used to restrict updates to a subset of rows in the table. Searched updates cannot update array slices.

Caution Without a WHERE clause, a searched update modifies all rows in a table.

When performing a positioned update with a cursor, the WHERE CURRENT OF clause must be specified to update one row at a time in the active set.

Note When updating a BLOB column, UPDATE replaces the entire BLOB with a new value.

Examples

The following **isql** statement modifies a column for all rows in a table:

```
UPDATE CITIES
   SET POPULATION = POPULATION * 1.03;
```

The next embedded SQL statement uses a WHERE clause to restrict column modification to a subset of rows:

```
EXEC SQL
   UPDATE PROJECT
   SET PROJ_DESC = :blob_id
   WHERE PROJ_ID = :proj_id;
```

See Also

DELETE, GRANT, INSERT, REVOKE, SELECT

UPPER( )

Converts a string to all uppercase. Available in SQL, DSQL, and **isql**.

Syntax

```
UPPER (<val>)
```

Argument	Description
<val>	A column, constant, host-language variable, expression, function, or UDF that evaluates to a character data type.

Description

UPPER() converts a specified string to all uppercase characters. If applied to character sets that have no case differentiation, UPPER() has no effect.

Examples

The following **isql** statement changes the name, BMatthews, to BMATTHEWS:

```
UPDATE EMPLOYEE
   SET EMP_NAME = UPPER (BMatthews)
   WHERE EMP_NAME = "BMatthews";
```

The next **isql** statement creates a domain called PROJNO with a CHECK constraint that requires the value of the column to be all uppercase:

```
CREATE DOMAIN PROJNO
   AS CHAR(5)
   CHECK (VALUE = UPPER (VALUE));
```

See Also

CAST()

WHENEVER

Traps SQLCODE errors and warnings. Available in SQL.

Syntax

```
WHENEVER {NOT FOUND | SQLERROR | SQLWARNING}
    {GOTO label | CONTINUE};
```

Argument	Description
NOT FOUND	Traps SQLCODE = 100, no qualifiying rows found for the executed statement.
SQLERROR	Traps SQLCODE < 0, failed statement.
SQLWARNING	Traps SQLCODE > 0 AND < 100, system warning or informational message.

Argument	Description
GOTO *label*	Jumps to program location specified by *label* when a warning or error occurs.
CONTINUE	Ignores the warning or error and attempts to continue processing.

Description

WHENEVER traps for SQLCODE errors and warnings. Every executable SQL statement returns an SQLCODE value to indicate its success or failure. If SQLCODE is zero, statement execution is successful. A non-zero value indicates an error, warning, or not found condition.

If the appropriate condition is trapped for, WHENEVER can:

- Use GOTO *label* to jump to an error-handling routine in an application.
- Use CONTINUE to ignore the condition.

WHENEVER can help limit the size of an application, because the application can use a single suite of routines for handling all errors and warnings.

WHENEVER statements should precede any SQL statement that can result in an error. Each condition to trap for requires a separate WHENEVER statement. If WHENEVER is omitted for a particular condition, it is not trapped.

Tip Precede error-handling routines with WHENEVER . . . CONTINUE statements to prevent the possibility of infinite looping in the error-handling routines.

Example

In the following code from an embedded SQL application, three WHENEVER statements determine which label to branch to for error and warning handling:

```
EXEC SQL
   WHENEVER SQLERROR GO TO Error; /* Trap all errors. */

EXEC SQL
   WHENEVER NOT FOUND GO TO AllDone; /* Trap SQLCODE = 100 */

EXEC SQL
   WHENEVER SQLWARNING CONTINUE; /* Ignore all warnings. */
```

See Also

For a complete discussion of error-handling methods and programming, see the *Programmer's Guide*.

WHENEVER

CHAPTER 3

Procedure and Trigger Language Reference

InterBase procedure and trigger language is a complete programming language for writing stored procedures and triggers in **isql** and DSQL. It includes:

- SQL data manipulation statements: INSERT, UPDATE, DELETE, and singleton SELECT.
- Powerful extensions to SQL, including assignment statements, control-flow statements, context variables, event-posting, exceptions, and error handling.

Although stored procedures and triggers are used in entirely different ways and for different purposes, they both use procedure and trigger language. Both triggers and stored procedures can use any statements in procedure and trigger language, with some exceptions:

- OLD and NEW context variables are unique to triggers.
- Input and output parameters, and the SUSPEND and EXIT statements are unique to stored procedures.

The *Data Definition Guide* explains how to create and use stored procedures and triggers. This chapter is a reference for the statements that are unique to trigger and procedure language or that have special syntax when used in triggers and procedures.

Creating Triggers and Stored Procedures

Stored procedures and triggers are defined with the CREATE PROCEDURE and CREATE TRIGGER statements, respectively. Each of these statements is composed of a *header* and a *body*.

The header contains:

- The name of the procedure or trigger, unique within the database.

- For a trigger:
  - A table name, identifying the table that causes the trigger to fire.
  - Statements that determine *when* the trigger fires.
- For a stored procedure:
  - An optional list of *input parameters* and their data types.
  - If the procedure returns values to the calling program, RETURNS followed by a list of *output parameters* and their data types.

The 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 may be many levels of nesting.

Important Because each statement in a stored procedure body must be terminated by a semicolon, you must define a different symbol to terminate the CREATE PROCEDURE statement in **isql**. Use SET TERM before CREATE PROCEDURE to specify a terminator other than a semicolon. After the CREATE PROCEDURE statement, include another SET TERM to change the terminator back to a semicolon.

Nomenclature Conventions

This chapter uses the following nomenclature:

- A *block* is one or more compound statements enclosed by BEGIN and END.
- A *compound statement* is either a block or a statement.
- A *statement* is a single statement in procedure and trigger language.

To illustrate in a syntax diagram:

```
<block> =
BEGIN
   <compound_statement>
   [<compound_statement> ...]
END

<compound_statement> =
   {<block> | statement;}
```

Assignment Statement

Assigns a value to an input or output parameter or local variable. Available in triggers and stored procedures.

Syntax

```
variable = <expression>;
```

Argument	Description
variable	A local variable, input parameter, or output parameter.
<expression>	Any valid combination of variables, SQL operators, and expressions, including user-defined functions (UDFs) and generators.

Description An assignment statement sets the value of a local variable, input parameter, or output parameter. Variables must be declared before they can be used in assignment statements.

Example The first assignment statement below sets the value of *x* to 9. The second statement sets the value of *y* at twice the value of *x*. The third statement uses an arithmetic expression to assign *z* a value of 3.

```
DECLARE VARIABLE x INTEGER;
DECLARE VARIABLE y INTEGER;
DECLARE VARIABLE z INTEGER;

x = 9;
y = 2 * x;
z = 4 * x / (y - 6);
```

See Also DECLARE VARIABLE, Input Parameters, Output Parameters

BEGIN . . . END

Defines a block of statements executed as one. Available in triggers and stored procedures.

Syntax

```
<block> =
    BEGIN
    <compound_statement>
    [<compound_statement> ...]
    END

<compound_statement> =
    {<block> | statement;}
```

Description Each block of statements in the procedure body starts with a BEGIN statement and ends with an END statement. As shown in the above syntax diagram, a block can itself contain other blocks, so there may be many levels of nesting.

BEGIN and END are not followed by a semicolon. In **isql**, the final END in the procedure body is followed by the terminator specified by SET TERM.

The final END statement in a trigger terminates the trigger. The final END statement in a stored procedure operates differently, depending 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 current values of output parameters, if any, to the calling application.

Example The following **isql** fragment of the DELETE_EMPLOYEE procedure shows two examples of BEGIN . . . END blocks.

```
SET TERM !! ;
CREATE PROCEDURE DELETE_EMPLOYEE (EMP_NUM INTEGER)
AS
   DECLARE VARIABLE ANY_SALES INTEGER;
BEGIN
   ANY_SALES = 0;
   . . .
   IF (ANY_SALES > 0) THEN
   BEGIN
      EXCEPTION REASSIGN_SALES;
      EXIT;
   END
   . . .
END !!
```

See Also EXIT, SUSPEND

Comment

Allows programmers to add comments to procedure and trigger code. Available in triggers and stored procedures.

Syntax `/* comment_text */`

Argument	Description
comment_text	Any number of lines of comment text.

Description Comments can be placed on the same line as code, or on separate lines.

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.

Example The following **isql** procedure fragment illustrates 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
   . . .
```

DECLARE VARIABLE

Declares a local variable. Available in triggers and stored procedures.

Syntax `DECLARE VARIABLE` *`var datatype`*`;`

Argument	Description
var	Name of the local variable, unique within the trigger or procedure.
datatype	Data type of the local variable. Can be any InterBase data type except BLOB and arrays.

Description 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. Each local variable requires a separate DECLARE VARIABLE statement, followed by a semicolon (;).

Example The following header declares the local variable, ANY_SALES:

```
CREATE PROCEDURE DELETE_EMPLOYEE (EMP_NUM INTEGER)
AS
   DECLARE VARIABLE ANY_SALES INTEGER;
BEGIN
. . .
```

See Also Input parameters, Output parameters

EXCEPTION

Raises the specified exception. Available in triggers and stored procedures.

Syntax `EXCEPTION` *`name`*`;`

Argument	Description
name	Name of the exception being raised.

Description An exception is a user-defined error that has a name and an associated text message. When raised, an exception:

- Terminates the procedure or trigger in which it was raised and undoes any actions performed (directly or indirectly) by the procedure or trigger.
- Returns an error message to the calling application. In **isql**, the error message is displayed to the screen.

Exceptions can be handled with the WHEN statement. If an exception is handled, it will behave differently.

Example The following **isql** statement defines an exception named REASSIGN_SALES:

```
CREATE EXCEPTION REASSIGN_SALES
      "Reassign the sales records before deleting this employee." !!
```

Then these statements from a procedure body raise the exception:

```
IF (ANY_SALES > 0) THEN
   EXCEPTION REASSIGN_SALES;
```

See Also WHEN . . . DO

For more information on creating exceptions, see “CREATE EXCEPTION” in Chapter 2: “SQL Statement and Function Reference.”

EXECUTE PROCEDURE

Executes a stored procedure. Available in triggers and stored procedures.

Syntax

```
EXECUTE PROCEDURE name [:param [, :param ...]]
    [RETURNING_VALUES :param [, :param ...]];
```

Argument	Description
name	Name of the procedure being executed. Must have been previously defined to the database with CREATE PROCEDURE.
[*param* [, *param* ...]]	List of input parameters, if the procedure requires them. Can be constants or variables. Precede variables with a colon, except NEW and OLD context variables.
[RETURNING_ VALUES *param* [, *param* ...]]	List of output parameters, if the procedure returns values. Precede each with a colon, except NEW and OLD context variables.

Description

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 may be restricted to fewer than 1,000 levels by memory and stack limitations of the server.

Example

The following **isql** example illustrates a recursive procedure, FACTORIAL, which calculates factorials. The procedure calls itself recursively to calculate the factorial of NUM, the input parameter.

```
SET TERM !!;
CREATE PROCEDURE FACTORIAL (NUM INT)
   RETURNS (N_FACTORIAL DOUBLE PRECISION)
AS
DECLARE VARIABLE NUM_LESS_ONE INT;
BEGIN
   IF (NUM = 1) THEN
```

```
    BEGIN /**** Base case: 1 factorial is 1 ****/
       N_FACTORIAL = 1;
       EXIT;
    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;
       EXIT;
    END
END!!
SET TERM ;!!
```

See Also CREATE PROCEDURE, Input Parameters, Output Parameters

For more information on executing procedures, see “EXECUTE PROCEDURE” in Chapter 2: “SQL Statement and Function Reference.”

EXIT

Jumps to the final END statement in the procedure. Available in stored procedures only.

Syntax

```
EXIT;
```

Description 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.

SUSPEND also returns values to the calling program. Each of these statements has specific behavior for executable and select procedures, as shown in the following table.

Table 3-1: SUSPEND, EXIT, and END

Procedure Type	SUSPEND	EXIT	END
Select procedure	Suspends execution of procedure until next FETCH is issued. Returns output values.	Jumps to final END.	Returns control to application. Sets SQLCODE to 100 (end of record stream).
Executable procedure	Jumps to final END. Not Recommended.	Jumps to final END.	Returns values. Returns control to application.

Example

Consider the following procedure from an **isql** script:

```
SET TERM !!;
CREATE PROCEDURE P RETURNS (r INTEGER)
AS
BEGIN
   r = 0;
   WHILE (r < 5) DO
   BEGIN
      r = r + 1;
      SUSPEND;
      IF (r = 3) THEN
         EXIT;
   END
END!!
SET TERM ;!!
```

If this procedure is used as a select procedure in **isql**, for example,

```
SELECT * FROM P;
```

then it will return 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 in **isql**, for example,

```
EXECUTE PROCEDURE P;
```

then it will return 1, since the SUSPEND statement will terminate the procedure and return the current value of *r* to the calling application. SUSPEND should not be used in an executable procedure, so EXIT would be used instead.

See Also

BEGIN . . . END, SUSPEND

FOR SELECT . . . DO

Repeats a block or statement for each row retrieved by the SELECT statement. Available in triggers and stored procedures.

Syntax

```
FOR
    <select_expr>
DO
    <compound_statement>
```

Argument	Description
<select_expr>	SELECT statement that retrieves rows from the database. The INTO clause is required and must come last.
<compound_statement>	Statement or block executed once for each row retrieved by the SELECT statement.

Description

FOR SELECT 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 *<select_expr>* is a normal SELECT, except the INTO clause is required and must be the last clause.

Example

The following **isql** statement 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 RETURNING_VALUES :SUMB;
      TOT = TOT + SUMB;
   END
```

See Also

SELECT

IF . . . THEN . . . ELSE

Conditional statement that performs a block or statement in the IF clause if the specified condition is TRUE, otherwise performs the block or statement in the optional ELSE clause. Available in triggers and stored procedures.

Syntax

```
IF (<condition>) THEN
    <compound_statement>
[ELSE
    <compound_statement>]
```

Argument	Description
<condition>	Boolean expression that evaluates to TRUE, FALSE, or UNKNOWN. Must be enclosed in parentheses.
THEN *<compound_statement>*	Statement or block executed if <*condition*> is TRUE.
[ELSE *<compound_statement>*]	Optional statement or block executed if <*condition*> is not TRUE.

Description The IF . . . THEN . . . ELSE statement selects alternative courses of action by testing a specified condition. <*condition*> 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 executed if <*condition*> is not TRUE.

Example 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;
. . .
```

See Also WHILE . . . DO

Input Parameters

Used to pass values from an application to a stored procedure. Available in stored procedures only.

Syntax

```
CREATE PROCEDURE name
    [(param datatype [, param datatype ...])]
```

Description Input parameters are used to pass values from an application to a stored procedure. They are declared in a comma-delimited list in parentheses following the procedure name in the header of CREATE PROCEDURE. Once declared, they can be used in the procedure body anywhere a variable 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 variable, the change has effect only within the procedure. When control returns to the calling program, the input variable will still have its original value.

Input parameters can be of any InterBase data type except BLOB. Arrays of data types are also unsupported.

Example The following procedure header, from an **isql** script, declares two input parameters, EMP_NO and PROJ_ID:

```
CREATE PROCEDURE ADD_EMP_PROJ (EMP_NO SMALLINT, PROJ_ID CHAR(5))
AS
. . .
```

See Also DECLARE VARIABLE

For more information on declaring input parameters in a procedure header, see "CREATE PROCEDURE" in Chapter 2: "SQL Statement and Function Reference."

NEW Context Variables

Indicates a new column value in an INSERT or UPDATE operation. Available in triggers only.

Syntax

```
NEW.column
```

Argument	Description
column	Name of a column in the affected row.

Description Triggers support two context variables: OLD and NEW. A NEW context variable refers to the new value of a column in an INSERT or UPDATE operation.

Context variables are often used to compare the values of a column before and after it is modified. 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. However, 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.

Example The following **isql** script is a trigger that fires after the EMPLOYEE table is updated, and compares an employee's old and new salary. If there is a change in salary, the trigger inserts an entry in the SALARY_HISTORY table.

```
SET TERM !! ;
CREATE TRIGGER SAVE_SALARY_CHANGE FOR EMPLOYEE
AFTER UPDATE AS
BEGIN
   IF (OLD.SALARY <> NEW.SALARY) THEN
   INSERT INTO SALARY_HISTORY
      (EMP_NO, CHANGE_DATE, UPDATER_ID, OLD_SALARY, PERCENT_CHANGE)
    VALUES (OLD.EMP_NO, "NOW", USER, OLD.SALARY,
      (NEW.SALARY - OLD.SALARY) * 100 / OLD.SALARY);
END !!
SET TERM ; !!
```

See Also OLD Context Variables

For more information on creating triggers, see "CREATE TRIGGER" in Chapter 2: "SQL Statement and Function Reference."

OLD Context Variables

Indicates a current column value in an UPDATE or DELETE operation. Available in triggers only.

Syntax `OLD.`*`column`*

Argument	Description
column	Name of a column in the affected row.

Description Triggers support two context variables: OLD and NEW. An OLD context variable refers to the current or previous value of a column in an INSERT or UPDATE operation.

Context variables are often used to compare the values of a column before and after it is modified. Context variables can be used anywhere a regular variable can be used.

Example The following **isql** script is a trigger that fires after the EMPLOYEE table is updated, and compares an employee's old and new salary. If there is a change in salary, the trigger inserts an entry in the SALARY_HISTORY table.

```
SET TERM !! ;
CREATE TRIGGER SAVE_SALARY_CHANGE FOR EMPLOYEE
AFTER UPDATE AS
BEGIN
   IF (OLD.SALARY <> NEW.SALARY) THEN
   INSERT INTO SALARY_HISTORY
      (EMP_NO, CHANGE_DATE, UPDATER_ID, OLD_SALARY, PERCENT_CHANGE)
    VALUES (OLD.EMP_NO, 'NOW', USER, OLD.SALARY,
      (NEW.SALARY - OLD.SALARY) * 100 / OLD.SALARY);
END !!
SET TERM ; !!
```

See Also NEW Context Variables

For more information about creating triggers, see "CREATE TRIGGER" in Chapter 2: "SQL Statement and Function Reference."

Output Parameters

Used to return values from a stored procedure to the calling application. Available in stored procedures only.

Syntax

```
CREATE PROCEDURE name
    [(param datatype [, param datatype ...])]
    [RETURNS (param datatype [, param datatype ...])]
```

Description Output parameters are used to return values from a procedure to the calling application. They are declared in a comma-delimited list in parentheses following the RETURNS keyword in the header of CREATE PROCEDURE. Once declared, they can be used in the procedure body anywhere a variable can appear. They can be of any InterBase data type except BLOB. Arrays of data types are also unsupported.

If output parameters are declared in a procedure's header, the procedure must assign them values to return to the calling application. Values can be derived from any valid expression in the procedure.

A procedure returns output parameter values to the calling application with a SUSPEND statement. An application receives values of output parameters from a select procedure by using the INTO clause of the SELECT statement. An application receives values of output parameters from an executable procedure by using the RETURNING_VALUES clause.

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.

Example The following **isql** script is a 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)
```

See Also For more information on declaring output parameters in a procedure, see "CREATE PROCEDURE" in Chapter 2: "SQL Statement and Function Reference."

POST_EVENT

Posts an event. Available in triggers and stored procedures.

Syntax `POST_EVENT "`*`event_name`*`" | `*`col`*`;`

Argument	Description
"*event_name*"	Name of the event being posted. Must be enclosed in quotes.

Description POST_EVENT posts an event to the event manager. When an event occurs, this statement will notify the event manager, which alerts applications waiting for the named event.

Example The following statement posts an event named "new_order":

```
POST_EVENT "new_order";
```

The next statement posts an event based on the current value of a column:

```
POST_EVENT NEW.COMPANY;
```

See Also EVENT INIT, EVENT WAIT

For more information on events, see the *Programmer's Guide*.

SELECT

Retrieves a single row that satisfies the requirements of the search condition. The same as standard singleton SELECT, with some differences in syntax. Available in triggers and stored procedures.

Syntax

```
<select_expr> = <select_clause> <from_clause>
    [<where_clause>] [<group_by_clause>]
    [<having_clause>]
    [<union_expression>] [<plan_clause>]
    [<ordering_clause>]
    <into_clause>;
```

Description

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.

The INTO clause comes at the end of the SELECT statement to allow the use of UNION operators. UNION is not allowed in singleton SELECT statements in embedded SQL.

Example

The following statement is a standard singleton SELECT statement in an embedded application:

```
EXEC SQL
   SELECT SUM(BUDGET), AVG(BUDGET)
   INTO :TOT_BUDGET, :AVG_BUDGET
   FROM DEPARTMENT
   WHERE HEAD_DEPT = :HEAD_DEPT
```

To use the above 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;
```

See Also

FOR SELECT . . . DO

For a complete explanation of the standard SELECT syntax, see “SELECT” in Chapter 2: “SQL Statement and Function Reference.”

SUSPEND

Suspends execution of a select procedure until the next FETCH is issued and returns values to the calling application. Available in stored procedures only.

Syntax

```
SUSPEND;
```

Description

The SUSPEND statement:

- Suspends execution of a stored procedure until the application issues the next FETCH.
- Returns values of output parameters, if any.

A procedure should ensure that all output parameters are assigned values before a SUSPEND.

SUSPEND should not be used in an executable procedure. Use EXIT instead to indicate to the reader explicitly that the statement terminates the procedure.

The following table summarizes the behavior of SUSPEND, EXIT, and END.

Table 3-2: SUSPEND, EXIT, and END

Procedure Type	SUSPEND	EXIT	END
Select procedure	Suspends execution of procedure until next FETCH is issued. Returns output values.	Jumps to final END.	Returns control to application. Sets SQLCODE to 100 (end of record stream).
Executable procedure	Jumps to final END. Not Recommended.	Jumps to final END.	Returns values. Returns control to application.

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, which sets SQLCODE to 100.

The SUSPEND statement also delimits atomic statement blocks in select procedures. If an error occurs in a select procedure—either an SQLCODE error, GDSCODE error, or exception—the statements executed since the last SUSPEND and undone. Statements before the last SUSPEND are never undone, unless the transaction comprising the procedure is rolled back.

Example The following procedure. from an **isql** script, illustrates the use of SUSPEND and EXIT:

```
SET TERM !!;
CREATE PROCEDURE P RETURNS (R INTEGER)
AS
BEGIN
   R = 0;
   WHILE (R < 5) DO
   BEGIN
      R = R + 1;
      SUSPEND;
      IF (R = 3) THEN
         EXIT;
   END
END;
SET TERM ;!!
```

If this procedure is used as a select procedure in **isql**, for example,

```
SELECT * FROM P;
```

then it will return values 1, 2, and 3 to the calling application, since the SUSPEND statement returns the current value of *r* to the calling application until *r* = 3, when the procedure performs an EXIT and terminates.

If the procedure is used as an executable procedure in **isql**, for example,

```
EXECUTE PROCEDURE P;
```

then it will return 1, since the SUSPEND statement will terminate the procedure and return the current value of *r* to the calling application. Since SUSPEND should not be used in executable procedures, EXIT would be used instead, indicating that when the statement is encountered, the procedure is exited.

See Also EXIT, BEGIN . . . END

WHEN . . . DO

Error-handling statement that performs the statements following DO when the specified error occurs. Available in triggers and stored procedures.

Syntax

```
WHEN {<error> [, <error> ...] | ANY}
    DO <compound_statement>
```

```
<error>=
    {EXCEPTION exception_name | SQLCODE number | GDSCODE errcode}
```

Argument	Description
EXCEPTION *exception_name*	The name of an exception already in the database.
SQLCODE *number*	An SQLCODE error code number.
GDSCODE *errcode*	An InterBase error code number.
ANY	Keyword that handles any of the above types of errors.
<compound_statement>	Statement or block executed when any of the specified errors occur.

Description

Procedures can handle three kinds of errors with a WHEN 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 error codes.

The WHEN ANY statement handles any of the three types.

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:

- Terminates execution of the BEGIN . . . END block containing the exception and undoes any actions performed in the block.
- Backs out one level to the next BEGIN . . . END block and seeks an exception-handling (WHEN) statement, 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 after WHEN, if found.
- Returns program control to the block or statement in the procedure following the WHEN statement.

Note An exception that is handled with WHEN 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. It can also contain a warning status, such as when there are no more rows to retrieve in a FOR SELECT loop.

Handling InterBase Error Codes

Procedures can also handle InterBase error codes. 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 code, **isc_lock_conflict**. Perhaps if the procedure retries its update, the other transaction may 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.

Example

For example, if a procedure attempts to insert a duplicate value into a column defined as a PRIMARY KEY, InterBase will return SQLCODE -803. This error can be handled in a procedure with the following statement:

```
WHEN SQLCODE -803
DO
   BEGIN
   . . .
```

For example, the following procedure, from an **isql** script, includes a WHEN statement to handle errors that may occur as the procedure runs. If an error occurs and SQLCODE is as expected, the procedure continues with the new value of B. If not, the procedure cannot handle the error, and rolls back all actions of the procedure, returning the active SQLCODE.

```
SET TERM !!;
CREATE PROCEDURE NUMBERPROC (A INTEGER) RETURNS (B INTEGER) AS
BEGIN
   B = 0;
   BEGIN
      UPDATE R SET F1 = F1 + :A;
      UPDATE R SET F2 = F2 * F2;
      UPDATE R SET F1 = F1 + :A;
      WHEN SQLCODE -803 DO
         B = 1;
   END
   EXIT;
END!!
SET TERM; !!
```

See Also EXCEPTION

For more information about InterBase error codes and SQLCODE values, see Appendix B: “Error Codes and Messages.”

WHILE . . . DO

Performs the statement or block following DO as long as the specified condition is TRUE. Available in triggers and stored procedures.

Syntax

```
WHILE (<condition>) DO
    <compound_statement>
```

Argument	Description
<condition>	Boolean expression tested before each execution of the statement or block following DO.
<compound_statement>	Statement or block executed as long as *<condition>* is TRUE.

Description 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.

Example The following procedure, from an **isql** script, uses a WHILE . . . DO loop to compute the sum of all integers from one up to the input parameter:

```
SET TERM !!;
CREATE PROCEDURE SUM_INT (I INTEGER) RETURNS (S INTEGER)
AS
BEGIN
   S = 0;
   WHILE (I > 0) DO
   BEGIN
      S = S + I;
      I = I - 1;
   END
END!!
SET TERM ; !!
```

If this procedure is called from **isql** with the command:

```
EXECUTE PROCEDURE SUM_INT 4;
```

then the results will be:

```
S
==========
10
```

See Also IF . . . THEN, FOR SELECT . . . DO

Appendix A

Keywords

The table in this appendix lists *keywords,* words reserved from use in SQL programs and **isql** (Interactive SQL). The list includes SQL, DSQL, **isql**, and **gpre** keywords.

Keywords are defined for special purposes, and are sometimes called reserved words. A keyword cannot occur in a user-declared identifier or as the name of a table, column, index, trigger, or constraint. Keywords are:

- Part of statements
- Used as statements
- Names of standard data structures or data types

InterBase Keywords

ACTIVE	COLLATE	DO	GRANT
ADD	COLLATION	DOMAIN	GROUP
AFTER	COLUMN	DOUBLE	GROUP_COMMIT_WAIT
ALL	COMMIT	DROP	GROUP_COMMIT_WAIT_TIME
ALTER	COMMITTED	ECHO	HAVING
AND	COMPILETIME	EDIT	HELP
ANY	COMPUTED	ELSE	IF
AS	CLOSE	END	IMMEDIATE
ASC	CONDITIONAL	ENTRY_POINT	IN
ASCENDING	CONNECT	ESCAPE	INACTIVE
AT	CONSTRAINT	EVENT	INDEX
AUTO	CONTAINING	EXCEPTION	INDICATOR
AUTODDL	CONTINUE	EXECUTE	INIT
AVG	COUNT	EXISTS	INNER
BASED	CREATE	EXIT	INPUT
BASENAME	CSTRING	EXTERN	INPUT_TYPE
BASE_NAME	CURRENT	EXTERNAL	INSERT
BEFORE	CURSOR	EXTRACT	INT
BEGIN	DATABASE	FETCH	INTEGER
BETWEEN	DATE	FILE	INTO
BLOB	DB_KEY	FILTER	IS
BLOBEDIT	DEBUG	FLOAT	ISOLATION
BUFFER	DEC	FOR	ISQL
BY	DECIMAL	FOREIGN	JOIN
CACHE	DECLARE	FOUND	KEY
CAST	DEFAULT	FROM	LC_MESSAGES
CHAR	DELETE	FULL	LC_TYPE
CHARACTER	DESC	FUNCTION	LEFT
CHARACTER_LENGTH	DESCENDING	GDSCODE	LENGTH
CHAR_LENGTH	DESCRIBE	GENERATOR	LEV
CHECK	DESCRIPTOR	GEN_ID	LEVEL
CHECK_POINT_LEN	DISCONNECT	GLOBAL	LIKE
CHECK_POINT_LENGTH	DISTINCT	GOTO	LOGFILE

LOG_BUFFER_SIZE
LOG_BUF_SIZE
LONG
MANUAL
MAX
MAXIMUM
MAXIMUM_SEGMENT
MAX_SEGMENT
MERGE
MESSAGE
MIN
MINIMUM
MODULE_NAME
NAMES
NATIONAL
NATURAL
NCHAR
NO
NOAUTO
NOT
NULL
NUMERIC
NUM_LOG_BUFS
NUM_LOG_BUFFERS
OCTET_LENGTH
OF
ON
ONLY
OPEN
OPTION
OR
ORDER
OUTER
OUTPUT
OUTPUT_TYPE
OVERFLOW
PAGE
PAGELENGTH
PAGES
PAGE_SIZE
PARAMETER
PASSWORD
PLAN
POSITION
POST_EVENT
PRECISION
PREPARE
PROCEDURE
PROTECTED
PRIMARY
PRIVILEGES
PUBLIC
QUIT
RAW_PARTITIONS
RDB$DB_KEY
READ
REAL
RECORD_VERSION
REFERENCES
RELEASE
RESERV
RESERVING
RETAIN
RETURN
RETURNING_VALUES
RETURNS
REVOKE
RIGHT
ROLLBACK
RUNTIME
SCHEMA
SEGMENT
SELECT
SET
SHADOW
SHARED
SHELL
SHOW
SINGULAR
SIZE
SMALLINT
SNAPSHOT
SOME
SORT
SQL
SQLCODE
SQLERROR
SQLWARNING
STABILITY
STARTING
STARTS
STATEMENT
STATIC
STATISTICS
SUB_TYPE
SUM
SUSPEND
TABLE
TERMINATOR
THEN
TO
TRANSACTION
TRANSLATE
TRANSLATION
TRIGGER
TRIM
UNCOMMITTED
UNION
UNIQUE
UPDATE
UPPER
USER
USING
VALUE
VALUES
VARCHAR
VARIABLE
VARYING
VERSION
VIEW
WAIT
WHEN
WHENEVER
WHERE
WHILE
WITH
WORK
WRITE

APPENDIX B

Error Codes and Messages

This appendix summarizes InterBase error-handling options and error codes. Tables in this appendix list SQLCODE and InterBase error codes and messages for embedded SQL, dynamic SQL (DSQL), and interactive SQL (**isql**). For a detailed discussion of error handling, see the *Programmer's Guide*.

Error Sources

Run-time errors occur at points of user input or program output. When you run a program or use **isql**, the following types of errors may occur:

Error Type	Description	Action
Database error	Database errors can result from any one of many problems, such as conversion errors, arithmetic exceptions, and validation errors.	If you encounter one of these messages: • Check any messages. • Check the file name or path name and try again.
Bugcheck or internal error	Bugchecks reflect software problems you should report.	If you encounter a bugcheck, execute a traceback and save the output. Submit output and script along with a copy of the database to Borland International, Inc.

Error Reporting and Handling

For reporting and dealing with errors, InterBase utilizes the SQLCODE variable and InterBase codes returned in the status array.

Every executable SQL statement sets the SQLCODE variable, which can serve as a status indicator. During preprocessing, **gpre** declares this variable automatically. An application can test for and use the SQLCODE variable in one of three ways:

- Use the WHENEVER statement to check the value of SQLCODE and direct the program to branch to error-handling routines coded in the application.
- Test for SQLCODE directly.
- Combine WHENEVER and direct SQLCODE testing.

For SQL programs that must be portable between InterBase and other database management systems, limit error-handling routines to one of these methods.

The InterBase status array displays information about errors that supplements SQLCODE messages.

InterBase applications can check both the SQLCODE message and the message returned in the status array.

Trapping Errors With WHENEVER

The WHENEVER statement traps SQL errors and warnings. WHENEVER tests SQLCODE return values and branches to appropriate error-handling routines in the application. Error routines can range from:

- Simple reporting of errors and transaction rollback, or a prompt to the user to reenter a query or data.
- More sophisticated routines that react to many possible error conditions in predictable ways.

WHENEVER helps limit the size of an application, since it can call on a single suite of routines for handling errors and warnings.

Checking SQLCODE Value Directly

Applications can test directly for a particular SQLCODE after each SQL statement. If that SQLCODE occurs, the program can branch to a specific routine.

To handle specific error situations, combine checking for SQLCODE with general WHENEVER statements. These steps outline the procedure, which is described in detail in the *Programmer's Guide*:

1. Override the WHENEVER branching by inserting a WHENEVER SQLERROR CONTINUE statement. The program now ignores SQLCODE.
2. Use an SQLCODE-checking statement to check for a particular SQLCODE and direct the program to an alternative procedure.

3. To return to WHENEVER branching, insert a new WHENEVER statement.

Where portability is not an issue, additional information may be available in the InterBase status array.

InterBase Status Array

Since each SQLCODE value can result from more than one type of error, the InterBase *status array* (**isc_status**) provides additional messages that enable further inquiry into SQLCODE errors.

gpre automatically declares **isc_status**, an array of twenty 32-bit integers, for all InterBase applications during preprocessing. When an error occurs, the status array is loaded with InterBase error codes, message string addresses, and sometimes other numeric, interpretive, platform-specific error data.

This appendix lists all status array codes in "SQL Error Codes and Messages." To see the codes online, display the *ibase.h* file. The location of this file is system-specific.

Access to Status Array Messages

InterBase provides library functions to retrieve and print status array codes and messages.

isc_print_sqlerror( )

When SQLCODE < 0, this function prints the returned SQLCODE value, the corresponding SQL error message, and any additional InterBase error messages in the status array to the screen. Use within an error-handling routine.

Syntax

```
isc_print_sqlerror (short SQLCODE, ISC_STATUS *status_vector);
```

isc_sql_interprete( )

This function retrieves an SQL error message and stores it in a user-supplied buffer for later printing, manipulation, or display. Allow a buffer length of 256 bytes to hold the message. Use when building error display routines or if you are using a windowing system that does not permit direct screen writes. Do not use this function when SQLCODE > 0.

Syntax

```
isc_sql_interprete(short SQLCODE, char *buffer, short length);
```

Action to Take for InterBase Error Codes

After any error occurs, you have the following options: ignore the error, log the error and continue processing, roll back the transaction and try again, or roll back the transaction and quit the application.

For the following errors, it is recommended that you roll back the current transaction and try the operation again:

Table B-1: Status Array Codes that Require Rollback and Retry

Status Array Code	Action to Take
isc_convert_error	Conversion error: A conversion between data types failed; correct the input and retry the operation.
isc_deadlock	Deadlock: Transaction conflicted with another transaction. Wait and try again.
isc_integ_fail	Integrity check: Operation failed due to a trigger. Examine the abort code, fix the error, and try again.
isc_lock_conflict	Lock conflict: Transaction unable to obtain the locks it needed. Wait and try again.
isc_no_dup	Duplicate index entry: Attempt to add a duplicate field. Correct field with duplicate and try again.
isc_not_valid	Validation error: Row did not pass validation test. Correct invalid row and try again.

For More Information

The following table is a guide to further information on planning and programming error-handling routines.

Table B-2: Where to Find Error-handling Topics

Topic	To Find . . .	See . . .
SQLCODE and error handling	Complete discussion and programming instructions	Programmer's Guide
List of SQLCODEs	SQLCODEs and associated messages for embedded SQL, DSQL, **isql**	This Appendix: "SQLCODE Codes and Messages"
WHENEVER syntax	Usage and syntax	Chapter 2: "SQL Statement Definitions"
Programming WHENEVER	Using and programming error-handling routines	Programmer's Guide

Table B-2: Where to Find Error-handling Topics (Continued)

Topic	To Find . . .	See . . .
InterBase status array and functions	Complete programming instructions	Programmer's Guide
List of status array codes	Status array error codes and associated messages for embedded SQL, DSQL, **isql**	This Appendix: "InterBase Status Array Error Codes for SQL"

SQLCODE Error Codes and Messages

This section lists SQLCODE error codes and associated messages in the following tables:

- SQLCODE Error Messages Summary
- SQLCODE Codes and Messages

SQLCODE Error Messages Summary

This table summarizes the types of messages SQLCODE can pass to a program:

Table B-3: SQLCODE and Messages Summary

SQLCODE	Message	Meaning
<0	SQLERROR	Error. The statement did not complete. Table B-4 lists SQLCODE error numbers and messages.
0	SUCCESS	Successful completion.
+1-99	SQLWARNING	System warning or informational message.
+100	NOT FOUND	No qualifying records found. End of file.

SQLCODE Codes and Messages

The following table lists SQLCODEs and associated messages for SQL and DSQL. Some SQLCODE values have more than one text message associated with them. In these cases, InterBase returns the most relevant string message for the error that occurred.

When code messages include the name of a database object or object type, the name is represented by a code in the SQLCODE Text column:

- <*string*>: String value, such as the name of a database object or object type.

- <*long*>: Long integer value, such as the identification number or code of a database object or object type.
- <*digit*>: Integer value, such as the identification number or code of a database object or object type.
- The InterBase number in the right-hand column is the actual error number returned in the error status vector. You can use InterBase error-handling functions to report messages based on these numbers instead of SQL code, but doing so results in non-portable SQL programs.

Table B-4: SQLCODE Codes and Messages

SQLCODE	SQLCODE Text	InterBase Number
101	segment buffer length shorter than expected	335544366L
100	no match for first value expression	335544338L
100	invalid database key	335544354L
100	attempted retrieval of more segments than exist	335544367L
100	attempt to fetch past the last record in a record stream	335544374L
-84	table/procedure has non-SQL security class defined	335544554L
-84	column has non-SQL security class defined	335544555L
-84	procedure *<string>* does not return any values	335544668L
-103	Data type for constant unknown	335544571L
-104	invalid request BLR at offset *<long>*	335544343L
-104	BLR syntax error: expected *<string>* at offset *<long>*, encountered *<long>*	335544390L
-104	context already in use (BLR error)	335544425L
-104	context not defined (BLR error)	335544426L
-104	bad parameter number	335544429L
-104		335544440L
-104	invalid slice description language at offset *<long>*	335544456L
-104	Invalid command	335544570L
-104	Internal error	335544579L
-104	Option specified more than once	335544590L
-104	Unknown transaction option	335544591L
-104	Invalid array reference	335544592L
-104	Token unknown - line *<long>*, char *<long>*	335544634L
-104	Unexpected end of command	335544608L
-104	Token unknown	335544612L

Table B-4: SQLCODE Codes and Messages (Continued)

SQLCODE	SQLCODE Text	InterBase Number
-150	attempted update of read-only table	335544360L
-150	cannot update read-only view *<string>*	335544362L
-150	not updatable	335544446L
-150	Cannot define constraints on views	335544546L
-151	attempted update of read-only column	335544359L
-155	*<string>* is not a valid base table of the specified view	335544658L
-157	must specify column name for view select expression	335544598L
-158	number of columns does not match select list	335544599L
-162	dbkey not available for multi-table views	335544685L
-170	parameter mismatch for procedure *<string>*	335544512L
-170	External functions cannot have more than10 parameters	335544619L
-171	function *<string>* could not be matched	335544439L
-171	column not array or invalid dimensions (expected *<long>*, encountered *<long>*)	335544458L
-171	Return mode by value not allowed for this data type	335544618L
-172	function *<string>* is not defined	335544438L
-204	generator *<string>* is not defined	335544463L
-204	reference to invalid stream number	335544502L
-204	CHARACTER SET *<string>* is not defined	335544509L
-204	procedure *<string>* is not defined	335544511L
-204	status code *<string>* unknown	335544515L
-204	exception *<string>* not defined	335544516L
-204	Name of Referential Constraint not defined in constraints table.	335544532L
-204	could not find table/procedure for GRANT	335544551L
-204	Implementation of text subtype *<digit>* not located.	335544568L
-204	Data type unknown	335544573L
-204	Table unknown	335544580L
-204	Procedure unknown	335544581L
-204	COLLATION *<string>* is not defined	335544588L
-204	COLLATION *<string>* is not valid for specified CHARACTER SET	335544589L
-204	Trigger unknown	335544595L
-204	alias *<string>* conflicts with an alias in the same statement	335544620L

Table B-4: SQLCODE Codes and Messages (Continued)

SQLCODE	SQLCODE Text	InterBase Number
-204	alias *<string>* conflicts with a procedure in the same statement	335544621L
-204	alias *<string>* conflicts with a table in the same statement	335544622L
-204	there is no alias or table named *<string>* at this scope level	335544635L
-204	there is no index *<string>* for table *<string>*	335544636L
-204	Invalid use of CHARACTER SET or COLLATE	335544640L
-204	BLOB SUB_TYPE *<string>* is not defined	335544662L
-205	column *<string>* is not defined in table *<string>*	335544396L
-205	could not find column for GRANT	335544552L
-206	Column unknown	335544578L
-206	Column is not a BLOB	335544587L
-206	Subselect illegal in this context	335544596L
-208	invalid ORDER BY clause	335544617L
-219	table *<string>* is not defined	335544395L
-230	WAL Writer error	335544487L
-231	Log file header of *<string>* too small	335544488L
-232	Invalid version of log file *<string>*	335544489L
-233	Log file *<string>* not latest in the chain but open flag still set	335544490L
-234	Log file *<string>* not closed properly; database recovery may be required	335544491L
-235	Database name in the log file *<string>* is different	335544492L
-236	Unexpected end of log file *<string>* at offset *<long>*	335544493L
-237	Incomplete log record at offset *<long>* in log file *<string>*	335544494L
-238	Log record header too small at offset *<long>* in log file *<string>*	335544495L
-239	Log block too small at offset *<long>* in log file *<string>*	335544496L
-239	Cache length too small	335544691L
-239	Log size too small	335544693L
-239	Log partition size too small	335544694L
-240	Illegal attempt to attach to an uninitialized WAL segment for *<string>*	335544497L
-241	Invalid WAL parameter block option *<string>*	335544498L
-242	Cannot roll over to the next log file *<string>*	335544499L
-243	database does not use Write-ahead Log	335544500L
-244	WAL subsystem encountered error	335544503L
-245	WAL subsystem corrupted	335544504L

Table B-4: SQLCODE Codes and Messages (Continued)

SQLCODE	SQLCODE Text	InterBase Number
-246	Database *<string>*: WAL subsystem bug for pid *<digit>*\	335544513L
-247	Could not expand the WAL segment for database *<string>*	335544514L
-248	Unable to roll over; please see InterBase log.	335544521L
-249	WAL I/O error. Please see InterBase log.	335544522L
-250	WAL writer - Journal server communication error. Please see InterBase log.	335544523L
-251	WAL buffers cannot be increased. Please see InterBase log.	335544524L
-252	WAL setup error. Please see InterBase log.	335544525L
-253	WAL writer synchronization error for the database *<string>*	335544526L
-254	Cannot start WAL writer for the database *<string>*	335544527L
-255	Write-ahead Log without shared cache configuration not allowed	335544556L
-257	WAL defined; Cache Manager must be started first	335544566L
-258	Overflow log specification required for round-robin log	335544567L
-259	Write-ahead Log with shadowing configuration not allowed	335544629L
-260	Cache redefined	335544690L
-260	Log redefined	335544692L
-261	Partitions not supported in series of log file specification	335544695L
-261	Total length of a partitioned log must be specified	335544696L
-281	table *<string>* is not referenced in plan	335544637L
-282	table *<string>* is referenced more than once in plan; use aliases to distinguish	335544638L
-282	the table *<string>* is referenced twice; use aliases to differentiate	335544643L
-282	table *<string>* is referenced twice in view; use an alias to distinguish	335544659L
-282	view *<string>* has more than one base table; use aliases to distinguish	335544660L
-283	table *<string>* is referenced in the plan but not the from list	335544639L
-284	index *<string>* cannot be used in the specified plan	335544642L
-291	Column used in a PRIMARY/UNIQUE constraint must be NOT NULL.	335544531L
-292	Cannot update constraints (RDB$REF_CONSTRAINTS).	335544534L
-293	Cannot update constraints (RDB$CHECK_CONSTRAINTS).	335544535L
-294	Cannot delete CHECK constraint entry (RDB$CHECK_CONSTRAINTS)	335544536L
-295	Cannot update constraints (RDB$RELATION_CONSTRAINTS).	335544545L
-296	internal isc software consistency check (invalid RDB$CONSTRAINT_TYPE)	335544547L

Table B-4: SQLCODE Codes and Messages (Continued)

SQLCODE	SQLCODE Text	InterBase Number
-297	Operation violates CHECK constraint *<string>* on view or table	335544558L
-313	count of column list and variable list do not match	335544669L
-314	Cannot transliterate character between character sets	335544565L
-401	invalid comparison operator for find operation	335544647L
-402	attempted invalid operation on a BLOB	335544368L
-402	BLOB and array data types are not supported for *<string>* operation	335544414L
-402	data operation not supported	335544427L
-406	subscript out of bounds	335544457L
-407	null segment of UNIQUE KEY	335544435L
-413	conversion error from string "*<string>*"	335544334L
-413	filter not found to convert type *<long>* to type *<long>*	335544454L
-501	invalid request handle	335544327L
-501	Attempt to reclose a closed cursor	335544577L
-502	Declared cursor already exists	335544574L
-502	Attempt to reopen an open cursor	335544576L
-504	Cursor unknown	335544572L
-508	no current record for fetch operation	335544348L
-510	Cursor not updatable	335544575L
-518	Request unknown	335544582L
-519	The PREPARE statement identifies a prepare statement with an open cursor	335544688L
-530	violation of FOREIGN KEY constraint: "*<string>*"	335544466L
-530	Cannot prepare a CREATE DATABASE/SCHEMA statement	335544597L
-532	transaction marked invalid by I/O error	335544469L
-551	no permission for *<string>* access to *<string>* *<string>*	335544352L
-552	only the owner of a table may reassign ownership	335544550L
-552	user does not have GRANT privileges for operation	335544553L
-553	cannot modify an existing user privilege	335544529L
-595	the current position is on a crack	335544645L
-596	illegal operation when at beginning of stream	335544644L
-597	Preceding file did not specify length, so *<string>* must include starting page number	335544632L

Table B-4: SQLCODE Codes and Messages (Continued)

SQLCODE	SQLCODE Text	InterBase Number
-598	Shadow number must be a positive integer	335544633L
-599	gen.c: node not supported	335544607L
-600	A node name is not permitted in a secondary, shadow, cache or log file name	335544625L
-600	sort error: corruption in data structure	335544680L
-601	database or file exists	335544646L
-604	Array declared with too many dimensions	335544593L
-604	Illegal array dimension range	335544594L
-605	Inappropriate self-reference of column	335544682L
-607	unsuccessful metadata update	335544351L
-607	cannot modify or erase a system trigger	335544549L
-607	Array/BLOB/DATE data types not allowed in arithmetic	335544657L
-615	lock on table *<string>* conflicts with existing lock	335544475L
-615	requested record lock conflicts with existing lock	335544476L
-615	cannot drop log file when journaling is enabled	335544501L
-615	refresh range number *<long>* already in use	335544507L
-616	Cannot delete PRIMARY KEY being used in FOREIGN KEY definition.	335544530L
-616	Cannot delete index used by an Integrity Constraint	335544539L
-616	Cannot modify index used by an Integrity Constraint	335544540L
-616	Cannot delete trigger used by a CHECK Constraint	335544541L
-616	Cannot delete column being used in an Integrity Constraint.	335544543L
-616	there are *<long>* dependencies	335544630L
-616	last column in a table cannot be deleted	335544674L
-617	Cannot update trigger used by a CHECK Constraint	335544542L
-617	Cannot rename column being used in an Integrity Constraint.	335544544L
-618	Cannot delete index segment used by an Integrity Constraint	335544537L
-618	Cannot update index segment used by an Integrity Constraint	335544538L
-625	validation error for column *<string>*, value "*<string>*"	335544347L
-637	duplicate specification of *<string>* - not supported	335544664L
-660	Non-existent PRIMARY or UNIQUE KEY specified for FOREIGN KEY.	335544533L
-660	cannot create index *<string>*	335544628L
-663	segment count of 0 defined for index *<string>*	335544624L

Table B-4: SQLCODE Codes and Messages (Continued)

SQLCODE	SQLCODE Text	InterBase Number
-663	too many keys defined for index *<string>*	335544631L
-663	too few key columns found for index *<string>* (incorrect column name?)	335544672L
-664	key size exceeds implementation restriction for index "*<string>*"	335544434L
-677	*<string>* extension error	335544445L
-685	invalid BLOB type for operation	335544465L
-685	attempt to index BLOB column in index *<string>*	335544670L
-685	attempt to index array column in index *<string>*	335544671L
-689	page *<long>* is of wrong type (expected *<long>*, found *<long>*)	335544403L
-689	wrong page type	335544650L
-690	segments not allowed in expression index *<string>*	335544679L
-691	new record size of *<long>* bytes is too big	335544681L
-692	maximum indexes per table (*<digit>*) exceeded	335544477L
-693	Too many concurrent executions of the same request	335544663L
-694	cannot access column *<string>* in view *<string>*	335544684L
-802	arithmetic exception, numeric overflow, or string truncation	335544321L
-803	attempt to store duplicate value (visible to active transactions) in unique index "*<string>*"	335544349L
-803	violation of PRIMARY or UNIQUE KEY constraint: "*<string>*"	335544665L
-804	wrong number of arguments on call	335544380L
-804	SQLDA missing or incorrect vresion, or incorrect number/type of variables	335544583L
-804	Count of columns not equal count of values	335544584L
-804	Function unknown	335544586L
-806	Only simple column names permitted for VIEW WITH CHECK OPTION	335544600L
-807	No where clause for VIEW WITH CHECK OPTION	335544601L
-808	Only one table allowed for VIEW WITH CHECK OPTION	335544602L
-809	DISTINCT, GROUP or HAVING not permitted for VIEW WITH CHECK OPTION	335544603L
-810	No subqueries permitted for VIEW WITH CHECK OPTION	335544605L
-811	multiple rows in singleton select	335544652L
-816	external file could not be opened for output	335544651L
-817	attempted update during read-only transaction	335544361L
-817	attempted write to read-only BLOB	335544371L

Table B-4: SQLCODE Codes and Messages (Continued)

SQLCODE	SQLCODE Text	InterBase Number
-817	operation not supported	335544444L
-820	metadata is obsolete	335544356L
-820	unsupported on-disk structure for file *<string>*; found *<long>*, support *<long>*	335544379L
-820	wrong DYN version	335544437L
-820	minor version too high found *<long>* expected *<long>*	335544467L
-823	invalid bookmark handle	335544473L
-824	invalid lock level *<digit>*	335544474L
-825	invalid lock handle	335544519L
-826	Invalid statement handle	335544585L
-827	invalid direction for find operation	335544655L
-828	invalid key position	335544678L
-829	invalid column reference	335544616L
-830	column used with aggregate	335544615L
-831	Attempt to define a second PRIMARY KEY for the same table	335544548L
-832	FOREIGN KEY column count does not match PRIMARY KEY	335544604L
-833	expression evaluation not supported	335544606L
-834	refresh range number *<long>* not found	335544508L
-835	bad checksum	335544649L
-836	exception *<digit>*	335544517L
-837	restart shared cache manager	335544518L
-838	database *<string>* shutdown in *<digit>* seconds	335544560L
-839	journal file wrong format	335544686L
-840	intermediate journal file full	335544687L
-841	too many versions	335544677L
-842	Precision should be greater than 0	335544697L
-842	Scale cannot be greater than precision	335544698L
-842	Short integer expected	335544699L
-842	Long integer expected	335544700L
-842	Unsigned short integer expected	335544701L
-901	invalid database key	335544322L
-901	unrecognized database parameter block	335544326L

Table B-4: SQLCODE Codes and Messages (Continued)

SQLCODE	SQLCODE Text	InterBase Number
-901	invalid BLOB handle	335544328L
-901	invalid BLOB ID	335544329L
-901	invalid parameter in transaction parameter block	335544330L
-901	invalid format for transaction parameter block	335544331L
-901	invalid transaction handle (expecting explicit transaction start)	335544332L
-901	attempt to start more than *<long>* transactions	335544337L
-901	information type inappropriate for object specified	335544339L
-901	no information of this type available for object specified	335544340L
-901	unknown information item	335544341L
-901	action cancelled by trigger (*<long>*) to preserve data integrity	335544342L
-901	lock conflict on no wait transaction	335544345L
-901	program attempted to exit without finishing database	335544350L
-901	transaction is not in limbo	335544353L
-901	BLOB was not closed	335544355L
-901	cannot disconnect database with open transactions (*<long>* active)	335544357L
-901	message length error (encountered *<long>*, expected *<long>*)	335544358L
-901	no transaction for request	335544363L
-901	request synchronization error	335544364L
-901	request referenced an unavailable database	335544365L
-901	attempted read of a new, open BLOB	335544369L
-901	attempted action on blob outside transaction	335544370L
-901	attempted reference to BLOB in unavailable database	335544372L
-901	table *<string>* was omitted from the transaction reserving list	335544376L
-901	request includes a DSRI extension not supported in this implementation	335544377L
-901	feature is not supported	335544378L
-901	*<string>*	335544382L
-901	unrecoverable conflict with limbo transaction *<long>*	335544383L
-901	internal error	335544392L
-901	database handle not zero	335544407L
-901	transaction handle not zero	335544408L
-901	transaction in limbo	335544418L
-901	transaction not in limbo	335544419L

Table B-4: SQLCODE Codes and Messages (Continued)

SQLCODE	SQLCODE Text	InterBase Number
-901	transaction outstanding	335544420L
-901	undefined message number	335544428L
-901	blocking signal has been received	335544431L
-901	database system cannot read argument *<long>*	335544442L
-901	database system cannot write argument *<long>*	335544443L
-901	*<string>*	335544450L
-901	transaction *<long>* is *<string>*	335544468L
-901	invalid statement handle	335544485L
-901	lock time-out on wait transaction	335544510L
-901	invalid service handle	335544559L
-901	wrong version of service parameter block	335544561L
-901	unrecognized service parameter block	335544562L
-901	service *<string>* is not defined	335544563L
-901	INDEX *<string>*	335544609L
-901	EXCEPTION *<string>*	335544610L
-901	COLUMN *<string>*	335544611L
-901	union not supported	335544613L
-901	Unsupported DSQL construct	335544614L
-901	Illegal use of keyword VALUE	335544623L
-901	TABLE *<string>*	335544626L
-901	PROCEDURE *<string>*	335544627L
-901	Specified domain or source column does not exist	335544641L
-901	variable *<string>* conflicts with parameter in same procedure	335544656L
-901	server version too old to support all CREATE DATABASE options	335544666L
-901	cannot delete	335544673L
-901	sort error	335544675L
-902	internal isc software consistency check (*<string>*)	335544333L
-902	database file appears corrupt (*<string>*)	335544335L
-902	I/O error during "*<string>*" operation for file "*<string>*"	335544344L
-902	corrupt system table	335544346L
-902	operating system directive *<string>* failed	335544373L
-902	internal error	335544384L

Table B-4: SQLCODE Codes and Messages (Continued)

SQLCODE	SQLCODE Text	InterBase Number
-902	internal error	335544385L
-902	internal error	335544387L
-902	block size exceeds implementation restriction	335544388L
-902	incompatible version of on-disk structure	335544394L
-902	internal error	335544397L
-902	internal error	335544398L
-902	internal error	335544399L
-902	internal error	335544400L
-902	internal error	335544401L
-902	internal error	335544402L
-902	database corrupted	335544404L
-902	checksum error on database page *<long>*	335544405L
-902	index is broken	335544406L
-902	transaction--request mismatch (synchronization error)	335544409L
-902	bad handle count	335544410L
-902	wrong version of transaction parameter block	335544411L
-902	unsupported BLR version (expected *<long>*, encountered *<long>*)	335544412L
-902	wrong version of database parameter block	335544413L
-902	database corrupted	335544415L
-902	internal error	335544416L
-902	internal error	335544417L
-902	internal error	335544422L
-902	internal error	335544423L
-902	lock manager error	335544432L
-902	SQL error code = *<long>*	335544436L
-902		335544448L
-902		335544449L
-902	cache buffer for page *<long>* invalid	335544470L
-902	there is no index in table *<string>* with id *<digit>*	335544471L
-902	Your user name and password are not defined. Ask your database administrator to set up an InterBase login.	335544472L
-902	enable journal for database before starting online dump	335544478L

Table B-4: SQLCODE Codes and Messages (Continued)

SQLCODE	SQLCODE Text	InterBase Number
-902	online dump failure. Retry dump	335544479L
-902	an online dump is already in progress	335544480L
-902	no more disk/tape space. Cannot continue online dump	335544481L
-902	journaling allowed only if database has Write-ahead Log	335544482L
-902	maximum number of online dump files that can be specified is 16	335544483L
-902	error in opening Write-ahead Log file during recovery	335544484L
-902	Write-ahead log subsystem failure	335544486L
-902	must specify archive file when enabling long term journal for databases with round-robin log files	335544505L
-902	database *<string>* shutdown in progress	335544506L
-902	long-term journaling already enabled	335544520L
-902	database *<string>* shutdown	335544528L
-902	database shutdown unsuccessful	335544557L
-902	cannot attach to password database	335544653L
-902	cannot start transaction for password database	335544654L
-902	long-term journaling not enabled	335544564L
-902	Dynamic SQL Error	335544569L
-904	invalid database handle (no active connection)	335544324L
-904	unavailable database	335544375L
-904	Implementation limit exceeded	335544381L
-904	too many requests	335544386L
-904	buffer exhausted	335544389L
-904	buffer in use	335544391L
-904	request in use	335544393L
-904	no lock manager available	335544424L
-904	unable to allocate memory from operating system	335544430L
-904	update conflicts with concurrent update	335544451L
-904	object *<string>* is in use	335544453L
-904	cannot attach active shadow file	335544455L
-904	a file in manual shadow *<long>* is unavailable	335544460L
-904	cannot add index, index root page is full.	335544661L
-904	sort error: not enough memory	335544676L

Table B-4: SQLCODE Codes and Messages (Continued)

SQLCODE	SQLCODE Text	InterBase Number
-904	request depth exceeded. (Recursive definition?)	335544683L
-906	product *<string>* is not licensed	335544452L
-909	drop database completed with errors	335544667L
-911	record from transaction *<long>* is stuck in limbo	335544459L
-913	deadlock	335544336L
-922	file *<string>* is not a valid database	335544323L
-923	connection rejected by remote interface	335544421L
-923	secondary server attachments cannot validate databases	335544461L
-923	secondary server attachments cannot start journaling	335544462L
-923	secondary server attachments cannot start logging	335544464L
-924	bad parameters on attach or create database	335544325L
-924	communication error with journal "*<string>*"	335544433L
-924	database detach completed with errors	335544441L
-924	Connection lost to pipe server	335544648L
-926	no rollback performed	335544447L
-999	InterBase error	335544689L

InterBase Status Array Error Codes

This section lists InterBase error codes and associated messages returned in the status array in the following tables. When code messages include the name of a database object or object type, the name is represented by a code in the Message column:

- <*string*>: String value, such as the name of a database object or object type.
- <*digit*>: Integer value, such as the identification number or code of a database object or object type.

- <*long*>: Long integer value, such as the identification number or code of a database object or object type.

The following table lists SQL Status Array codes for embedded SQL programs, DSQL, and **isql**.

Table B-5: InterBase Status Array Error Codes

Error Code	Number	Message
isc_arith_except	335544321L	arithmetic exception, numeric overflow, or string truncation
isc_bad_dbkey	335544322L	invalid database key
isc_bad_db_format	335544323L	file *<string>* is not a valid database
isc_bad_db_handle	335544324L	invalid database handle (no active connection)
isc_bad_dpb_content	335544325L	bad parameters on attach or create database
isc_bad_dpb_form	335544326L	unrecognized database parameter block
isc_bad_req_handle	335544327L	invalid request handle
isc_bad_segstr_handle	335544328L	invalid BLOB handle
isc_bad_segstr_id	335544329L	invalid BLOB ID
isc_bad_tpb_content	335544330L	invalid parameter in transaction parameter block
isc_bad_tpb_form	335544331L	invalid format for transaction parameter block
isc_bad_trans_handle	335544332L	invalid transaction handle (expecting explicit transaction start)
isc_bug_check	335544333L	internal isc software consistency check (*<string>*)
isc_convert_error	335544334L	conversion error from string "*<string>*"
isc_db_corrupt	335544335L	database file appears corrupt (*<string>*)
isc_deadlock	335544336L	deadlock
isc_excess_trans	335544337L	attempt to start more than *<long>* transactions
isc_from_no_match	335544338L	no match for first value expression
isc_infinap	335544339L	information type inappropriate for object specified
isc_infona	335544340L	no information of this type available for object specified
isc_infunk	335544341L	unknown information item
isc_integ_fail	335544342L	action cancelled by trigger (*<long>*) to preserve data integrity
isc_invalid_blr	335544343L	invalid request BLR at offset *<long>*
isc_io_error	335544344L	I/O error during "*<string>*" operation for file "*<string>*"
isc_lock_conflict	335544345L	lock conflict on no wait transaction
isc_metadata_corrupt	335544346L	corrupt system table

Table B-5: InterBase Status Array Error Codes (Continued)

Error Code	Number	Message
isc_not_valid	335544347L	validation error for column *<string>*, value "*<string>*"
isc_no_cur_rec	335544348L	no current record for fetch operation
isc_no_dup	335544349L	attempt to store duplicate value (visible to active transactions) in unique index "*<string>*"
isc_no_finish	335544350L	program attempted to exit without finishing database
isc_no_meta_update	335544351L	unsuccessful metadata update
isc_no_priv	335544352L	no permission for *<string>* access to *<string>* *<string>*
isc_no_recon	335544353L	transaction is not in limbo
isc_no_record	335544354L	invalid database key
isc_no_segstr_close	335544355L	BLOB was not closed
isc_obsolete_metadata	335544356L	metadata is obsolete
isc_open_trans	335544357L	cannot disconnect database with open transactions (*<long>* active)
isc_port_len	335544358L	message length error (encountered *<long>*, expected *<long>*)
isc_read_only_field	335544359L	attempted update of read-only column
isc_read_only_rel	335544360L	attempted update of read-only table
isc_read_only_trans	335544361L	attempted update during read-only transaction
isc_read_only_view	335544362L	cannot update read-only view *<string>*
isc_req_no_trans	335544363L	no transaction for request
isc_req_sync	335544364L	request synchronization error
isc_req_wrong_db	335544365L	request referenced an unavailable database
isc_segment	335544366L	segment buffer length shorter than expected
isc_segstr_eof	335544367L	attempted retrieval of more segments than exist
isc_segstr_no_op	335544368L	attempted invalid operation on a BLOB
isc_segstr_no_read	335544369L	attempted read of a new, open BLOB
isc_segstr_no_trans	335544370L	attempted action on blob outside transaction
isc_segstr_no_write	335544371L	attempted write to read-only BLOB
isc_segstr_wrong_db	335544372L	attempted reference to BLOB in unavailable database
isc_sys_request	335544373L	operating system directive *<string>* failed
isc_stream_eof	335544374L	attempt to fetch past the last record in a record stream
isc_unavailable	335544375L	unavailable database
isc_unres_rel	335544376L	table *<string>* was omitted from the transaction reserving list

Table B-5: InterBase Status Array Error Codes (Continued)

Error Code	Number	Message
isc_uns_ext	335544377L	request includes a DSRI extension not supported in this implementation
isc_wish_list	335544378L	feature is not supported
isc_wrong_ods	335544379L	unsupported on-disk structure for file *<string>*; found *<long>*, support *<long>*
isc_wronumarg	335544380L	wrong number of arguments on call
isc_imp_exc	335544381L	Implementation limit exceeded
isc_random	335544382L	*<string>*
isc_fatal_conflict	335544383L	unrecoverable conflict with limbo transaction *<long>*
isc_badblk	335544384L	internal error
isc_invpoolcl	335544385L	internal error
isc_nopoolids	335544386L	too many requests
isc_relbadblk	335544387L	internal error
isc_blktoobig	335544388L	block size exceeds implementation restriction
isc_bufexh	335544389L	buffer exhausted
isc_syntaxerr	335544390L	BLR syntax error: expected *<string>* at offset *<long>*, encountered *<long>*
isc_bufinuse	335544391L	buffer in use
isc_bdbincon	335544392L	internal error
isc_reqinuse	335544393L	request in use
isc_badodsver	335544394L	incompatible version of on-disk structure
isc_relnotdef	335544395L	table *<string>* is not defined
isc_fldnotdef	335544396L	column *<string>* is not defined in table *<string>*
isc_dirtypage	335544397L	internal error
isc_waifortra	335544398L	internal error
isc_doubleloc	335544399L	internal error
isc_nodnotfnd	335544400L	internal error
isc_dupnodfnd	335544401L	internal error
isc_locnotmar	335544402L	internal error
isc_badpagtyp	335544403L	page *<long>* is of wrong type (expected *<long>*, found *<long>*)
isc_corrupt	335544404L	database corrupted
isc_badpage	335544405L	checksum error on database page *<long>*
isc_badindex	335544406L	index is broken

Table B-5: InterBase Status Array Error Codes (Continued)

Error Code	Number	Message
isc_dbbnotzer	335544407L	database handle not zero
isc_tranotzer	335544408L	transaction handle not zero
isc_trareqmis	335544409L	transaction--request mismatch (synchronization error)
isc_badhndcnt	335544410L	bad handle count
isc_wrotpbver	335544411L	wrong version of transaction parameter block
isc_wroblrver	335544412L	unsupported BLR version (expected *<long>*, encountered *<long>*)
isc_wrodpbver	335544413L	wrong version of database parameter block
isc_blobnotsup	335544414L	BLOB and array data types are not supported for *<string>* operation
isc_badrelation	335544415L	database corrupted
isc_nodetach	335544416L	internal error
isc_notremote	335544417L	internal error
isc_trainlim	335544418L	transaction in limbo
isc_notinlim	335544419L	transaction not in limbo
isc_traoutsta	335544420L	transaction outstanding
isc_connect_reject	335544421L	connection rejected by remote interface
isc_dbfile	335544422L	internal error
isc_orphan	335544423L	internal error
isc_no_lock_mgr	335544424L	no lock manager available
isc_ctxinuse	335544425L	context already in use (BLR error)
isc_ctxnotdef	335544426L	context not defined (BLR error)
isc_datnotsup	335544427L	data operation not supported
isc_badmsgnum	335544428L	undefined message number
isc_badparnum	335544429L	bad parameter number
isc_virmemexh	335544430L	unable to allocate memory from operating system
isc_blocking_signal	335544431L	blocking signal has been received
isc_lockmanerr	335544432L	lock manager error
isc_journerr	335544433L	communication error with journal "*<string>*"
isc_keytoobig	335544434L	key size exceeds implementation restriction for index "*<string>*"
isc_nullsegkey	335544435L	null segment of UNIQUE KEY
isc_sqlerr	335544436L	SQL error code = *<long>*

Table B-5: InterBase Status Array Error Codes (Continued)

Error Code	Number	Message
isc_wrodynver	335544437L	wrong DYN version
isc_funnotdef	335544438L	function *<string>* is not defined
isc_funmismat	335544439L	function *<string>* could not be matched
isc_bad_msg_vec	335544440L	
isc_bad_detach	335544441L	database detach completed with errors
isc_noargacc_read	335544442L	database system cannot read argument *<long>*
isc_noargacc_write	335544443L	database system cannot write argument *<long>*
isc_read_only	335544444L	operation not supported
isc_ext_err	335544445L	*<string>* extension error
isc_non_updatable	335544446L	not updatable
isc_no_rollback	335544447L	no rollback performed
isc_bad_sec_info	335544448L	
isc_invalid_sec_info	335544449L	
isc_misc_interpreted	335544450L	*<string>*
isc_update_conflict	335544451L	update conflicts with concurrent update
isc_unlicensed	335544452L	product *<string>* is not licensed
isc_obj_in_use	335544453L	object *<string>* is in use
isc_nofilter	335544454L	filter not found to convert type *<long>* to type *<long>*
isc_shadow_accessed	335544455L	cannot attach active shadow file
isc_invalid_sdl	335544456L	invalid slice description language at offset *<long>*
isc_out_of_bounds	335544457L	subscript out of bounds
isc_invalid_dimension	335544458L	column not array or invalid dimensions (expected *<long>*, encountered *<long>*)
isc_rec_in_limbo	335544459L	record from transaction *<long>* is stuck in limbo
isc_shadow_missing	335544460L	a file in manual shadow *<long>* is unavailable
isc_cant_validate	335544461L	secondary server attachments cannot validate databases
isc_cant_start_journal	335544462L	secondary server attachments cannot start journaling
isc_gennotdef	335544463L	generator *<string>* is not defined
isc_cant_start_logging	335544464L	secondary server attachments cannot start logging
isc_bad_segstr_type	335544465L	invalid BLOB type for operation
isc_foreign_key	335544466L	violation of FOREIGN KEY constraint: "*<string>*"
isc_high_minor	335544467L	minor version too high found *<long>* expected *<long>*

Table B-5: InterBase Status Array Error Codes (Continued)

Error Code	Number	Message
isc_tra_state	335544468L	transaction *<long>* is *<string>*
isc_trans_invalid	335544469L	transaction marked invalid by I/O error
isc_buf_invalid	335544470L	cache buffer for page *<long>* invalid
isc_indexnotdefined	335544471L	there is no index in table *<string>* with id *<digit>*
isc_login	335544472L	Your user name and password are not defined. Ask your database administrator to set up an InterBase login.
isc_invalid_bookmark	335544473L	invalid bookmark handle
isc_bad_lock_level	335544474L	invalid lock level *<digit>*
isc_relation_lock	335544475L	lock on table *<string>* conflicts with existing lock
isc_record_lock	335544476L	requested record lock conflicts with existing lock
isc_max_idx	335544477L	maximum indexes per table (*<digit>*) exceeded
isc_jrn_enable	335544478L	enable journal for database before starting online dump
isc_old_failure	335544479L	online dump failure. Retry dump
isc_old_in_progress	335544480L	an online dump is already in progress
isc_old_no_space	335544481L	no more disk/tape space. Cannot continue online dump
isc_no_wal_no_jrn	335544482L	journaling allowed only if database has Write-ahead Log
isc_num_old_files	335544483L	maximum number of online dump files that can be specified is 16
isc_wal_file_open	335544484L	error in opening Write-ahead Log file during recovery
isc_bad_stmt_handle	335544485L	invalid statement handle
isc_wal_failure	335544486L	Write-ahead log subsystem failure
isc_walw_err	335544487L	WAL Writer error
isc_logh_small	335544488L	Log file header of *<string>* too small
isc_logh_inv_version	335544489L	Invalid version of log file *<string>*
isc_logh_open_flag	335544490L	Log file *<string>* not latest in the chain but open flag still set
isc_logh_open_flag2	335544491L	Log file *<string>* not closed properly; database recovery may be required
isc_logh_diff_dbname	335544492L	Database name in the log file *<string>* is different
isc_logf_unexpected_eof	335544493L	Unexpected end of log file *<string>* at offset *<long>*
isc_logr_incomplete	335544494L	Incomplete log record at offset *<long>* in log file *<string>*

Table B-5: InterBase Status Array Error Codes (Continued)

Error Code	Number	Message
isc_logr_header_small	335544495L	Log record header too small at offset *<long>* in log file *<string>*
isc_logb_small	335544496L	Log block too small at offset *<long>* in log file *<string>*
isc_wal_illegal_attach	335544497L	Illegal attempt to attach to an uninitialized WAL segment for *<string>*
isc_wal_invalid_wpb	335544498L	Invalid WAL parameter block option *<string>*
isc_wal_err_rollover	335544499L	Cannot roll over to the next log file *<string>*
isc_no_wal	335544500L	database does not use Write-ahead Log
isc_drop_wal	335544501L	cannot drop log file when journaling is enabled
isc_stream_not_defined	335544502L	reference to invalid stream number
isc_wal_subsys_error	335544503L	WAL subsystem encountered error
isc_wal_subsys_corrupt	335544504L	WAL subsystem corrupted
isc_no_archive	335544505L	must specify archive file when enabling long-term journal for databases with round-robin log files
isc_shutinprog	335544506L	database *<string>* shutdown in progress
isc_range_in_use	335544507L	refresh range number *<long>* already in use
isc_range_not_found	335544508L	refresh range number *<long>* not found
isc_charset_not_found	335544509L	CHARACTER SET *<string>* is not defined
isc_lock_timeout	335544510L	lock time-out on wait transaction
isc_prcnotdef	335544511L	procedure *<string>* is not defined
isc_prcmismat	335544512L	parameter mismatch for procedure *<string>*
isc_wal_bugcheck	335544513L	Database *<string>*: WAL subsystem bug for pid *<digit>*\
isc_wal_cant_expand	335544514L	Could not expand the WAL segment for database *<string>*
isc_codnotdef	335544515L	status code *<string>* unknown
isc_xcpnotdef	335544516L	exception *<string>* not defined
isc_except	335544517L	exception *<digit>*
isc_cache_restart	335544518L	restart shared cache manager
isc_bad_lock_handle	335544519L	invalid lock handle
isc_jrn_present	335544520L	long-term journaling already enabled
isc_wal_err_rollover2	335544521L	Unable to roll over; please see InterBase log.
isc_wal_err_logwrite	335544522L	WAL I/O error. Please see InterBase log.
isc_wal_err_jrn_comm	335544523L	WAL writer - Journal server communication error. Please see InterBase log.

Table B-5: InterBase Status Array Error Codes (Continued)

Error Code	Number	Message
isc_wal_err_expansion	335544524L	WAL buffers cannot be increased. Please see InterBase log.
isc_wal_err_setup	335544525L	WAL setup error. Please see InterBase log.
isc_wal_err_ww_sync	335544526L	WAL writer synchronization error for the database *<string>*
isc_wal_err_ww_start	335544527L	Cannot start WAL writer for the database *<string>*
isc_shutdown	335544528L	database *<string>* shutdown
isc_existing_priv_mod	335544529L	cannot modify an existing user privilege
isc_primary_key_ref	335544530L	Cannot delete PRIMARY KEY being used in FOREIGN KEY definition.
isc_primary_key_notnull	335544531L	Column used in a PRIMARY/UNIQUE constraint must be NOT NULL.
isc_ref_cnstrnt_notfound	335544532L	Name of Referential Constraint not defined in constraints table.
isc_foreign_key_notfound	335544533L	Non-existent PRIMARY or UNIQUE KEY specified for FOREIGN KEY.
isc_ref_cnstrnt_update	335544534L	Cannot update constraints (RDB$REF_CONSTRAINTS).
isc_check_cnstrnt_update	335544535L	Cannot update constraints (RDB$CHECK_CONSTRAINTS).
isc_check_cnstrnt_del	335544536L	Cannot delete CHECK constraint entry (RDB$CHECK_CONSTRAINTS)
isc_integ_index_seg_del	335544537L	Cannot delete index segment used by an Integrity Constraint
isc_integ_index_seg_mod	335544538L	Cannot update index segment used by an Integrity Constraint
isc_integ_index_del	335544539L	Cannot delete index used by an Integrity Constraint
isc_integ_index_mod	335544540L	Cannot modify index used by an Integrity Constraint
isc_check_trig_del	335544541L	Cannot delete trigger used by a CHECK Constraint
isc_check_trig_update	335544542L	Cannot update trigger used by a CHECK Constraint
isc_cnstrnt_fld_del	335544543L	Cannot delete column being used in an Integrity Constraint.
isc_cnstrnt_fld_rename	335544544L	Cannot rename column being used in an Integrity Constraint.
isc_rel_cnstrnt_update	335544545L	Cannot update constraints (RDB$RELATION_CONSTRAINTS).

Table B-5: InterBase Status Array Error Codes (Continued)

Error Code	Number	Message
isc_constaint_on_view	335544546L	Cannot define constraints on views
isc_invld_cnstrnt_type	335544547L	internal isc software consistency check (invalid RDB$CONSTRAINT_TYPE)
isc_primary_key_exists	335544548L	Attempt to define a second PRIMARY KEY for the same table
isc_systrig_update	335544549L	cannot modify or erase a system trigger
isc_not_rel_owner	335544550L	only the owner of a table may reassign ownership
isc_grant_obj_notfound	335544551L	could not find table/procedure for GRANT
isc_grant_fld_notfound	335544552L	could not find column for GRANT
isc_grant_nopriv	335544553L	user does not have GRANT privileges for operation
isc_nonsql_security_rel	335544554L	table/procedure has non-SQL security class defined
isc_nonsql_security_fld	335544555L	column has non-SQL security class defined
isc_wal_cache_err	335544556L	Write-ahead Log without shared cache configuration not allowed
isc_shutfail	335544557L	database shutdown unsuccessful
isc_check_constraint	335544558L	Operation violates CHECK constraint *<string>* on view or table
isc_bad_svc_handle	335544559L	invalid service handle
isc_shutwarn	335544560L	database *<string>* shutdown in *<digit>* seconds
isc_wrospbver	335544561L	wrong version of service parameter block
isc_bad_spb_form	335544562L	unrecognized service parameter block
isc_svcnotdef	335544563L	service *<string>* is not defined
isc_no_jrn	335544564L	long-term journaling not enabled
isc_transliteration_failed	335544565L	Cannot transliterate character between character sets
isc_start_cm_for_wal	335544566L	WAL defined; Cache Manager must be started first
isc_wal_ovflow_log_required	335544567L	Overflow log specification required for round-robin log
isc_text_subtype	335544568L	Implementation of text subtype *<digit>* not located.
isc_dsql_error	335544569L	Dynamic SQL Error
isc_dsql_command_err	335544570L	Invalid command
isc_dsql_constant_err	335544571L	Data type for constant unknown
isc_dsql_cursor_err	335544572L	Cursor unknown
isc_dsql_datatype_err	335544573L	Data type unknown
isc_dsql_decl_err	335544574L	Declared cursor already exists
isc_dsql_cursor_update_err	335544575L	Cursor not updatable

Table B-5: InterBase Status Array Error Codes (Continued)

Error Code	Number	Message
isc_dsql_cursor_open_err	335544576L	Attempt to reopen an open cursor
isc_dsql_cursor_close_err	335544577L	Attempt to reclose a closed cursor
isc_dsql_field_err	335544578L	Column unknown
isc_dsql_internal_err	335544579L	Internal error
isc_dsql_relation_err	335544580L	Table unknown
isc_dsql_procedure_err	335544581L	Procedure unknown
isc_dsql_request_err	335544582L	Request unknown
isc_dsql_sqlda_err	335544583L	SQLDA missing or incorrect vresion, or incorrect number/type of variables
isc_dsql_var_count_err	335544584L	Count of columns not equal count of values
isc_dsql_stmt_handle	335544585L	Invalid statement handle
isc_dsql_function_err	335544586L	Function unknown
isc_dsql_blob_err	335544587L	Column is not a BLOB
isc_collation_not_found	335544588L	COLLATION *<string>* is not defined
isc_collation_not_for_charset	335544589L	COLLATION *<string>* is not valid for specified CHARACTER SET
isc_dsql_dup_option	335544590L	Option specified more than once
isc_dsql_tran_err	335544591L	Unknown transaction option
isc_dsql_invalid_array	335544592L	Invalid array reference
isc_dsql_max_arr_dim_exceeded	335544593L	Array declared with too many dimensions
isc_dsql_arr_range_error	335544594L	Illegal array dimension range
isc_dsql_trigger_err	335544595L	Trigger unknown
isc_dsql_subselect_err	335544596L	Subselect illegal in this context
isc_dsql_crdb_prepare_err	335544597L	Cannot prepare a CREATE DATABASE/SCHEMA statement
isc_specify_field_err	335544598L	must specify column name for view select expression
isc_num_field_err	335544599L	number of columns does not match select list
isc_col_name_err	335544600L	Only simple column names permitted for VIEW WITH CHECK OPTION
isc_where_err	335544601L	No WHERE clause for VIEW WITH CHECK OPTION
isc_table_view_err	335544602L	Only one table allowed for VIEW WITH CHECK OPTION

Table B-5: InterBase Status Array Error Codes (Continued)

Error Code	Number	Message
isc_distinct_err	335544603L	DISTINCT, GROUP or HAVING not permitted for VIEW WITH CHECK OPTION
isc_key_field_count_err	335544604L	FOREIGN KEY column count does not match PRIMARY KEY
isc_subquery_err	335544605L	No subqueries permitted for VIEW WITH CHECK OPTION
isc_expression_eval_err	335544606L	expression evaluation not supported
isc_node_err	335544607L	gen.c: node not supported
isc_command_end_err	335544608L	Unexpected end of command
isc_index_name	335544609L	INDEX *<string>*
isc_exception_name	335544610L	EXCEPTION *<string>*
isc_field_name	335544611L	COLUMN *<string>*
isc_token_err	335544612L	Token unknown
isc_union_err	335544613L	union not supported
isc_dsql_construct_err	335544614L	Unsupported DSQL construct
isc_field_aggregate_err	335544615L	column used with aggregate
isc_field_ref_err	335544616L	invalid column reference
isc_order_by_err	335544617L	invalid ORDER BY clause
isc_return_mode_err	335544618L	Return mode by value not allowed for this data type
isc_extern_func_err	335544619L	External functions cannot have more than 10 parameters
isc_alias_conflict_err	335544620L	alias *<string>* conflicts with an alias in the same statement
isc_procedure_conflict_error	335544621L	alias *<string>* conflicts with a procedure in the same statement
isc_relation_conflict_err	335544622L	alias *<string>* conflicts with a table in the same statement
isc_dsql_domain_err	335544623L	Illegal use of keyword VALUE
isc_idx_seg_err	335544624L	segment count of 0 defined for index *<string>*
isc_node_name_err	335544625L	A node name is not permitted in a secondary, shadow, cache or log file name
isc_table_name	335544626L	TABLE *<string>*
isc_proc_name	335544627L	PROCEDURE *<string>*
isc_idx_create_err	335544628L	cannot create index *<string>*

Table B-5: InterBase Status Array Error Codes (Continued)

Error Code	Number	Message
isc_wal_shadow_err	335544629L	Write-ahead Log with shadowing configuration not allowed
isc_dependency	335544630L	there are *<long>* dependencies
isc_idx_key_err	335544631L	too many keys defined for index *<string>*
isc_dsql_file_length_err	335544632L	Preceding file did not specify length, so *<string>* must include starting page number
isc_dsql_shadow_number_err	335544633L	Shadow number must be a positive integer
isc_dsql_token_unk_err	335544634L	Token unknown - line *<long>*, char *<long>*
isc_dsql_no_relation_alias	335544635L	there is no alias or table named *<string>* at this scope level
isc_indexname	335544636L	there is no index *<string>* for table *<string>*
isc_no_stream_plan	335544637L	table *<string>* is not referenced in plan
isc_stream_twice	335544638L	table *<string>* is referenced more than once in plan; use aliases to distinguish
isc_stream_not_found	335544639L	table *<string>* is referenced in the plan but not the from list
isc_collation_requires_text	335544640L	Invalid use of CHARACTER SET or COLLATE
isc_dsql_domain_not_found	335544641L	Specified domain or source column does not exist
isc_index_unused	335544642L	index *<string>* cannot be used in the specified plan
isc_dsql_self_join	335544643L	the table *<string>* is referenced twice; use aliases to differentiate
isc_stream_bof	335544644L	illegal operation when at beginning of stream
isc_stream_crack	335544645L	the current position is on a crack
isc_db_or_file_exists	335544646L	database or file exists
isc_invalid_operator	335544647L	invalid comparison operator for find operation
isc_conn_lost	335544648L	Connection lost to pipe server
isc_bad_checksum	335544649L	bad checksum
isc_page_type_err	335544650L	wrong page type
isc_ext_readonly_err	335544651L	external file could not be opened for output
isc_sing_select_err	335544652L	multiple rows in singleton select
isc_psw_attach	335544653L	cannot attach to password database
isc_psw_start_trans	335544654L	cannot start transaction for password database
isc_invalid_direction	335544655L	invalid direction for find operation

Table B-5: InterBase Status Array Error Codes (Continued)

Error Code	Number	Message
isc_dsql_var_conflict	335544656L	variable *<string>* conflicts with parameter in same procedure
isc_dsql_no_blob_array	335544657L	Array/BLOB/DATE data types not allowed in arithmetic
isc_dsql_base_table	335544658L	*<string>* is not a valid base table of the specified view
isc_duplicate_base_table	335544659L	table *<string>* is referenced twice in view; use an alias to distinguish
isc_view_alias	335544660L	view *<string>* has more than one base table; use aliases to distinguish
isc_index_root_page_full	335544661L	cannot add index, index root page is full.
isc_dsql_blob_type_unknown	335544662L	BLOB SUB_TYPE *<string>* is not defined
isc_req_max_clones_exceeded	335544663L	Too many concurrent executions of the same request
isc_dsql_duplicate_spec	335544664L	duplicate specification of *<string>* - not supported
isc_unique_key_violation	335544665L	violation of PRIMARY or UNIQUE KEY constraint: "*<string>*"
isc_srvr_version_too_old	335544666L	server version too old to support all CREATE DATABASE options
isc_drdb_completed_with_errs	335544667L	drop database completed with errors
isc_dsql_procedure_use_err	335544668L	procedure *<string>* does not return any values
isc_dsql_count_mismatch	335544669L	count of column list and variable list do not match
isc_blob_idx_err	335544670L	attempt to index BLOB column in index *<string>*
isc_array_idx_err	335544671L	attempt to index array column in index *<string>*
isc_key_field_err	335544672L	too few key columns found for index *<string>* (incorrect column name?)
isc_no_delete	335544673L	cannot delete
isc_del_last_field	335544674L	last column in a table cannot be deleted
isc_sort_err	335544675L	sort error
isc_sort_mem_err	335544676L	sort error: not enough memory
isc_version_err	335544677L	too many versions
isc_inval_key_posn	335544678L	invalid key position
isc_no_segments_err	335544679L	segments not allowed in expression index *<string>*
isc_crrp_data_err	335544680L	sort error: corruption in data structure
isc_rec_size_err	335544681L	new record size of *<long>* bytes is too big
isc_dsql_field_ref	335544682L	Inappropriate self-reference of column
isc_req_depth_exceeded	335544683L	request depth exceeded. (Recursive definition?)

Table B-5: InterBase Status Array Error Codes (Continued)

Error Code	Number	Message
isc_no_field_access	335544684L	cannot access column *<string>* in view *<string>*
isc_no_dbkey	335544685L	dbkey not available for multi-table views
isc_jrn_format_err	335544686L	journal file wrong format
isc_jrn_file_full	335544687L	intermediate journal file full
isc_dsql_open_cursor_request	335544688L	The prepare statement identifies a prepare statement with an open cursor
isc_ib_error	335544689L	InterBase error
isc_cache_redef	335544690L	Cache redefined
isc_cache_too_small	335544691L	Cache length too small
isc_log_redef	335544692L	Log redefined
isc_log_too_small	335544693L	Log size too small
isc_partition_too_small	335544694L	Log partition size too small
isc_partition_not_supp	335544695L	Partitions not supported in series of log file specification
isc_log_length_spec	335544696L	Total length of a partitioned log must be specified
isc_precision_err	335544697L	Precision should be greater than 0
isc_scale_nogt	335544698L	Scale cannot be greater than precision
isc_expec_short	335544699L	Short integer expected
isc_expec_long	335544700L	Long integer expected
isc_expec_ushort	335544701L	Unsigned short integer expected
isc_like_escape_invalid	335544702L	Invalid ESCAPE sequence
isc_svcnoexe	335544703L	service *<string>* does not have an associated executable
isc_net_lookup_err	335544704L	Network lookup failure for host "*<string>*"
isc_service_unknown	335544705L	Undefined service *<string>*/*<string>*
isc_host_unknown	335544706L	Host unknown
isc_grant_nopriv_on_base	335544707L	user does not have GRANT privileges on base table/view for operation
isc_dyn_fld_ambiguous	335544708L	Ambiguous column reference.
isc_dsql_agg_ref_err	335544709L	Invalid aggregate reference
isc_complex_view	335544710L	navigational stream *<long>* references a view with more than one base table.
isc_unprepared_stmt	335544711L	attempt to execute an unprepared dynamic SQL statement
isc_expec_positive	335544712L	Positive value expected.

Table B-5: InterBase Status Array Error Codes (Continued)

Error Code	Number	Message
isc_dsql_sqlda_value_err	335544713L	Incorrect values within SQLDA structure
isc_invalid_array_id	335544714L	invalid blob id
isc_ext_file_uns_op	335544715L	Operation not supported for EXTERNAL FILE table *<string>*

Appendix C

System Tables and Views

This appendix describes the InterBase system tables and SQL system views.

Overview

The InterBase system tables contain and track metadata. InterBase automatically creates system tables when a database is created. Each time a user creates or modifies metadata through data definition, the SQL data definition utility automatically updates the system tables.

The SQL system views provide information about existing integrity constraints for a database. They are a subset of views defined in the SQL-92 standard. System views are created separately by running an **isql** script after database definition.

To see system tables, use this **isql** command:

```
SHOW SYSTEM TABLES;
```

The following **isql** command lists system views along with database views:

```
SHOW VIEWS;
```

System Tables

This table lists the InterBase system tables. The names of system tables and their columns start with RDB$.

Table C-1: System Tables

RDB$CHARACTER_SETS	RDB$LOG_FILES
RDB$COLLATIONS	RDB$PAGES
RDB$CHECK_CONSTRAINTS	RDB$PROCEDURE_PARAMETERS
RDB$DATABASE	RDB$PROCEDURES

Table C-1: System Tables (Continued)

RDB$DEPENDENCIES	RDB$REF_CONSTRAINTS
RDB$EXCEPTIONS	RDB$RELATION_CONSTRAINTS
RDB$FIELD_DIMENSIONS	RDB$RELATION_FIELDS
RDB$FIELDS	RDB$RELATIONS
RDB$FILES	RDB$SECURITY_CLASSES
RDB$FILTERS	RDB$TRANSACTIONS
RDB$FORMATS	RDB$TRIGGER_MESSAGES
RDB$FUNCTION_ARGUMENTS	RDB$TRIGGERS
RDB$FUNCTIONS	RDB$TYPES
RDB$GENERATORS	RDB$USER_PRIVILEGES
RDB$INDEX_SEGMENTS	RDB$VIEW_RELATIONS
RDB$INDICES	

System Views

This table lists SQL system views. Since they are not automatically defined by InterBase, the names of system views and their columns do not start with RDB$.

Table C-2: System Views

CHECK_CONSTRAINTS	REFERENTIAL_CONSTRAINTS
CONSTRAINTS_COLUMN_USAGE	TABLE_CONSTRAINTS

RDB$CHARACTER_SETS

RDB$CHARACTER_SETS describes the valid character sets available in InterBase.

Table C-3: RDB$CHARACTER_SETS

Column Name	Data Type	Length	Description
RDB$CHARACTER_SET_NAME	CHAR	31	Name of a character set that InterBase recognizes.
RDB$FORM_OF_USE	CHAR	31	Reserved for internal use. Subtype 2.
RDB$NUMBER_OF_CHARACTERS	INTEGER		Number of characters in a particular character set. For example, the set of Japanese characters.
RDB$DEFAULT_COLLATE_NAME	CHAR	31	Subtype 2. Default collation sequence for the character set.

Table C-3: RDB$CHARACTER_SETS (Continued)

Column Name	Data Type	Length	Description
RDB$CHARACTER_ SET_ID	SMALLINT		A unique identification for the character set.
RDB$SYSTEM_FLAG	SMALLINT		Indicates whether the character set is: • User-defined (value of 0) • System-defined (value of 1)
RDB$DESCRIPTION	BLOB	80	Subtype text. Contains a user-written description of the character set.
RDB$FUNCTION_NAME	CHAR	31	Reserved for internal use. Subtype 2.
RDB$BYTES_PER_ CHARACTER	SMALLINT		Size of character in bytes.

RDB$CHECK_CONSTRAINTS

RDB$CHECK_CONSTRAINTS stores database integrity constraint information for CHECK constraints. In addition, the table stores information for constraints implemented with NOT NULL.

Table C-4: RDB$CHECK_CONSTRAINTS

Column Name	Data Type	Length	Description
RDB$CONSTRAINT_ NAME	CHAR	31	Subtype 2. Name of a CHECK or NOT NULL constraint.
RDB$TRIGGER_NAME	CHAR	31	Subtype 2. Name of the trigger that enforces the CHECK constraint. For a NOT NULL constraint, name of the source column in RDB$RELATION_FIELDS.

RDB$COLLATIONS

RDB$COLLATIONS records the valid collating sequences available for use in InterBase.

Table C-5: RDB$COLLATIONS

Column Name	Data Type	Length	Description
RDB$COLLATION_NAME	CHAR	31	Name of a valid collation sequence in InterBase.
RDB$COLLATION_ID	SMALLINT		Unique identifier for the collation sequence.

Table C-5: RDB$COLLATIONS (Continued)

Column Name	Data Type	Length	Description
RDB$CHARACTER_ SET_ID	SMALLINT		Identifier of the underlying character set of this collation sequence. Required before collation can proceed; determines which character set is in use. Corresponds to the RDB$CHARACTER_SET_ID column in the RDB$CHARACTER_SETS table.
RDB$COLLATION_ ATTRIBUTES	SMALLINT		Reserved for internal use.
RDB$SYSTEM_FLAG	SMALLINT		Indicates whether the generator is: • User-defined (value of 0) • System-defined (value greater than 0)
RDB$DESCRIPTION	BLOB	80	Subtype Text. Contains a user-written description of the collation sequence.
RDB$FUNCTION_NAME	CHAR	31	Reserved for internal use.

RDB$DATABASE

RDB$DATABASE defines a database.

Table C-6: RDB$DATABASE

Column Name	Data Type	Length	Description
RDB$DESCRIPTION	BLOB	80	Subtype Text. Contains a user-written description of the database. When a comment is included in a CREATE or ALTER SCHEMA | DATABASE statement, **isql** writes to this column.
RDB$RELATION_ID	SMALLINT		For internal use by InterBase.
RDB$SECURITY_CLASS	CHAR	31	Subtype 2. Security class defined in the RDB$SECURITY_CLASSES table. The access control limits described in the named security class apply to all database usage.
RDB$CHARACTER_ SET_NAME	CHAR	31	Subtype 2. Name of character set.

RDB$DEPENDENCIES

RDB$DEPENDENCIES keeps track of the tables and columns upon which other system objects depend. These objects include views, triggers, and computed columns. InterBase uses this table to ensure that a column or table cannot be deleted if it is used by any other object.

Table C-7: RDB$DEPENDENCIES

Column Name	Data Type	Length	Description
RDB$DEPENDENT_NAME	CHAR	31	Subtype 2. Names the object this table tracks: a view, trigger, or computed column.
RDB$DEPENDED_ON_NAME	CHAR	31	Subtype 2. Names the table referenced by the object named above.
RDB$FIELD_NAME	CHAR	31	Subtype 2. Names the column referenced by the object named above.
RDB$DEPENDENT_TYPE	SMALLINT		Describes the object type of the object referenced in the RDB$DEPENDENT_NAME column. Type codes (RDB$TYPES): • 0 - table • 1 - view • 2 - trigger • 3 - computed_field • 4 - validation • 5 - procedure • 6 - expression_index • 7 - exception • 8 - user • 9 - field • 10 - index All other values are reserved for future use.
RDB$DEPENDED_ON_TYPE	SMALLINT		Describes the object type of the object referenced in the RDB$DEPENDED_ON_NAME column. Type codes (RDB$TYPES): • 0 - table • 1 - view • 2 - trigger • 3 - computed_field • 4 - validation • 5 - procedure • 6 - expression_index • 7 - exception • 8 - user • 9 - field • 10 - index All other values are reserved for future use.

RDB$EXCEPTIONS

RDB$EXCEPTIONS describes error conditions related to stored procedures, including user-defined exceptions.

Table C-8: RDB$EXCEPTIONS

Column Name	Data Type	Length	Description
RDB$EXCEPTION_NAME	CHAR	31	Subtype 2. Exception name.
RDB$EXCEPTION_ NUMBER	INTEGER		Number for the exception.
RDB$MESSAGE	VARCHAR	78	Text of exception message.
RDB$DESCRIPTION	BLOB	80	Subtype Text. Text description of the exception.
RDB$SYSTEM_FLAG	SMALLINT		Indicates whether the exception is: • User-defined (value of 0) • System-defined (value greater than 0)

RDB$FIELD_DIMENSIONS

RDB$FIELD_DIMENSIONS describes each dimension of an array column.

Table C-9: RDB$FIELD_DIMENSIONS

Column Name	Data Type	Length	Description
RDB$FIELD_NAME	CHAR	31	Subtype 2. Names the array column described by this table. The column name must exist in the RDB$FIELD_NAME column of RDB$FIELDS.
RDB$DIMENSION	SMALLINT		Identifies one dimension of the ARRAY column. The first dimension is identified by the integer 0.
RDB$LOWER_BOUND	INTEGER		Indicates the lower bound of the previously specified dimension.
RDB$UPPER_BOUND	INTEGER		Indicates the upper bound of the previously specified dimension.

RDB$FIELDS

RDB$FIELDS defines the characteristics of a column. Each domain or column has a corresponding row in RDB$FIELDS. Columns are added to tables by means of an entry in the RDB$RELATION_FIELDS table, which describes local characteristics.

For domains, RDB$FIELDS includes domain name, null status, and default values. SQL columns are defined in RDB$RELATION_FIELDS. For both domains and simple columns, RDB$RELATION_FIELDS may contain default and null status information.

Table C-10: RDB$FIELDS

Column Name	Data Type	Length	Description
RDB$FIELD_NAME	CHAR	31	Unique name of a domain or system-assigned name for a column, starting with SQL*nnn*. The actual column names are stored in the RDB$FIELD_SOURCE column of RDB$RELATION_FIELDS.
RDB$QUERY_NAME	CHAR	31	Not used for SQL objects.
RDB$VALIDATION_BLR	BLOB	80	Not used for SQL objects.
RDB$VALIDATION_ SOURCE	BLOB	80	Not used for SQL objects.
RDB$COMPUTED_BLR	BLOB	80	Subtype BLR. For computed columns, contains the BLR (Binary Language Representation) of the expression the database evaluates at the time of execution.
RDB$COMPUTED_ SOURCE	BLOB	80	Subtype Text. For computed columns, contains the original CHAR source expression for the column.
RDB$DEFAULT_VALUE	BLOB	80	Stores default rule. Subtype BLR.
RDB$DEFAULT_SOURCE	BLOB	80	Subtype Text. SQL description of a default value.
RDB$FIELD_LENGTH	SMALLINT		Contains the length of the column defined in this row. Non-CHAR column lengths are: • D_FLOAT - 8 • DOUBLE - 8 • DATE - 8 • BLOB - 8 • SHORT - 2 • LONG - 4 • QUAD - 8 • FLOAT - 4
RDB$FIELD_SCALE	SMALLINT		Stores negative scale for numeric and decimal types.

Table C-10: RDB$FIELDS (Continued)

Column Name	Data Type	Length	Description
RDB$FIELD_TYPE	SMALLINT		Specifies the data type of the column being defined. Changing the value of this column automatically changes the data type for all columns based on the column being defined. Valid values are: • SMALLINT - 7 • INTEGER - 8 • QUAD - 9 • FLOAT - 10 • D_FLOAT - 11 • CHAR - 14 • DOUBLE - 27 • DATE - 35 • VARCHAR - 37 • BLOB - 261 Restrictions: • The value of this column cannot be changed to or from BLOB. • Non-numeric data causes a conversion error in a column changed from CHAR to numeric. Changing data from CHAR to numeric and back again adversely affects index performance. For best results, delete and re-create indexes when making this type of change.
RDB$FIELD_SUB_TYPE	SMALLINT		Used to distinguish types of BLOBs. Predefined subtypes for BLOB columns are: • 0 - unspecified • 1 - text • 2 - BLR (Binary Language Representation) • 3 - access control list • 4 - reserved for future use • 5 - encoded description of a table's current metadata • 6 - description of multi-database transaction that finished irregularly Predefined subtypes for CHAR columns are: • 0 - unspecified • 1 - fixed BINARY data Corresponds to the RDB$FIELD_SUB_TYPE column in the RDB$COLLATIONS table.
RDB$MISSING_VALUE	BLOB	80	Not used for SQL objects.
RDB$MISSING_SOURCE	BLOB	80	Not used for SQL objects.
RDB$DESCRIPTION	BLOB	80	Subtype Text. Contains a user-written description of the column being defined.
RDB$SYSTEM_FLAG	SMALLINT		For system tables.

Table C-10: RDB$FIELDS (Continued)

Column Name	Data Type	Length	Description
RDB$QUERY_HEADER	BLOB	80	Not used for SQL objects.
RDB$SEGMENT_ LENGTH	SMALLINT		Used for BLOB columns only; a non-binding suggestion for the length of BLOB buffers.
RDB$EDIT_STRING	VARCHAR	125	Not used for SQL objects.
RDB$EXTERNAL_ LENGTH	SMALLINT		Length of the column as it exists in an external table. If the column is not in an external table, this value is 0.
RDB$EXTERNAL_SCALE	SMALLINT		Scale factor for an external column of an integer data type. The scale factor is the power of 10 by which the integer is multiplied.
RDB$EXTERNAL_TYPE	SMALLINT		Indicates the data type of the column as it exists in an external table. Valid values are: • SMALLINT - 7 • INTEGER - 8 • QUAD - 9 • FLOAT - 10 • D_FLOAT - 11 • CHAR - 14 • DOUBLE - 27 • DATE - 35 • VARCHAR - 37 • ‘C’ string (null terminated text) - 40 • BLOB - 261
RDB$DIMENSIONS	SMALLINT		For an ARRAY data type, specifies the number of dimensions in the array. For a non-array column, the value is 0.
RDB$NULL_FLAG	SMALLINT		Indicates whether a column can contain a NULL value. Values: • Empty: Can contain NULL values. • 1: Cannot contain NULL values.
RDB$CHARACTER_ LENGTH	SMALLINT		Length of character in bytes.
RDB$COLLATION_ID	SMALLINT		Unique identifier for the collation sequence.
RDB$CHARACTER_ SET_ID	SMALLINT		ID indicating character set for the character or BLOB columns. Joins to the CHARACTER_SET_ID column of the RDB$CHARACTER_SETS system table.

RDB$FILES

RDB$FILES lists the secondary files and shadow files for a database.

Table C-11: RDB$FILES

Column Name	Data Type	Length	Description
RDB$FILE_NAME	VARCHAR	253	Names either a secondary file or a shadow file for the database.
RDB$FILE_SEQUENCE	SMALLINT		*Either* the order that secondary files are to be used in the database *or* the order of files within a shadow set.
RDB$FILE_START	INTEGER		Specifies the starting page number for a secondary file or shadow file.
RDB$FILE_LENGTH	INTEGER		Specifies the file length in blocks.
RDB$FILE_FLAGS	SMALLINT		Reserved for system use.
RDB$SHADOW_NUMBER	SMALLINT		Set number of a shadow file. Set number indicates to which shadow set the file belongs. If the value of this column is 0 or missing, InterBase assumes the file being defined is a secondary file, not a shadow file.

RDB$FILTERS

RDB$FILTERS tracks information about a BLOB filter.

Table C-12: RDB$FILTERS

Column Name	Data Type	Length	Description
RDB$FUNCTION_NAME	CHAR	31	Unique name for the filter defined by this row.
RDB$DESCRIPTION	BLOB	80	Subtype Text. Contains a user-written description of the filter being defined.
RDB$MODULE_NAME	VARCHAR	253	Names the library where the filter executable is stored.
RDB$ENTRYPOINT	CHAR	31	The entry point within the filter library for the BLOB filter being defined.
RDB$INPUT_SUB_TYPE	SMALLINT		The BLOB subtype of the input data.
RDB$OUTPUT_SUB_TYPE	SMALLINT		The BLOB subtype of the output data.
RDB$SYSTEM_FLAG	SMALLINT		Indicates whether the filter is: • User-defined (value of 0) • System-defined (value greater than 0)

RDB$FORMATS

RDB$FORMATS keeps track of the formats of the columns in a table. InterBase assigns the table a new format number at each change to a column definition. This table allows existing application programs to access a changed table, without needing to be recompiled.

Table C-13: RDB$FORMATS

Column Name	Data Type	Length	Description
RDB$RELATION_ID	SMALLINT		Names a table that exists in RDB$RELATIONS.
RDB$FORMAT	SMALLINT		Specifies the format number of the table. A table can have any number of different formats, depending on the number of updates to the table.
RDB$DESCRIPTOR	BLOB	80	Subtype Format. Lists each column in the table, along with its data type, length, and scale (if applicable).

RDB$FUNCTION_ARGUMENTS

RDB$FUNCTION_ARGUMENTS defines the attributes of a function argument.

Table C-14: RDB$FUNCTION_ARGUMENTS

Column Name	Data Type	Length	Description
RDB$FUNCTION_NAME	CHAR	31	Unique name of the function with which the argument is associated. Must correspond to a function name in RDB$FUNCTIONS.
RDB$ARGUMENT_POSITION	SMALLINT		Position of the argument described in the RDB$FUNCTION_NAME column in relation to the other arguments.
RDB$MECHANISM	SMALLINT		Specifies whether the argument is passed by value (value of 0) or by reference (value of 1).
RDB$FIELD_TYPE	SMALLINT		Data type of the argument being defined. Valid values: • SMALLINT - 7 • INTEGER - 8 • QUAD - 9 • FLOAT - 10 • D_FLOAT - 11 • CHAR - 14 • DOUBLE - 27 • DATE - 35 • VARCHAR - 37 • BLOB - 261

Table C-14: RDB$FUNCTION_ARGUMENTS (Continued)

Column Name	Data Type	Length	Description
RDB$FIELD_ SCALE	SMALLINT		Scale factor for an argument that has an integer data type. The scale factor is the power of 10 by which the integer is multiplied.
RDB$FIELD_ LENGTH	SMALLINT		Contains the length of the argument defined in this row. Valid column lengths: • SMALLINT - 2 • INTEGER - 4 • QUAD - 8 • FLOAT - 4 • D_FLOAT - 8 • DOUBLE - 8 • DATE - 8 • BLOB - 8
RDB$FIELD_ SUBTYPE	SMALLINT		Reserved for future use.
RDB$CHARACTER _SET_ID	SMALLINT		Unique numeric identifier for a character set.

RDB$FUNCTIONS

RDB$FUNCTIONS defines a user-defined function.

Table C-15: RDB$FUNCTIONS

Column Name	Data Type	Length	Description
RDB$FUNCTION_NAME	CHAR	31	Unique name for a function.
RDB$FUNCTION_TYPE	SMALLINT		Reserved for future use.
RDB$QUERY_NAME	CHAR	31	Alternate name for the function that can be used in **isql**.
RDB$DESCRIPTION	BLOB	80	Subtype Text. Contains a user-written description of the function being defined.
RDB$MODULE_NAME	VARCHAR	253	Names the function library where the executable function is stored.
RDB$ENTRYPOINT	CHAR	31	Entry point within the function library for the function being defined.
RDB$RETURN_ ARGUMENT	SMALLINT		Position of the argument returned to the calling program. This position is specified in relation to other arguments.
RDB$SYSTEM_FLAG	SMALLINT		Indicates whether the function is: • User-defined (value of 0) • System-defined (value of 1)

RDB$GENERATORS

RDB$GENERATORS stores information about generators, which provide the ability to generate a unique identifier for a table.

Table C-16: RDB$GENERATORS

Column Name	Data Type	Length	Description
RDB$GENERATOR_ NAME	CHAR	31	Name of the table to contain the unique identifier produced by the number generator.
RDB$GENERATOR_ID	SMALLINT		Unique system-assigned ID number for the generator.
RDB$SYSTEM_FLAG	SMALLINT		Indicates whether the generator is: • User-defined (value of 0) • System-defined (value greater than 0)

RDB$INDEX_SEGMENTS

RDB$INDEX_SEGMENTS specifies the columns that comprise an index for a table. Modifying these rows corrupts rather than changes an index unless the RDB$INDICES row is deleted and re-created in the same transaction.

Table C-17: RDB$INDEX_SEGMENTS

Column Name	Data Type	Length	Description
RDB$INDEX_NAME	CHAR	31	The index associated with this index segment. If the value of this column changes, the RDB$INDEX_NAME column in RDB$INDICES must also be changed.
RDB$FIELD_NAME	CHAR	31	The index segment being defined. The value of this column must match the value of the RDB$FIELD_NAME column in RDB$RELATION_FIELDS.
RDB$FIELD_POSITION	SMALLINT		Position of the index segment being defined. Corresponds to the sort order of the index.

RDB$INDICES

RDB$INDICES defines the index structures that allow InterBase to locate rows in the database more quickly. Because InterBase provides both simple indexes (a single-key column) and multi-segment indexes (multiple-key columns), each index defined in this table must have corresponding occurrences in the RDB$INDEX_SEGMENTS table.

Table C-18: RDB$INDICES

Column Name	Data Type	Length	Description
RDB$INDEX_NAME	CHAR	31	Names the index being defined. If the value of this column changes, change its value in the RDB$INDEX_SEGMENTS table.
RDB$RELATION_NAME	CHAR	31	Names the table associated with this index. The table must be defined in the RDB$RELATIONS table.
RDB$INDEX_ID	SMALLINT		Contains an internal identifier for the index being defined. Do *not* write to this column.
RDB$UNIQUE_FLAG	SMALLINT		Specifies whether the index allows duplicate values. Values: • 0 - allows duplicate values • 1 - does not allow duplicate values Eliminate duplicates before creating a unique index.
RDB$DESCRIPTION	BLOB	80	Subtype Text. User-written description of the index. When including a comment in a CREATE INDEX or ALTER INDEX statement, **isql** writes to this column.
RDB$SEGMENT_COUNT	SMALLINT		Number of segments in the index. A value of 1 indicates a simple index.
RDB$INDEX_INACTIVE	SMALLINT		Indicates whether the index is: • Active (value of 0) • Inactive (value of 2)
RDB$INDEX_TYPE	SMALLINT		Reserved for future use.
RDB$FOREIGN_KEY	CHAR	31	Name of FOREIGN KEY constraint for which the index is implemented.
RDB$SYSTEM_FLAG	SMALLINT		Indicates whether the index is: • User-defined (value of 0) • System-defined (value greater than 0)
RDB$EXPRESSION_BLR	BLOB	80	Subtype BLR. Contains the BLR (Binary Language Representation) for the expression, evaluated by the database at execution time. Used for PC semantics.

Table C-18: RDB$INDICES (Continued)

Column Name	Data Type	Length	Description
RDB$EXPRESSION_ SOURCE	BLOB	80	Subtype Text. Contains original text source for the column. Used for PC semantics.
RDB$STATISTICS	DOUBLE PRECI-SION		Selectivity factor for the index. Index selectivity, a measure of uniqueness for indexed columns, is used by the optimizer to choose an access strategy for a query.

RDB$LOG_FILES

RDB$LOG_FILES stores WAL protocol file information for a database.

Table C-19: RDB$LOG_FILES

Column Name	Data Type	Length	Description
RDB$FILE_NAME	VARCHAR	253	Name of the current log file.
RDB$FILE_SEQUENCE	SMALLINT		Log file sequence file number.
RDB$FILE_LENGTH	INTEGER		Log file size in kilobytes.
RDB$FILE_PARTITIONS	SMALLINT		Number of log file partitions.
RDB$FILE_P_OFFSET	INTEGER		
RDB$FILE_FLAGS	SMALLINT		Flags for RBD$LOG_FILES: • 1 - LOG_serial. Any serial log file, including default log configuration and overflow files. • 2 - LOG_default. If log file name is not specified, the default is *database-name.log.n.* • 4 - LOG_raw. On raw device. • 8 - LOG_overflow. Overflow files.

RDB$PAGES

RDB$PAGES keeps track of each page allocated to the database. Modifying this table in any way corrupts a database.

Table C-20: RDB$PAGES

Column Name	Data Type	Length	Description
RDB$PAGE_NUMBER	INTEGER		The physically allocated page number.
RDB$RELATION_ID	SMALLINT		Identifier number of the table for which this page is allocated.

Table C-20: RDB$PAGES (Continued)

Column Name	Data Type	Length	Description
RDB$PAGE_SEQUENCE	INTEGER		The sequence number of this page in the table to other pages allocated for the previously identified table.
RDB$PAGE_TYPE	SMALLINT		Describes the type of page. This information is for system use only.

RDB$PROCEDURE_PARAMETERS

RDB$PROCEDURE_PARAMETERS stores information about each parameter for each of a database's procedures.

Table C-21: RDB$PROCEDURE_PARAMETERS

Column Name	Data Type	Length	Description
RDB$PARAMETER_ NAME	CHAR	31	Parameter name.
RDB$PROCEDURE_ NAME	CHAR	31	Name of the procedure in which the parameter is used.
RDB$PARAMETER_ NUMBER	SMALLINT		Parameter sequence number.
RDB$PARAMETER_TYPE	SMALLINT		Parameter data type. Values: • 0 = input • 1 = output
RDB$FIELD_SOURCE	CHAR	31	Global column name.
RDB$DESCRIPTION	BLOB	80	Subtype Text. User-written description of the parameter.
RDB$SYSTEM_FLAG	SMALLINT		Indicates whether the parameter is: • User-defined (value of 0) • System-defined (value greater than 0)

RDB$PROCEDURES

RDB$PROCEDURES stores information about a database's stored procedures.

Table C-22: RDB$PROCEDURES

Column Name	Data Type	Length	Description
RDB$PROCEDURE_ NAME	CHAR	31	Procedure name.

Table C-22: RDB$PROCEDURES (Continued)

Column Name	Data Type	Length	Description
RDB$PROCEDURE_ID	SMALLINT		Procedure number.
RDB$PROCEDURE_ INPUTS	SMALLINT		Number of input parameters.
PROCEDURE_ OUTPUTS	SMALLINT		Number of output parameters.
RDB$DESCRIPTION	BLOB	80	Subtype Text. User-written description of the procedure.
RDB$PROCEDURE_ SOURCE	BLOB	80	Subtype Text. Source code for the procedure.
RDB$PROCEDURE_BLR	BLOB	80	Subtype BLR. BLR (Binary Language Representation) of the procedure source.
RDB$SECURITY_CLASS	CHAR	31	Security class of the procedure.
RDB$OWNER_NAME	CHAR	31	User who created the procedure. (Owner for SQL security purposes.)
RDB$RUNTIME	BLOB	80	Subtype Summary. Describes procedure metadata. Used for performance enhancement.
RDB$SYSTEM_FLAG	SMALLINT		Indicates whether the procedure is: • User-defined (value of 0) • System-defined (value greater than 0)

RDB$REF_CONSTRAINTS

RDB$REF_CONSTRAINTS stores referential integrity constraint information.

Table C-23: RDB$REF_CONSTRAINTS

Column Name	Data Type	Length	Description
RDB$CONSTRAINT_ NAME	CHAR	31	Name of a referential constraint.
RDB$CONST_NAME_UQ	CHAR	31	Name of a referenced PRIMARY KEY or UNIQUE constraint.
RDB$MATCH_OPTION	CHAR	7	Reserved for later use. Currently defaults to FULL.
RDB$UPDATE_RULE	CHAR	11	Reserved for later use. Currently defaults to RESTRICT.
RDB$DELETE_RULE	CHAR	11	Reserved for later use. Currently defaults to RESTRICT.

RDB$RELATION_CONSTRAINTS

RDB$RELATION_CONSTRAINTS stores information about integrity constraints for tables.

Table C-24: RDB$RELATION_CONSTRAINTS

Column Name	Data Type	Length	Description
RDB$CONSTRAINT_ NAME	CHAR	31	Name of a table constraint.
RDB$CONSTRAINT_ TYPE	CHAR	11	Type of table constraint. Types: • PRIMARY KEY • UNIQUE • FOREIGN KEY • CHECK • NOT NULL
RDB$RELATION_NAME	CHAR	31	Name of the table for which the constraint is defined.
RDB$DEFERRABLE	CHAR	3	Reserved for later use. Currently defaults to No.
RDB$INITIALLY_ DEFERRED	CHAR	3	Reserved for later use. Currently defaults to No.
RDB$INDEX_NAME	CHAR	31	Name of the index used by UNIQUE, PRIMARY KEY, or FOREIGN KEY constraints.

RDB$RELATION_FIELDS

For database tables, RDB$RELATION_FIELDS lists columns and describes column characteristics for domains.

SQL columns are defined in RDB$RELATION_FIELDS. The column name is correlated in the RDB$FIELD_SOURCE column to an underlying entry in RDB$FIELDS that contains a system name ("SQL$<*n*>"). This entry includes information about column type, length, etc. For both domains and simple columns, this table may contain default and nullability information.

Table C-25: RDB$RELATION_FIELDS

Column Name	Data Type	Length	Description
RDB$FIELD_NAME	CHAR	31	Name of the column whose characteristics being defined. The combination of the values in this column and in the RDB$RELATION_NAME column in this table must be unique.

Table C-25: RDB$RELATION_FIELDS (Continued)

Column Name	Data Type	Length	Description
RDB$RELATION_NAME	CHAR	31	Table to which a particular column belongs. A table with this name must appear in RDB$RELATIONS. The combination of the values in this column and in the RDB$FIELD column in this table must be unique.
RDB$FIELD_SOURCE	CHAR	31	The name for this column in the RDB$FIELDS table. If the column is based on a domain, contains the domain name.
RDB$QUERY_NAME	CHAR	31	Alternate column name for use in **isql**; supersedes the value in RDB$FIELDS.
RDB$BASE_FIELD	CHAR	31	Views only: The name of the column from RDB$FIELDS in a table or view that is the base for a view column being defined. For the base column: • RDB$BASE_FIELD provides the column name. • RDB$VIEW_CONTEXT, a column in this table, provides the source table name.
RDB$EDIT_STRING	VARCHAR	125	Not used in SQL.
RDB$FIELD_POSITION	SMALLINT		The position of the column in relation to other columns: • **isql** obtains the ordinal position for displaying column values when printing rows from this column. • **gpre** uses the column order for SELECT and INSERT statements. If two or more columns in the same table have the same value for this column, those columns appear in random order.
RDB$QUERY_HEADER	BLOB	80	Not used in SQL.
RDB$UPDATE_FLAG	SMALLINT		Not used by InterBase. Included for compatibility with other DSRI-based systems.
RDB$FIELD_ID	SMALLINT		Identifier for use in BLR (Binary Language Representation) to name the column. Because this identifier changes during backup and restoration of the database, try to use it in transient requests only. Do *not* modify this column.
RDB$VIEW_CONTEXT	SMALLINT		Alias used to qualify view columns by specifying the table location of the base column. It must have the same value as the alias used in the view BLR (Binary Language Representation) for this context stream.
RDB$DESCRIPTION	BLOB	80	Subtype Text. User-written description of the column being defined.

Table C-25: RDB$RELATION_FIELDS (Continued)

Column Name	Data Type	Length	Description
RDB$DEFAULT_VALUE	BLOB	80	Subtype BLR. BLR (Binary Language Representation) for default clause.
RDB$SYSTEM_FLAG	SMALLINT		Indicates whether the column is: • User-defined (value of 0) • System-defined (value greater than 0)
RDB$SECURITY_CLASS	CHAR	31	Names a security class defined in the RDB$SECURITY_CLASSES table. The access restrictions defined by this security class apply to all users of this column.
RDB$COMPLEX_NAME	CHAR	31	Reserved for future use.
RDB$NULL_FLAG	SMALLINT		Indicates whether the column may contain NULLs.
RDB$DEFAULT_ SOURCE	BLOB	80	Subtype Text. SQL source to define defaults.
RDB$COLLATION_ID	SMALLINT		Unique identifier for the collation sequence.

RDB$RELATIONS

RDB$RELATIONS defines some of the characteristics of tables and views. Other characteristics, such as the columns included in the table and a description of each column, are stored in the RDB$RELATION_FIELDS table.

Table C-26: RDB$RELATIONS

Column Name	Data Type	Length	Description
RDB$VIEW_BLR	BLOB	80	Subtype BLR. For a view, contains the BLR (Binary Language Representation) of the query InterBase evaluates at the time of execution.
RDB$VIEW_SOURCE	BLOB	80	Subtype Text. For a view, contains the original source query for the view definition.
RDB$_DESCRIPTION	BLOB	80	Subtype Text. Contains a user-written description of the table being defined.
RDB$RELATION_ID	SMALLINT		Contains the internal identification number used in BLR (Binary Language Representation) requests. Do *not* modify this column.
RDB$SYSTEM_FLAG	SMALLINT		Indicates the contents of a table, either: • User-data (value of 0). • System information (value greater than 0) Do *not* set this column to 1 when creating tables.

Table C-26: RDB$RELATIONS (Continued)

Column Name	Data Type	Length	Description
RDB$DBKEY_LENGTH	SMALLINT		Length of the database key. Values: • For tables: 8 • For views: 8 times the number of tables referenced in the view definition Do *not* modify the value of this column.
RDB$FORMAT	SMALLINT		For InterBase internal use only. Do *not* modify.
RDB$FIELD_ID	SMALLINT		The number of columns in the table. This column is maintained by InterBase. Do *not* modify the value of this column.
RDB$RELATION_NAME	CHAR	31	The unique name of the table defined by this row.
RDB$SECURITY_CLASS	CHAR	31	Security class defined in the RDB$SECURITY_CLASSES table. Access controls defined in the security class apply to all uses of this table.
RDB$EXTERNAL_FILE	VARCHAR	253	The file in which the external table is stored. An external file can be either an Apollo AEGIS stream file or a VAX RMS file. If blank, the table does not correspond to an external file.
RDB$RUNTIME	BLOB	80	Subtype Summary. Describes table metadata. Used for performance enhancement.
RDB$EXTERNAL_ DESCRIPTION	BLOB	80	Subtype EXTERNAL_FILE_DESCRIPTION. User-written description of the external file.
RDB$OWNER_NAME	CHAR	31	Identifies the creator of the table or view. The creator is considered the owner for SQL security (GRANT/ REVOKE) purposes.
RDB$DEFAULT_CLASS	CHAR	31	Default security class that InterBase applies to columns newly added to a table using the SQL security system.
RDB$FLAGS	SMALLINT		

RDB$SECURITY_CLASSES

RDB$SECURITY_CLASSES defines access control lists and associates them with databases, tables, views, and columns in tables and views. For all SQL objects, the information in this table is duplicated in the RDB$USER_PRIVILEGES system table.

Table C-27: RDB$SECURITY_CLASSES

Column Name	Data Type	Length	Description
RDB$SECURITY_CLASS	CHAR	31	Security class being defined. If the value of this column changes, change its name in the RDB$SECURITY_CLASS column in RDB$_DATABASE, RDB$RELATIONS, and RDB$RELATION_FIELDS.
RDB$ACL	BLOB	80	Subtype ACL. Access control list that specifies users and the privileges granted to those users.
RDB$DESCRIPTION	BLOB	80	Subtype Text. User-written description of the security class being defined.

RDB$TRANSACTIONS

RDB$TRANSACTIONS keeps track of all multi-database transactions.

Table C-28: RDB$TRANSACTIONS

Column Name	Data Type	Length	Description
RDB$TRANSACTION_ID	INTEGER		Identifies the multi-database transaction being described.
RDB$TRANSACTION_STATE	SMALLINT		Indicates the state of the transaction. Valid values are: • 0 - limbo • 1 - committed • 2 - rolled back
RDB$TIMESTAMP	DATE		Reserved for future use.
RDB$TRANSACTION_DESCRIPTION	BLOB	80	Subtype TRANSACTION_DESCRIPTION. Describes a prepared multi-database transaction. This description is available if the reconnect fails.

RDB$TRIGGER_MESSAGES

RDB$TRIGGER_MESSAGES defines a trigger message and associates the message with a particular trigger.

Table C-29: RDB$TRIGGER_MESSAGES

Column Name	Data Type	Length	Description
RDB$TRIGGER_NAME	CHAR	31	Names the trigger associated with this trigger message. The trigger name must exist in RDB$TRIGGERS.
RDB$MESSAGE_ NUMBER	SMALLINT		The message number of the trigger message being defined. The maximum number of messages is 32,767.
RDB$MESSAGE	VARCHAR	78	The source for the trigger message.

RDB$TRIGGERS

RDB$TRIGGERS defines triggers.

Table C-30: RDB$TRIGGERS

Column Name	Data Type	Length	Description
RDB$TRIGGER_NAME	CHAR	31	Names the trigger being defined.
RDB$RELATION_NAME	CHAR	31	Name of the table associated with the trigger being defined. This name must exist in RDB$RELATIONS.
RDB$TRIGGER_ SEQUENCE	SMALLINT		Sequence number for the trigger being defined. Determines when a trigger is executed in relation to others of the same type. Triggers with the same sequence number execute in random order. If this number is not assigned by the user, InterBase assigns a value of 0.
RDB$TRIGGER_TYPE	SMALLINT		The type of trigger being defined. Values: • 1 - BEFORE INSERT • 2 - AFTER INSERT • 3 - BEFORE UPDATE • 4 - AFTER UPDATE • 5 - BEFORE DELETE • 6 - AFTER DELETE
RDB$TRIGGER_SOURCE	BLOB	80	Subtype Text. Original source of the trigger definition. The **isql** SHOW TRIGGERS statement displays information from this column.

Table C-30: RDB$TRIGGERS (Continued)

Column Name	Data Type	Length	Description
RDB$TRIGGER_BLR	BLOB	80	Subtype BLR. BLR (Binary Language Representation) of the trigger source.
RDB$DESCRIPTION	BLOB	80	Subtype Text. User-written description of the trigger being defined. When including a comment in a CREATE TRIGGER or ALTER TRIGGER statement, **isql** writes to this column.
RDB$TRIGGER_ INACTIVE	SMALLINT		Indicates whether the trigger being defined is: • Active (value of 0) • Inactive (value of 1)
RDB$SYSTEM_FLAG	SMALLINT		Indicates whether the trigger is: • User-defined (value of 0) • System-defined (value greater than 0)
RDB$FLAGS	SMALLINT		

RDB$TYPES

RDB$TYPES records enumerated data types and alias names for InterBase character sets and collation orders. This capability is not available in the current release.

Table C-31: RDB$TYPES

Column Name	Data Type	Length	Description
RDB$FIELD_NAME	CHAR	31	Column for which the enumerated data type is being defined.
RDB$TYPE	SMALLINT		Identifies the internal number that represents the column specified above. Type codes (same as RDB$DEPENDENT_TYPES): • 0 - table • 1 - view • 2 - trigger • 3 - computed_field • 4 - validation • 5 - procedure All other values are reserved for future use.
RDB$TYPE_NAME	CHAR	31	Text that corresponds to the internal number.
RDB$DESCRIPTION	BLOB	80	Subtype Text. Contains a user-written description of the enumerated data type being defined.

Table C-31: RDB$TYPES (Continued)

Column Name	Data Type	Length	Description
RDB$SYSTEM_FLAG	SMALLINT		Indicates whether the data type is: • User-defined (value of 0) • System-defined (value greater than 0)

RDB$USER_PRIVILEGES

RDB$USER_PRIVILEGES keeps track of the privileges assigned to a user through an SQL GRANT statement. There is one occurrence of this table for each user / privilege intersection.

Table C-32: RDB$USER_PRIVILEGES

Column Name	Data Type	Length	Description
RDB$USER	CHAR	31	Names the user who was granted the privilege listed in the following column, RDB$PRIVILEGE.
RDB$GRANTOR	CHAR	31	Names the user who granted the privilege.
RDB$PRIVILEGE	CHAR	6	Identifies the privilege granted to the user listed in the RDB$USER column, above. Valid values are: • ALL • SELECT • DELETE • INSERT • UPDATE
RDB$GRANT_OPTION	SMALLINT		Indicates whether the privilege was granted with the WITH GRANT OPTION (value of 1) or not (value of 0). This option enables a user to grant the same authority to other users.
RDB$RELATION_NAME	CHAR	31	Identifies the table to which the privilege applies.
RDB$FIELD_NAME	CHAR	31	For update privileges, identifies the column to which the privilege applies.
RDB$USER_TYPE	SMALLINT		
RDB$OBJECT_TYPE	SMALLINT		

RDB$VIEW_RELATIONS

RDB$VIEW_RELATIONS is not used by SQL objects.

Table C-33: RDB$VIEW_RELATIONS

Column Name	Data Type	Length	Description
RDB$VIEW_NAME	CHAR	31	Name of a view. The combination of RDB$VIEW_NAME and RDB$VIEW_CONTEXT must be unique.
RDB$RELATION_NAME	CHAR	31	Name of a table referenced in the view definition.
RDB$VIEW_CONTEXT	SMALLINT		Alias used to qualify view columns. Must have the same value as the alias used in the view BLR (Binary Language Representation) for this query.
RDB$CONTEXT_NAME	CHAR	31	Textual version of the alias identified in RDB$VIEW_CONTEXT. This variable must: • Match the value of the RDB$VIEW_SOURCE column for the corresponding table in RDB$RELATIONS. • Be unique in the view.

System Views

The four views in this section provide information about existing integrity constraints for a database. They must be created after a database is created. SQL system views are a subset of system views defined in the SQL-92 standard. Since they are not defined by InterBase, the names of the system view and their columns do not start with RDB$.

- CHECK_CONSTRAINTS
- CONSTRAINTS_COLUMN_USAGE
- REFERENTIAL_CONSTRAINTS
- TABLE_CONSTRAINTS

CHECK_CONSTRAINTS

CHECK_CONSTRAINTS identifies all CHECK constraints defined in the database.

Table C-34: CHECK_CONSTRAINTS

Column Name	Data Type	Length	Description
CONSTRAINT_NAME	CHAR	31	Unique name for the CHECK constraint. Nullable.
CHECK_CLAUSE	BLOB	80	Subtype Text. Nullable. Original source of the trigger definition, stored in the RDB$TRIGGER_SOURCE COLUMN in RDB$TRIGGERS.

CONSTRAINTS_COLUMN_USAGE

CONSTRAINTS_COLUMN_USAGE identifies columns used by PRIMARY KEY and UNIQUE constraints. For FOREIGN KEY constraints, this view identifies the columns defining the constraint.

Table C-35: CONSTRAINTS_COLUMN_USAGE

Column Name	Data Type	Length	Description
TABLE_NAME	CHAR	31	Table for which the constraint is defined. Nullable.
COLUMN_NAME	CHAR	31	Column used in the constraint definition. Nullable.
CONSTRAINT_NAME	CHAR	31	Unique name for the constraint. Nullable.

REFERENTIAL_CONSTRAINTS

REFERENTIAL_CONSTRAINTS identifies all referential constraints defined in a database.

Table C-36: REFERENTIAL_CONSTRAINTS

Column Name	Data Type	Length	Description
CONSTRAINT_NAME	CHAR	31	Unique name for the constraint. Nullable.
UNIQUE_CONSTRAINT_NAME	CHAR	31	Name of the UNIQUE or PRIMARY KEY constraint corresponding to the specified referenced column list. Nullable.
MATCH_OPTION	CHAR	7	Reserved for future use. Always set to FULL. Nullable.

Table C-36: REFERENTIAL_CONSTRAINTS (Continued)

Column Name	Data Type	Length	Description
UPDATE_RULE	CHAR	11	Reserved for future use. Always set to RESTRICT. Nullable.
DELETE_RULE	CHAR	11	Reserved for future use. Always set to RESTRICT. Nullable.

TABLE_CONSTRAINTS

TABLE_CONSTRAINTS identifies all constraints defined in a database.

Table C-37: TABLE_CONSTRAINTS

Column Name	Data Type	Length	Description
CONSTRAINT_NAME	CHAR	31	Unique name for the constraint. Nullable.
TABLE_NAME	CHAR	31	Table for which the constraint is defined. Nullable.
CONSTRAINT_TYPE	CHAR	11	Possible values are UNIQUE, PRIMARY KEY, FOREIGN KEY, and CHECK. Nullable.
IS_DEFERRABLE	CHAR	3	Reserved for future use. Always set to No. Nullable.
INITIALLY_DEFERRED	CHAR	3	Reserved for future use. Always set to No. Nullable.

Appendix D

Character Sets and Collation Orders

CHAR, VARCHAR, and text BLOB columns in InterBase can use many different character sets. A *character set* defines the symbols that can be entered as text in a column, and its also defines the maximum number of bytes of storage necessary to represent each symbol. In some character sets, such as ISO8859_1, each symbol requires only a single byte of storage. In others, such as UNICODE_FSS, each symbol requires from 1 to 3 bytes of storage.

Each character set also has an implicit *collation order* that specifies how its symbols are sorted and ordered. Some character sets also support alternative collation orders. In all cases, choice of character set limits choice of collation orders.

This appendix lists available character sets and their corresponding collation orders.

This appendix also describes how to specify:

- Default character set for an entire database.
- Alternative character set for a particular column in a table.
- Client application character set that the server should use when translating data between itself and the client.
- Collation order for a column.
- Collation order for a value in a comparison operation.
- Collation order in an ORDER BY clause.
- Collation order in a GROUP BY clause.

InterBase Character Sets and Collation Orders

The following table lists each character set that can be used in InterBase. For each character set, the minimum and maximum number of bytes used to store each symbol is listed, and all collation orders supported for that character set are also listed. The first collation order for a given character set is that set's implicit collation, the one that is used if no COLLATE clause specifies an alternative order. The implicit collation order cannot be specified in the COLLATE clause.

Table D-1: Character Sets and Collation Orders

Character Set	Char. Set ID	Max. Char. Size	Min. Char. Size	Collation Orders
ASCII	2	1 byte	1 byte	ASCII
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
DOS850	11	1 byte	1 byte	DOS850 DB_DEU850 DB_ESP850 DB_FRA850 DB_FRC850 DB_ITA850 DB_NLD850 DB_PTB850 DB_SVE850 DB_UK850 DB_US850

Table D-1: Character Sets and Collation Orders (Continued)

Character Set	Char. Set ID	Max. Char. Size	Min. Char. Size	Collation Orders
DOS852	45	1 byte	1 byte	DOS852 DB_CSY DB_PLK DB_SLO PDOX_CSY PDOX_HUN PDOX_PLK PDOX_SLO
DOS857	46	1 byte	1 byte	DOS857 DB_TRK
DOS860	13	1 byte	1 byte	DOS860 DB_PTG860
DOS861	47	1 byte	1 byte	DOS861 PDOX_ISL
DOS863	14	1 byte	1 byte	DOS863 DB_FRC863
DOS865	12	1 byte	1 byte	DOS865 DB_DAN865 DB_NOR865 PDOX_NORDAN4
EUCJ_0208	6	2 bytes	1 byte	EUJC_0208
ISO8859_1	21	1 byte	1 byte	ISO8859_1 DA_DA DE_DE DU_NL EN_UK EN_US ES_ES FI_FI FR_CA FR_FR IS_IS IT_IT NO_NO PT_PT SV_SV
NEXT	56	1 byte	1 byte	NEXT NXT_DEU NXT_FRA NXT_ITA NXT_US

Table D-1: Character Sets and Collation Orders (Continued)

Character Set	Char. Set ID	Max. Char. Size	Min. Char. Size	Collation Orders
NONE	0	1 byte	1 byte	NONE
OCTETS	1	1 byte	1 byte	OCTETS
SJIS_0208	5	2 bytes	1 byte	SJIS_0208
UNICODE_FSS	3	3 bytes	1 byte	UNICODE_FSS
WIN1250	51	1 byte	1 byte	WIN1250 PXW_CSY PXW_HUNDC PXW_PLK PXW_SLO
WIN1251	52	1 byte	1 byte	WIN1251 PXW_CYRL
WIN1252	53	1 byte	1 byte	WIN1252 PXW_INTL PXW_INTL850 PXW_NORDAN4 PXW_SPAN PXW_SWEDFIN
WIN1253	54	1 byte	1 byte	WIN1253 PXW_GREEK
WIN1254	55	1 byte	1 byte	WIN1254 PXW_TURK

Character Set Storage Requirements

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

For character sets that require only a single byte of storage, the maximum number of symbols that can be stored in a single field corresponds to the number of bytes. For character sets that require up to three bytes per symbol, the maximum number of symbols that can be safely stored in a field is 1/3 of the maximum number of bytes for the data type. For example, for a CHAR column defines to use the UNICODE_FSS character set, the maximum number of characters that can be specified is 10,922 (32,767/3):

```
. . .
CHAR(10922) CHARACTER SET UNICODE_FSS,
. . .
```

Paradox and dBASE Character Sets and Collations

Many character sets and their corresponding collations are provided to support Borland Paradox for DOS, Paradox for Windows, dBASE for DOS, and dBASE for Windows.

Character Sets for DOS

The following character sets correspond to MS-DOS code pages, and should be used to specify character sets for InterBase databases that are accessed by Paradox for DOS and dBASE for DOS:

Table D-2: 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 WIN125 character set supports a Paradox for Windows collation order for Norwegian and Danish called “PXW_NORDAN4”.

For more information about Windows character sets and Paradox for Windows collation orders, see the appropriate Paradox for Windows documentation and driver books.

Additional Character Sets and Collations

Support for additional character sets and collation orders is constantly being added to InterBase. To see if additional character sets and collations are available for a newly created database, connect to the database with **isql**, then use the following set of queries to generate a list of available character sets and collations:

```
SELECT RDB$CHARACTER_SET_NAME, RDB$CHARACTER_SET_ID
   FROM RDB$CHARACTER_SETS
   ORDER BY RDB$CHARACTER_SET_NAME;

SELECT RDB$COLLATION_NAME, RDB$CHARACTER_SET_ID
   FROM RDB$COLLATIONS
   ORDER BY RDB$COLLATION_NAME;
```

Specifying a Default Character Set for a Database

A database’s default character set designation specifies the character set the server uses to tag CHAR, VARCHAR, and text BLOB columns in the database when no other character set information is provided. When data is stored in such columns without additional character set information, the server uses the tag to determine how to store and transliterate that data. A default character set should always be specified for a database when it is created with CREATE DATABASE.

To specify a default character set, use the DEFAULT CHARACTER SET clause of CREATE DATABASE. For example, the following statement creates a database that uses the ISO8859_1 character set:

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

Important If you do not specify a character set, the character set defaults to NONE. Using character set NONE means that there is no character set assumption for columns; data is stored and retrieved just as you originally entered it. You can load any character set into a column defined with NONE, but you cannot later move that data into another column that has been defined

with a different character set. In this case, no transliteration is performed between the source and destination character sets, and errors may occur during assignment.

For the complete syntax of CREATE DATABASE, see “CREATE DATABASE” in Chapter 2: “SQL Statement and Function Reference.”

Specifying a Character Set for a Column in a Table

Character sets for individual columns in a table can be specified as part of the column’s CHAR or VARCHAR 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.gdb" DEFAULT CHARACTER SET ISO8859_1;

CREATE TABLE RUS_NAME(
LNAME VARCHAR(30) NOT NULL CHARACTER SET CYRL,
FNAME VARCHAR(20) NOT NULL CHARACTER SET CYRL,
);
```

For the complete syntax of CREATE TABLE, see “CREATE TABLE” in Chapter 2: “SQL Statement and Function Reference.”

Specifying a Character Set for a Client Attachment

When a client application, such as **isql**, connects to a database, it may have its own character set requirements. The server providing database access to the client does not know about these requirements unless the client specifies them. The client application specifies its character set requirement using the SET NAMES statement *before* it connects to the database.

SET NAMES specifies the character set the server should use when translating data from the database to the client application. Similarly, when the client sends data to the database, the server translates the data from the client’s character set to the database’s default character set (or the character set for an individual column if it differs from the database’s default character set).

For example, the following **isql** command specifies that **isql** is using the DOS437 character set. The next command connects to the *europe* database created above, in “Specifying a Character Set for a Column in a Table”:

```
SET NAMES DOS437;
CONNECT "europe.gdb" USER "JAMES" PASSWORD "U4EEAH";
```

For the complete syntax of SET NAMES, see “SET NAMES” in Chapter 2: “SQL Statement and Function Reference.” For the complete syntax of CONNECT, see “CONNECT” in Chapter 2: “SQL Statement and Function Reference.”

Specifying Collation Order for a Column

When a CHAR or VARCHAR column is created for a table, either with CREATE TABLE or ALTER TABLE, the collation order for the column can be specified using the COLLATE clause. COLLATE is especially useful for character sets such as ISO8859_1 or DOS437 that support many different collation orders.

For example, the following **isql** ALTER TABLE statement adds a new column to a table, and specifies both a character set and a collation order:

```
ALTER TABLE "FR_CA_EMP"
   ADD ADDRESS VARCHAR(40) CHARACTER SET ISO8859_1 NOT NULL
      COLLATE FR_CA;
```

For the complete syntax of ALTER TABLE, see “ALTER TABLE” in Chapter 2: “SQL Statement and Function Reference.”

Specifying Collation Order in a Comparison Operation

When CHAR or VARCHAR values are compared in a WHERE clause, it can be necessary to specify a collation order for the comparisons if the values being compared use different collation orders.

To specify the collation order to use for a value during a comparison, include a COLLATE clause after the value. For example, in the following WHERE clause fragment from an embedded application, the value to the left of the comparison operator is forced to be compared using a specific collation:

```
WHERE LNAME COLLATE FR_CA = :lname_search;
```

For the complete syntax of the WHERE clause, see “SELECT” in Chapter 2: “SQL Statement and Function Reference.”

Specifying Collation Order in an ORDER BY Clause

When CHAR or VARCHAR columns are ordered in a SELECT statement, it can be necessary to specify a collation order for the ordering, especially if columns used for ordering use different collation orders.

To specify the collation order to use for ordering a column in the ORDER BY clause, include a COLLATE clause after the column name. For example, in the following ORDER BY clause, the collation order for two columns is specified:

```
. . .
ORDER BY LNAME COLLATE FR_CA, FNAME COLLATE FR_CA;
```

For the complete syntax of the ORDER BY clause, see "SELECT" in Chapter 2: "SQL Statement and Function Reference."

Specifying Collation Order in a GROUP BY Clause

When CHAR or VARCHAR columns are grouped in a SELECT statement, it can be necessary to specify a collation order for the grouping, especially if columns used for grouping use different collation orders.

To specify the collation order to use for grouping columns in the GROUP BY clause, include a COLLATE clause after the column name. For example, in the following GROUP BY clause, the collation order for two columns is specified:

```
. . .
GROUP BY LNAME COLLATE FR_CA, FNAME COLLATE FR_CA;
```

For the complete syntax of the GROUP BY clause, see "SELECT" in Chapter 2: "SQL Statement and Function Reference."