Product Documentation

InterBase 2017

Update 2

Language Reference Guide

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

March, 2018

TABLE OF CONTENTS

LANGUAGE REFERENCE GUIDE

SQL STATEMENT AND FUNCTION REFERENCE

Language Reference Guide

This reference guide covers the InterBase elements.
For a listing of functions provided in the InterBase UDF library, see Working with UDFs and Blob Filters.

Using the InterBase Language Reference

The InterBase Language Reference details the syntax and semantics of SQL and Dynamic SQL (DSQL) statements for embedded applications programming and for isql, the InterBase interactive SQL utility. It also describes additional language and syntax that is 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.

For a listing of functions provided in the InterBase UDF library, see the "Working with UDFs and Blob Filters" chapter in the Developer's Guide.

SQL Statement and Function Reference

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

- SQL variants and dialects
- Database object naming conventions
- Lists of SQL statements and functions
- A description of each InterBase data type
- An introduction to using SQLCODE to handle errors
- How to use statement and function definitions
- A reference entry for each SQL statement supported by InterBase

SQL Flavors

Although InterBase SQL follows the ISO/IEC 9075:1992 standard closely, there are small differences. Differences also exist among the three major flavors of InterBase SQL: embedded SQL, dynamic SQL (DSQL), and the procedure and trigger language.

Embedded SQL (ESQL)

The embedded form of SQL is used in programs written in traditional languages such as C and Pascal. A preprocessor turns SQL statements into host language data structures and calls to the InterBase server. The embedded language is written into the program; its statements cannot be generated dynamically. Statements in embedded SQL are terminated with a semicolon.

Dynamic SQL (DSQL)

DSQL allows a program to create statements at run time. It can be used from conventional languages through the InterBase API. More often, it is used from modern development environments such as Delphi, which hide the nuts and bolts of the API. A completed DSQL statement is very much like the "embedded" language, without the "EXEC SQL" and without the terminating semicolon.

Stored Procedure and Trigger Language

Triggers and stored procedures are written in a variant of the embedded language, extended to provide flow control, conditional expressions, and error handling. Certain constructs, including all DDL (Data Definition Language) statements, are omitted. Within a trigger or stored procedure, statements are separated by semicolons.

Interactive SQL (isql)

The interactive query language, ***isql,*** is very similar to DSQL, with some omissions (cursors, for example) and a few additions (***SET*** and ***SHOW*** statements). Like embedded SQL, ***isql*** statements must be terminated with a semicolon.

SQL Dialects

Starting with version 6, InterBase is closer to the ISO/IEC 9075:1992 standard than previous versions in several ways. Some of those ways are incompatible with earlier implementations of SQL. In the current InterBase, each client and database has a SQL dialect: an indicator that instructs an InterBase server how to interpret transition features: those features whose meanings have changed between InterBase versions. See the Migration appendix in the Operations Guide for information about using dialects and transition features.

Dialects

- Dialect 1: transition features are interpreted as in InterBase version 5.6 and earlier.
- Dialect 2: diagnostic mode, where transition features are recognized and flagged with a warning.
- Dialect 3: transition features are interpreted as SQL-92 compliant.

Transition Features

- Double quote (“): changed from a synonym for the single quote (’) to the delimiter for an object name.
- Large exact numerics: ***DECIMAL*** and ***NUMERIC*** data types with precision greater than 9 are stored at INT64 instead of ***DOUBLE PRECISION***.
- ***DATE***, ***TIME***, and ***TIMESTAMP*** data types:
- ***DATE*** has changed from a 64-bit quantity containing both date and time information to a 32-bit quantity containing only date information.
- ***TIME*** is a 32-bit quantity containing only time information.
- ***TIMESTAMP*** is a 64-bit quantity containing both date and time information (same as ***DATE*** in InterBase 5 and older).

Database Object Naming Conventions

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 67 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–all tables, for example–must be unique. In most cases, object names must also be unique within the database.

To use keywords, ASCII characters, case-sensitive strings, or spaces (except for trailing spaces) in an object name, enclose the name in double quotes. It is then a delimited identifier. Delimited identifiers must always be referenced in double quotes. In InterBase dialect 3, names enclosed in double quotes are case sensitive. For example:

```
SELECT "CodAR" FROM MyTable
```

is different from:

```
SELECT "CODAR" FROM MyTable
```

This behavior conforms to ANSI SQL semantics for delimited identifiers.

For more information about naming database objects with ***CREATE*** or ***DECLARE*** statements, see the Language Reference Guide.

Statement List

This chapter describes the following SQL statements:

A		
ALTER DATABASE	ALTER DOMAIN	ALTER EXCEPTION
ALTER INDEX	ALTER PROCEDURE	ALTER TABLE
ALTER TRIGGER	ALTER USER	
B		
BASED ON	BEGIN DECLARE SECTION	
C		
CASE	CLOSE	CLOSE (BLOB)
COALESCE( )	COMMIT	CONNECT
CREATE DATABASE	CREATE DOMAIN	CREATE ENCRYPTION
CREATE EXCEPTION	CREATE GENERATOR	CREATE INDEX
CREATE JOURNAL	CREATE JOURNAL ARCHIVE	CREATE PROCEDURE
CREATE ROLE	CREATE SHADOW	CREATE SUBSCRIPTION
CREATE TABLE	CREATE TRIGGER	CREATE USER
CREATE VIEW		
D		
DECLARE CURSOR	DECLARE CURSOR (BLOB)	DECLARE EXTERNAL FUNCTION
DECLARE FILTER	DECLARE STATEMENT	DECLARE TABLE
DELETE	DESCRIBE	DISCONNECT
DROP DATABASE	DROP DOMAIN	DROP ENCRYPTION
DROP EXCEPTION	DROP EXTERNAL FUNCTION	DROP FILTER
DROP GENERATOR	DROP INDEX	DROP JOURNAL
DROP JOURNAL ARCHIVE	DROP PROCEDURE	DROP ROLE
DROP SUBSCRIPTION*	DROP SHADOW	DROP TRIGGER
DROP VIEW	DROP USER	
E		
END DECLARE SECTION	EVENT INIT	EVENT WAIT
EXECUTE	EXECUTE IMMEDIATE	EXECUTE PROCEDURE
F		
FETCH	FETCH (BLOB)	
G		

GRANT	GRANT SUBSCRIBE	GRANT TEMPORARY SUBSCRIBE
I		
INSERT	INSERT CURSOR (BLOB)	
N		
NULLIF( )		
O		
OPEN	OPEN (BLOB)	
P		
PREPARE		
R		
RELEASE SAVEPOINT	REVOKE	ROLLBACK
S		
SAVEPOINT	SELECT	SET DATABASE
SET GENERATOR	SET NAMES (Reference)	SET SQL DIALECT
SET STATISTICS	SET SUBSCRIPTION	SET TRANSACTION
SHOW SQL DIALECT		
T		
Truncate Table		
U		
UPDATE		
W		
WHENEVER		

* For more information about creating subscriptions, see the Change View chapter in the Data Definition Guide.

Function List

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

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.
EXTRACT()	Conversion	Extracts date and time information from ***DATE***, ***TIME***, and ***TIMESTAMP*** values.
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 changing the scale or precision of numeric values, 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, a dynamically sizable data type called a Blob, and arrays of data types. It does not support arrays of Blobs. The following table lists the data types available to SQL statements in InterBase:

Data types supported by InterBase			
Name	**Size**	**Range/Precision**	**Description**
BLOB	Variable	• None • Blob segment size is limited to 64K. • Dynamically sizable data type for storing large data such as graphics, text, and digitized voice. • Basic structural unit is the segment. • Blob subtype describes Blob contents.	
BOOLEAN	16 bits	• TRUE • FALSE • UNKNOWN • Represents truth values TRUE, FALSE, and UNKNOWN. • Requires ODS 11 or higher, any dialect.	
CHAR(<n>)	<n> characters	• 1 to 32,767 bytes • Character set character size determines the maximum number of characters that can fit in 32K. • Fixed length CHAR or text string type • Alternate keyword: CHARACTER	
DATE	32 bits, signed[1]	1 Jan 100 a.d. to 29 Feb 32768 a.d.	ISC_DATE; stores a date as a 32-bit longword.
DECIMAL (<precision>, <scale>)	Variable (16, 32, or 64 bits)	• <precision> = 1 to 18; specifies at least <precision> digits of precision to store. • <scale> = 1 to 18; 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 • Example: DECIMAL(10, 3) holds numbers accurately in the following

Data types supported by InterBase			
Name	**Size**	**Range/Precision**	**Description**
			format: ppppppp.sss
DOUBLE PRECISION	64 bits[2]	2.225 x 10^{-308} to 1.797 x 10^{308}	IEEE double precision: 15 digits
FLOAT	32 bits	1.175 x 10–38 to 3.402 x 10^{38}	IEEE single precision: 7 digits
INTEGER	32 bits	–2,147,483,648 to 2,147,483,647	Signed long (longword)
NUMERIC - (<precision>, <scale>)	Variable (16, 32, or 64 bits)	• <precision> = 1 to 18; specifies exactly <precision> digits of precision to store. • <scale> = 1 to 18; specifies number of decimal places for storage. • Must be less than or equal to <precision>. o Number with a decimal point <scale> digits from the right o Example: NUMERIC(10,3) holds numbers accurately in the following format: ppppppp.[sss]	
SMALLINT	16 bits	–32,768 to 32,767	Signed short (word)
TIME	32 bits, unsigned	0:00 AM to 23:59.9999 PM	ISC_TIME
TIMESTAMP	64 bits	1 Jan 100 a.d. to 29 Feb 32768 a.d.	Also includes time information.
VARCHAR (<n>)	<n> characters	• 1 to 32,765 bytes • Character set character size determines the maximum number of characters that can fit in 32K. o Variable length CHAR or text string type o Alternate keywords: CHAR VARYING, CHARACTER VARYING	

1. InterBase version 5 had a ***DATE*** data type that was 64 bits long and included both the date and time. InterBase version 6 and later recognizes that type if you have specified dialect 1; in dialect 3, that type is called ***TIMESTAMP***.
2. Actual size of ***DOUBLE*** is platform-dependent. Most platforms support the 64-bit size.

Exact Numerics

All ***NUMERIC*** and ***DECIMAL*** data types are stored as exact numerics: 16, 32, or 64 bits, depending on the precision. ***NUMERIC*** and ***DECIMAL*** data types with precision greater than 9 are referred to as large exact numerics.

- If one operand is an approximate numeric, the result of any dyadic operation (addition, subtraction, multiplication, division) is ***DOUBLE PRECISION***.
- Any value that can be stored in a ***DECIMAL(18,S)*** can also be specified as the default value for a column or a domain.

Addition and Subtraction

If both operands are exact numeric, adding or subtracting the operands produces an exact numeric with a precision of 18 and a scale equal to the larger of the two. For example:

```
CREATE TABLE t1 (n1 NUMERIC(16,2), n2 NUMERIC(16,3));
INSERT INTO t1 VALUES (12.12, 123.123);
COMMIT;
```

The following query returns the integer 135.243. The largest scale of the two operands is 3; therefore, the scale of the sum is 3.

```
SELECT n1 + n2 FROM t1;
```

Similarly, the following query returns the integer -111.003:

```
SELECT n1 - n2 FROM t1;
```

If either of the operands is approximate numeric (***FLOAT***, ***REAL***, or ***DOUBLE PRECISION***), the result is ***DOUBLE PRECISION***.

Multiplication

If both operands are exact numeric, multiplying the operands produces an exact numeric with a precision of 18 and a scale equal to the sum of the scales of the operands. For example:

```
CREATE TABLE t1 (n1 NUMERIC(16,2), n2 NUMERIC(16,3));
INSERT INTO t1 VALUES (12.12, 123.123);
COMMIT;
```

The following query returns the integer 1492.25076 because n1 has a scale of 2 and n2 has a scale of 3. the sum of the scales is 5.

```
SELECT n1*n2 FROM t1
```

If one of the operands is approximate numeric (***FLOAT***, ***REAL***, or ***DOUBLE PRECISION***), the result is ***DOUBLE PRECISION***.

Division

If both operands are exact numeric, dividing the operands produces an exact numeric with a precision of 18 and a scale equal to the sum of the scales of the operands. If at least one operand of a division operator has an approximate numeric type (***FLOAT***, ***REAL***, or ***DOUBLE PRECISION***), the result is ***DOUBLE PRECISION***.

For example, in the following table, division operations produce a variety of results:

```
CREATE TABLE t1 (i1 INTEGER), i2 INTEGER, n1 NUMERIC(16,2)
n2 NUMERIC(16,2));
```

```
INSERT INTO t1 VALUES (1, 3, 1.00, 3.00);
COMMIT;
```

The following query returns the integer 0 because each operand has a scale of 0, so the sum of the scales is 0:

```
SELECT i1/i2 FROM t1
```

The following query returns the ***NUMERIC(18,2)*** value 0.33, because the sum of the scales 0 (operand 1) and 2 (operand 2) is 2:

```
SELECT i1/n2 FROM t1
```

The following query returns the ***NUMERIC(18,4)*** value 0.3333, because the sum of the two operand scales is 4:

```
SELECT n1/n2 FROM t1
```

In InterBase 5 and earlier, any of the above division operations would have returned the ***DOUBLE PRECISION*** value 0.3333333333333333.

Error Handling

Every time an executable SQL statement is executed, the SQLCODE variable is set to indicate its success or failure. No SQLCODE is 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:

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

When an error occurs in ***isql,*** InterBase displays an error message.

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 Embedded SQL Guide.

Statement and Function Reference

The following is the reference of SQL statements and functions available in InterBase.

Each statement and function definition includes the following elements:

Element	Description
Title	Statement name
Definition	The main purpose and availability of the statement
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 EXECSQL keywords. EXECSQL 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

Changes the characteristics of the current database. Available in ***gpre,*** DSQL, and ***isql,*** but not in the trigger or stored procedure language.

```
ALTER {DATABASE | SCHEMA}

   {ADD <add_clause> | DROP <drop_clause> | ENCRYPT <key_name> | DECRYPT <key_name> |
SET <set_clause>};

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

fileinfo = LENGTH [=] INT [PAGE[S]]
| STARTING [AT [PAGE]] INT [fileinfo]

<drop_clause> = ADMIN OPTION

<key_name> = ENCRYPT <|> DECRYPT

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

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
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.
ADD ADMIN OPTION	Enables embedded user authentication.
DROP ADMIN OPTION	Disables embedded user authentication.
ENCRYPT <key_name>	Uses the named encryption key to encrypt the database. Encrypting a database causes all pages to be encrypted. Only the database owner can encrypt a database.
DECRYPT <key_name>	Uses the named encryption key to decrypt the database. Decrypting a database causes all pages to be decrypted and rewritten in plaintext. Only the database owner can decrypt a database.
SET FLUSH INTERVAL<number>	Enables database flush. The interval <number> is interpreted in units of seconds.
SET NO FLUSH INTERVAL	Disables database flush.
SET GROUP COMMIT	Allows transactions to be committed by a background cache writer thread.
SET NO GROUP COMMIT	Disables group commit.
SET LINGER INTERVAL	Allows a database to remain in memory after the last user detaches. Interval is seconds.
SET NO LINGER INTERBAL	Disables database linger.
SET RECLAIM INTERVAL	Reclaims the interval is in seconds. Determines how often the garbage collector thread will run to release memory from unused procedures, triggers, and internal system queries back to InterBase memory heap.
SET NO RECLAIM INTERVAL	Disables memory reclamation.
SET SYSTEM ENCRYPTION PASSWORD	Necessary to create encryption keys and perform encryption. InterBase uses a System Encryption Password (SEP) to protect the encryption keys that are used to encrypt the database and/or database columns. For more information about using InterBase encryption, see "Encrypting Your Data" in the Data Definition Guide. **Note:** Only the SYSDSO (Data Security Owner) can create this password.
SET NO SYSTEM ENCRYPTION PASSWORD	Deletes the password if there are no existing encryption keys. **Note:** Only SYSDSO can delete a password.
SET PAGE CACHE	Sets database page buffer cache limit. Also, tries to expand cache to that limit.
SET PASSWORD DIGEST '<digest_name>'	Sets the password hash function. The default value is 'DES-CRYPT'. See Implementing Stronger Password Protection for more information.

Description: *ALTER DATABASE* adds secondary files to an existing database. Secondary files permit databases to spread across storage devices, but they 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.

InterBase dynamically expands the last file in a database as needed. The maximum size of the last file is system-dependent. You should be aware that specifying a ***LENGTH*** for such files has no effect.

You cannot use ***ALTER DATABASE*** to split an existing database file. For example, if your existing database is 80,000 pages long and you add a secondary file ***STARTING AT*** 50000, InterBase starts the new database file at page 80,001.

TIP

To split an existing database file into smaller files, back it up and restore it. When you restore a database, you are free to specify secondary file sizes at will, without reference to the number and size of the original files.

Example: The following isql statement adds two secondary files to an existing database. The command creates a secondary database file called ***employee2.ib*** that is 10,000 pages long and another called ***employee3.ib.*** InterBase starts using ***employee2.ib*** only when the primary file reaches 10,000 pages.

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

ALTER DOMAIN

Changes a domain definition. Available in ***gpre,*** DSQL, and ***isql,*** but not in the stored procedure or trigger language.

```
ALTER DOMAIN { name |
old_name TO new_name }
SET DEFAULT {literal | NULL | USER}
| DROP DEFAULT
| ADD [CONSTRAINT] CHECK (dom_search_condition)
| DROP CONSTRAINT
| new_col_name
| TYPE data_type;
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 = {= | < | > | <= | >= | !< | !> | <> | !=}
```

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>	Represents the 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; the olumn must be of compatible text type to use the default. • Defaults set at the column level overrides defaults set at the domain level.
DROP DEFAULT	Drops an existing default.
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 the CHECK constraint from the domain definition.
<new_col_name>	Changes the domain name.
TYPE <data_type>	Changes the domain data type.

Description: ***ALTER DOMAIN*** changes any aspect of an existing domain except its ***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 the ***NOT NULL*** setting of a domain, drop the domain and recreate it with the desired combination of features.

The TYPE clause of ***ALTER DOMAIN*** does not allow you to make data type conversions that could lead to data loss.

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;
```

ALTER EXCEPTION

Changes the message associated with an existing exception. Available in DSQL and isql, but not in the embedded language or stored procedure and trigger language.

```
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.';
```

ALTER INDEX

Activates or deactivates an index. Available in embedded SQL, DSQL, and ***isql,*** but not in the stored procedure or trigger language.

```
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 an index is exactly like dropping it, except that the index definition remains in the database. Activating an index creates a new index structure.

Before inserting, updating, or deleting a large number of rows, deactivate a table's indexes to avoid altering the index incrementally. When finished, reactivate the index. A reasonable metric is that if you intend to add or delete more than 15% of the rows in a table, or update an indexed column in more than 10% of the rows, you should consider deactivating and reactivating the 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;
```

ALTER PROCEDURE

Changes the definition of an existing stored procedure. Available in DSQL and isql but not in the embedded language or in the stored procedures or triggers.

```
ALTER PROCEDURE '' ''<name> [(<param>'' ''data_type [,'' ''<param>'' ''data_type ...])]
[RETURNS (<param>'' ''data_type [, <param> data_type ...])]
AS ''procedure_body'' ;
```

Argument	Description
<name>	Name of an existing procedure.
<param data_type>	Input parameters used by the procedure; legal data types are listed under ***CREATE PROCEDURE***.
RETURNS param data_type	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.

Description: ***ALTER PROCEDURE*** changes an existing stored procedure without affecting its dependencies. It can modify the input parameters, output parameters, and body of a procedure.

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***.

IMPORTANT

Be careful about changing the type, number, and order of input and output parameters to a procedure, because existing application code may assume the procedure has its original format. Check for dependencies between procedures before changing parameters. Should you change parameters and find that another procedure can neither be altered to accept the new parameters or deleted, change the original procedure back to its original parameters, fix the calling procedure, then change the called procedure.

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)***:

```
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;
```

ALTER TABLE

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

IMPORTANT

To alter a global temporary table, see: "Altering a global temporary table" in the Data Definition Guide.

```
 ALTER TABLE <table> operation [, operation ...];
operation = ADD col_def
| ADD tconstraint
| ALTER [COLUMN] column_name alt_col_clause
| DROP col
| DROP CONSTRAINT constraint
| [ON COMMIT {PRESERVE | DELETE} ROWS [RESTRICT
| CASCADE]] | [SET [NO] RESERVE SPACE]
alt_col_clause = TO new_col_name
| TYPE new_col_data_type
| POSITION new_col_position
col_def = col {data_type | COMPUTED [BY] (expr) | domain}
[DEFAULT {literal | NULL | USER}]
[NOT NULL]
[col_constraint]
[COLLATE collation]
data_type =
{SMALLINT | INTEGER | FLOAT | DOUBLE PRECISION}[array_dim]
| {DATE | TIME | TIMESTAMP} [array_dim]
| {DECIMAL | NUMERIC} [(precision [, scale])] [array_dim]
| {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR} [(int)]
[array_dim] [CHARACTER SET charname]
| {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
[VARYING] [(int)] [array_dim]
| BLOB [SUB_TYPE {int | subtype_name}] [SEGMENT SIZE int]
[CHARACTER SET charname]
| BLOB [(seglen [, subtype])]array_dim = [[x:]y [, [x:]y ...]]
| BOOLEAN
expr = A valid SQL expression that results in a single value.
```

```
col_constraint = [CONSTRAINT constraint]
{ UNIQUE
| PRIMARY KEY
| REFERENCES other_table [(other_col [, other_col ...])]
[ON DELETE {RESTRICT|NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
[ON UPDATE {RESTRICT|NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
| CHECK (search_condition)}
tconstraint = [CONSTRAINT constraint]
{{PRIMARY KEY | UNIQUE} (col [, col ...])
| FOREIGN KEY (col [, col ...])
REFERENCES other_table [(other_col [, other_col ...])]
[ON DELETE {RESTRICT|NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
[ON UPDATE {RESTRICT|NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
| CHECK (search_condition)}
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 {>= | <=}
| 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]
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 data_type)
| UPPER (val)
| GEN_ID (generator, val)
operator = {= | < | > | <= | >= | !< | !> | <> | !=}
select_one = SELECT on a single column; returns exactly one value.
select_list = SELECT on a single column; returns zero or more values.
select_expr = SELECT on a list of values; returns zero or more values.
```

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.

Notes on ***ALTER TABLE*** syntax:

- The column constraints for referential integrity were new in InterBase 5.
- You cannot specify a COLLATE clause for Blob columns.
- When declaring arrays, you must include the outermost brackets, shown below in bold. For example, the following statement creates a 5 by 5 two-dimensional array of strings, each of which is 6 characters long:

```
my_array = varchar(6)[5,5]
```

- Use the colon (:) to specify an array with a starting point other than 1. The following example creates an array of integers that begins at 20 and ends at 30:

```
my_array = integer[20:30]
```

- 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
<col_def>	Description of a new column to add. • Must include a column name and <data_type>. • Can also include default values, column constraints, and a specific collation order.
<col>	Name of the column to add or drop; column name must be unique within the table.
<data_type>	Data type of the column; see Data Types.
ALTER [COLUMN]	Modifies column names, data types, and positions. Can also be used with ***ENCRYPT*** and ***DECRYPT*** options to encrypt and decrypt a column. For more information about encrypting databases and columns, see "Encrypting Your Data" in the Data Definition Guide.
COMPUTED [BY]<expr>	Specifies that the value of the column's data is calculated from expr at runtime and is therefore not allocated storage space in the database. • <expr> can be any arithmetic expression valid for the data types in the expression. • Any columns referenced in <expr> must exist before they can be used in <expr>. • <expr> cannot reference Blob columns. • <expr> must return a single value, and cannot return an array.
<domain>	Name of an existing domain.
DEFAULT	Specifies a default value for column data; this value is entered when no other entry is made; possible values are: • <literal>: Inserts a specified string, numeric value, or date value. • ***NULL***: Enters a ***NULL*** value; this is the default ***DEFAULT***.

Argument	Description
	• ***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 domain level.
CONSTRAINT <constraint>	Name of a column or table constraint; the constraint name must be unique within the table.
<constraint_def>	Specifies the kind of column constraint; valid options are ***UNIQUE***, ***PRIMARY KEY***, ***CHECK***, and ***REFERENCES***.
CHECK <search_condition>	An attempt to enter a new value in the column fails if the value does not meet the <search_condition>.
REFERENCES	Specifies that the column values are derived from column values in another table; if you do not specify column names, InterBase looks for a column with the same name as the referencing column in the referenced table.
ON DELETE|ON UPDATE	Used with ***REFERENCES***: Changes a foreign key whenever the referenced primary key changes; valid options are: • [Default] ***NO ACTION***: Does not change the foreign key; may cause the primary key update to fail due to referential integrity checks. • ***CASCADE***: For ***ON DELETE***, deletes the corresponding foreign key; for ***ON UPDATE***, updates the corresponding foreign key to the new value of the primary key. • ***SET NULL***: Sets all the columns of the corresponding foreign key to ***NULL***. • ***SET DEFAULT***: Sets every column of the corresponding foreign key to its default value in effect when the referential integrity constraint is defined; when the default for a foreign column change after the referential integrity constraint is defined, the change does not have an effect on the default value used in the referential integrity constraint.
NOTNULL	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 a column attribute only.
DROP CONSTRAINT	Drops the specified table constraint.
<table_constraint>	Description of the new table constraint; constraints can be ***PRIMARY KEY***, ***UNIQUE***, ***FOREIGN KEY***, or ***CHECK***.
COLLATE <collation>	Establishes a default sorting behavior for the column; see Character Sets and Collation Orders for more information.

Description: ***ALTER TABLE*** modifies the structure of an existing table. A single ***ALTER TABLE*** statement can perform multiple adds and drops.

- A table can be altered by its creator, the SYSDBA user, and any users with operating system superuser privileges.
- ***ALTER TABLE*** fails if the new data in a table violates a ***PRIMARY KEY*** or ***UNIQUE*** constraint definition added to the table. Dropping or altering a column fails if any of the following are true:
- The column is part of a ***UNIQUE***, ***PRIMARY***, or ***FOREIGN KEY*** constraint.
- The column is used in a ***CHECK*** constraint.
- The column is used in the <value> expression of a computed column.
- The column is referenced by another database object such as a view.

IMPORTANT

When a column is dropped, all data stored in it is lost.

Constraints:

- Referential integrity constraints include optional ***ON UPDATE*** and ***ON DELETE*** clauses. They define the change to be made to the referencing column when the referenced column is updated or deleted.
- To delete a column referenced by a computed column, you must drop the computed column before dropping the referenced column. To drop a column referenced in a ***FOREIGN KEY*** constraint, you must drop the constraint before dropping the referenced column. To drop a ***PRIMARY KEY*** or ***UNIQUE*** constraint on a column that is referenced by ***FOREIGN KEY*** constraints, drop the ***FOREIGN KEY*** constraint before dropping the ***PRIMARY KEY*** or ***UNIQUE*** key it references.
- You can create a ***FOREIGN KEY*** reference to a table that is owned by someone else only if that owner has explicitly granted you the ***REFERENCES*** privilege on that table using ***GRANT***. Any user who updates your foreign key table must have ***REFERENCES*** or ***SELECT*** privileges on the referenced primary key table.
- You can add a check constraint to a column that is based on a domain but be aware that changes to tables that contain ***CHECK*** constraints with subqueries may cause constraint violations.
- Naming column constraints is optional. If you do not specify a name, InterBase assigns a system-generated name. Assigning a descriptive name can make a constraint easier to find for changing or dropping, and more descriptive when its name appears in a constraint violation error message.
- When creating new columns in tables with data, do not use the ***UNIQUE*** constraint. If you use the ***NOT NULL*** constraint on a table with data, you should also specify a default value.

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

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

This statement results in the loss of all data in the dropped CURRENCY column.

The next ***isql*** statement changes the name of the LARGEST_CITY column to BIGGEST_CITY:

```
ALTER TABLE COUNTRY ALTER LARGEST_CITY TO BIGGEST_CITY;
```

NO RESERVE SPACE for Database and User Tables

This feature is useful if you have very, large databases (VLDB) with tables that are archival in nature. An archival table means that the rows of a table are infrequently or never UPDATED or DELETED; have complex queries, such as aggregates and analytics that process a high percentage of rows; and where indexes are rebuilt and the database is backed and/or restored frequently. These database operations could see a performance improve of 20% or more with a savings in storage space.

By default, InterBase reserves a small amount of space in each data page of a table to optimize UPDATE and DELETE operations on resident rows. This reserve space can amount to 20% or more of the total space occupied by all of the rows of the table. Some tables archive historical data or data that are UPDATED infrequently or not at all and their rows may never be deleted. Database operations that process most

or all of the rows, such as backup, restore, index creation, aggregate computation have always suffered performance penalties proportional to this reservation overhead.

For this reason, a CREATE/ALTER TABLE clause is introduced that prevents space reservation and maximizes row packing for the most efficient fill ratio. At the database level, it has been possible to restore a database with the -USE_ALL_SPACE switch so that no space is reserved for any table. To change the storage behavior in a like manner for new or existing databases, the same clause is introduced for CREATE/ALTER DATABASE.

User Interface

To effect the new storage behavior, a non-standard SQL clause is added:

Clause is presented before the secondary file specification.

Clause is presented in any order with other SET elements.

```
ALTER DATABASE ... SET [NO] RESERVE SPACE
```

Clause is presented in any order with other ADD, DROP, ALTER elements.

```
ALTER TABLE <TABLE name> ... SET [NO] RESERVE SPACE
```

This causes newly INSERTED rows to not reserve space on their data page for a DELETE record version stub, as would normally be the case. Over many row insertions, a decrease in storage size should be observed relative to what the table size would be in the absence of this feature. The optional NO keyword when used with ALTER TABLE toggles the behavior to the alternate state of the current storage behavior for the table.

The NO RESERVE storage modifier is preserved across database backup and restore. This state is stored as flag bit 64 (0x100) of RDB$RELATIONS.RDB$FLAGS for the user's table entry in the system table RDB $RELATIONS.

The clause is displayed by ISQL's SHOW TABLE command following the enumeration of a table's column definitions. It is also visible using ISQL's Extract (-x) command in a syntax-correct manner for the CREATE TABLE output of the respective table listing. The state for database-wide storage behavior is stored in a like manner for the RDB$DATABASE entry in RDB$RELATIONS.

ON COMMIT

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

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

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

For example, assume that TT1 is a temporary table with ON COMMIT PRESERVE and has a foreign reference to temporary table TT2 which is also ON COMMIT PRESERVE. If an attempt is made to modify TT2 to ON

COMMIT DELETE, an error is raised because an ON COMMIT PRESERVE table is not allowed by the SQL standard to have a referential constraint on an ON COMMIT DELETE table. RESTRICT returns this error while CASCADE would also alter TT1 to have ON COMMIT DELETE. Thus, CASCADE implements transitive closure when ON COMMIT behavior is modified.

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

ALTER TRIGGER

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

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

Argument	Description
<name>	Name of an existing trigger.
ACTIVE	[Default] Specifies that a trigger action takes effect when fired.
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 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.

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 statement modifies the trigger, SET_CUST_NO, to be inactive:

```
ALTER TRIGGER SET_CUST_NO INACTIVE;
```

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

```
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 ;
```

ALTER USER

Change an existing user. Available in DSQL and ***isql.***

```
ALTER USER <name> SET
[PASSWORD <password>]
[[NO] DEFAULT ROLE <name>]
[[NO] SYSTEM USER NAME <name>]
[[NO] GROUP NAME <name>]
[[NO] UID <number>]
[[NO] GID <number>]
[[NO] DESCRIPTION <string>]
[[NO] FIRST NAME <string>]
[[NO] MIDDLE NAME <string>]
[[NO] LAST NAME <string>]
[ACTIVE]
[INACTIVE];
```

Argument	Description
PASSWORD	Password of user.
[NO]DEFAULT ROLE	Default role.
[NO] SYSTEM USER NAME	System user name for target user.
[NO]GROUP NAME	Group name for target user.
[NO] UID	Target user ID.
[NO] GID	Group ID for target user.
[NO] DESCRIPTION	Description
[NO]FIRST NAME	First name for target user.
[NO] MIDDLE NAME	Middle name for target user.

Argument	Description
[NO]LAST NAME	Last name for target user.
ACTIVE	Default. After inactive, reinstates selected user.
INACTIVE	Prevents a user from logging into database.

Description: Alter user changes the definition of an existing user. Only used with database under embedded user authentication.

If you choose to set more than one property value for the user, include a comma between each property-value pair.

NOTE

When ***NO*** is specified, an argument to the option must not be supplied. No sets the option to a ***NULL*** state.

Examples: The following statement modifies the user, JDOE, to be inactive:

```
ALTER USER JDOE SET INACTIVE;
```

The next statement modifies the user, JDOE, to be active:

```
ALTER USER JDOE SET ACTIVE;
```

AVG( )

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

```
AVG ([ALL] VALUE | DISTINCT VALUE)
```

Argument	Description
ALL	Returns the average of all values.
DISTINCT	Eliminates duplicate values before calculating the average.
<value>	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;
```

BASED ON

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

```
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 variable declaration section of a program.

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 size of the variable 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;
```

BEGIN DECLARE SECTION

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

```
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;
```

CASE

The ***CASE*** function allows you to evaluate a column value on a row against multiple criteria, where each criterion might return a different value.

```
CASE <expression>
WHEN <expression> THEN <expression> | NULL
[ELSE <expression> | NULL]
[COALESCE <expression>]
[NULLIF <expression, expression, ...>]
END
```

Description: The ***CASE*** expression is a conditional value expression that consists of a list of value expressions, each of which is associated with a conditional expression. A ***CASE*** expression evaluates to the first value expression in the list for which its associated conditional expression evaluates to ***TRUE***. The ***CASE*** expression has simple and searched forms of syntax.

The ***COALESCE*** and ***NULLIF*** expressions are common, shorthand forms of use for the ***CASE*** expression involving the ***NULL*** state. A ***COALESCE*** expression consists of a list of value expressions. It evaluates to the first value expression in the list that evaluates to ***non-NULL.*** If none of the value expressions in the list evaluates to ***non-NULL,*** then the ***COALESCE*** expression evaluates to ***NULL***.

The ***NULLIF*** expression consists of a list of two value expressions. If the two expressions are unequal then the ***NULLIF*** expression evaluates to the first value expression in the list. Otherwise, it evaluates to ***NULL***.

Example: The following example demonstrates the use of ***CASE*** using the sample employee.ib database:

```
SELECT emp.first_name || ' ' || emp.last_name AS NAME,
CASE proj.proj_name
   WHEN 'DigiPizza' THEN 'Digital Pizza'
   WHEN 'AutoMap' THEN 'AutoMobile Map'
   WHEN 'Translator upgrade' THEN 'Universal Language Translator'
   ELSE 'Other'
END
AS project
FROM employee emp
   INNER JOIN employee_project emp_proj
      ON emp.emp_no = emp_proj.emp_no
   INNER JOIN project proj
      ON emp_proj.proj_id = proj.proj_id
```

CAST( )

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

```
CAST (VALUE AS <data_type>)
```

Argument	Description
<*val*>	A column, constant, or expression; in SQL, <val> can also be a host-language variable, function, or UDF.
<*data_type*>	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:

From data type class	To data type class
Numeric	character, varying character, numeric
Character, varying character	numeric, date, time, timestamp
Date	character, varying character, timestamp
Time	character, varying character, timestamp
Timestamp	character, varying character, date, time
Blob, arrays	—
Boolean	character, varying character

An error results if a given data type cannot be converted into the data type specified in CAST(). For example, you will get a string conversion error if you attempt to cast from a numeric type which is unable to represent in a date type to a date (e.g. a numeric type attempting to represent "year 99/12/31"(December) or "year 32768/3/1"(March)).

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);
```

To cast a ***VARCHAR*** data type, you must specify the length of the string, for example:

```
UPDATE client SET charef = CAST (clientref AS VARCHAR(20));
```

CLOSE

Closes an open cursor. Available in ***gpre.***

```
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 ***DECLARECURSOR***; the resulting rows become the active set of the cursor.
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;
```

CLOSE (BLOB)

Terminates a specified Blob cursor and releases associated system resources. Available in ***gpre.***

```
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 be closed only 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;
```

COALESCE( )

The ***COALESCE*** function evaluates to the first value expression in a list that evaluates to non-***NULL***. If none of the value expressions in the list evaluates to non-***NULL***, then the ***COALESCE*** expression evaluates to ***NULL***.

```
COALESCE(<expression1>,<expression2>,...<expression_n>)
```

Description: The ***COALESCE*** and ***NULLIF*** expressions are common, shorthand forms of use for the ***CASE*** expression involving the ***NULL*** state. A ***COALESCE*** expression consists of a list of value expressions. It evaluates to the first value expression in the list that evaluates to non-***NULL***. If none of the value expressions in the list evaluates to non-***NULL***, then the ***COALESCE*** expression evaluates to ***NULL***.

Example: The following example demonstrates the use of ***CASE*** using the sample employee.ib database:

```
select coalesce(department, head_dept, location) from department
```

COMMIT

Makes changes of a transaction to the database permanent, and ends the transaction. Available in gpre, DSQL, and ***isql.***

```
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 changes of transaction 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;
```

CONNECT

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

isql:

```
CONNECT 'filespec' [USER 'username'][PASSWORD 'password']
[CACHE INT] [ROLE 'rolename']
```

SQL:

```
CONNECT [TO] {ALL | DEFAULT} <config_opts>
| <db_specs> <config_opts> [, <db_specs> <config_opts>...];
<db_specs> = dbhandle
| {'filespec' | :variable} AS dbhandle
<config_opts> = [USER {'username' | :variable}]
[PASSWORD {'password' | :variable}]
[ROLE {'rolename' | :variable}]
[CACHE INT [BUFFERS]]
```

Argument	Description
{ALL|DEFAULT}	Connects to all databases specified with ***SET DATABASE***; options specified with ***CONNECT TO ALL*** affect all databases.
<'filespec>'	Database file name; can include path specification and node. The filespec must be in quotes if it includes spaces.
<dbhandle>	Database handle declared in a previous ***SET DATABASE*** statement; available in embedded SQL but not in ***isql***.
<:variable>	Host-language variable specifying a database, user name, or password; available in embedded SQL but not in ***isql***.
AS<dbhandle>	Attaches to a database and assigns a previously-declared handle to it; available in embedded SQL but not in ***isql***.
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.
ROLE{'<rolename>' | :<variable>}	String or host-language variable, up to 67 characters in size, which specifies the role that the user adopts on connection to the database. The user must have previously been granted membership in the role to gain the privileges of that role. Regardless of role memberships granted, the user has the privileges of a role at connect time only if a ROLE clause is specified in the connection. The user can adopt at most one role per connection, and cannot switch roles except by reconnecting.
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: 256 • Maximum value: system-dependent Do not use the <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). An error message occurs if InterBase cannot locate the database.
- Optionally specifies one or more of a user name, password, or role for use when attaching to the database. PC clients must always send a valid user name and password. InterBase recognizes only the first 8 characters of a password.

If an InterBase user has ***ISC_USER*** and ***ISC_PASSWORD*** environment variables set and the user defined by those variables is not in the InterBase security database (admin.ib by default), the user receives the following error when attempting to view users from the local server manager connection: "undefined user name and password." This applies only to the local connection; the automatic connection made through Server Manager bypasses user security.

- 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.ib' USER 'ACCT_REC' PASSWORD 'peanuts';
```

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

```
EXEC SQL
SET DATABASE DB1 = 'employee.ib';
EXEC SQL
CONNECT :db_file AS DB1;
```

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

```
EXEC SQL
CONNECT 'accounts.ib' 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.ib, with 80 buffers and database employee2.ib with the default of 75 buffers:

```
EXEC SQL
CONNECT 'employee.ib' CACHE 80, 'employee2.ib';
```

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 v, 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';
```

COUNT( )

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

```
COUNT ( * | [ALL] VALUE | DISTINCT VALUE)
```

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 the search condition of a query. It can be used in views and joins, as well as in 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;
```

CREATE DATABASE

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

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

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
'<filespec>'	• A new database file specification. • File naming conventions are platform-specific.
USER *'<username>'*	• Checks the <username> 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 when attaching to a server.

Argument	Description
PASSWORD '<password>'	• Checks the <password> against valid user name and password combinations in the security database on the server where the database will reside; can be up to 8 characters. • Windows client applications must provide a password when attaching to a server.
PAGE_SIZE [=] <int>	• Size, in bytes, for database pages. • int can be 1024 (default), 2048, 4096, 8129, or 16384.
PREALLOCATE [=] <number> [PAGE[S]]	• Reserves storage space in a file system for the requested number of database pages. It guarantees that a write will not fail due to lack of storage space over this range of pages.
WITH ADMIN OPTION	• Create new database with embedded user authentication enabled.
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 new, empty 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 might 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 dialect of the database.

The initial dialect of the database is the dialect of the client that creates it. For example, if you are using ***isql***, either start it with the ***-sql_dialect*** <n> switch or issue the ***SETSQL DIALECT <n>*** command before issuing the ***CREATE DATABASE*** command. Typically, you would create all databases in dialect 3. Dialect 1 exists to ease the migration of legacy databases.

To change the dialect of a database, use ***gfix*** or the Properties dialog in IBConsole. See the Migration appendix in the InterBase Operations Guide for information about migrating databases.

- The character set used by the database.

For a list of the character sets recognized by InterBase, see 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, you define its tables, views, indexes, and system views as well as any triggers, generators, stored procedures, and UDFs that you need.

IMPORTANT

In DSQL, you must execute ***CREATE DATABASE EXECUTE IMMEDIATE***. The database handle and transaction name, if present, must be initialized to zero prior to use.

Read-only databases :

Databases are always created in read-write mode. You can change a table to read-only mode in one of two ways: you can specify mode ***-read_only*** when you restore a backup, or you can use ***gfix -mode read_only*** to change the mode of a table to read-only. See “Database User Management” in the Operations Guide for more information on database configuration and maintenance.

About file sizes:

InterBase dynamically expands the last file in a database as needed. The maximum file size is system-dependent. This applies to single-file databases as well as to the last file of multifile databases. You should be aware that specifying a ***LENGTH*** for such files has no effect.

The total file size is the product of the number of database pages times the page size. The default page size is 4KB and the maximum page size is 16KB. However, InterBase files are small at creation time and increase in size as needed. The product of number of pages times page size represents a potential maximum size, not the size at creation.

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

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

The next embedded SQL statement creates a database with a page size of 2048 bytes rather than the default of 4096:

```
EXEC SQL
CREATE DATABASE 'employee.ib' 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.ib'
DEFAULT CHARACTER SET ISO8859_1
FILE 'employee2.ib' STARTING AT PAGE 10001;
```

CREATE DOMAIN

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

```
CREATE DOMAIN <domain> [AS] data_type
[DEFAULT {<literal> | NULL | USER}]
[NOT NULL] [CHECK (dom_search_condition)]
[COLLATE <collation>];
data_type> =
{SMALLINT|INTEGER|FLOAT|DOUBLE PRECISION} [array_dim]
| {DATE|TIME|TIMESTAMP} [array_dim]
| {DECIMAL | NUMERIC} [(precision [, scale])] [array_dim]
| {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR} [(int)]
[array_dim] [CHARACTER SET charname]
| {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
[VARYING] [(int)] [array_dim]
| BLOB [SUB_TYPE {int | subtype_name}] [SEGMENT SIZE int]
[CHARACTER SET charname]
| BLOB [(seglen [, subtype])]
| BOOLEAN
array_dim> = [[x:]y [, [x:]y ...]]
dom_search_condition> =
VALUE operator value
| VALUE [NOT] BETWEEN value AND value
| VALUE [NOT] LIKE value [ESCAPE value]
| VALUE [NOT] IN (value [, value ...])
| VALUE IS [NOT] NULL
| VALUE [NOT] CONTAINING value
| VALUE [NOT] STARTING [WITH] value
| (dom_search_condition)
| NOT dom_search_condition
| dom_search_condition OR dom_search_condition
```

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

Note on the ***CREATE DOMAIN*** syntax:

- ***COLLATE*** is useful only for text data, not for numeric types. Also, you cannot specify a ***COLLATE*** clause for Blob columns.
- When declaring arrays, you must include the outermost brackets, shown below in bold. For example, the following statement creates a 5 by 5 two-dimensional array of strings, each of which is six characters long:

```
my_array = varchar(6)[5,5]
```

- Use the colon (:) to specify an array with a starting point other than 1. The following example creates an array of ***INTEGER*** values that begins at 20 and ends at 30:

```
my_array = integer[20:30]
```

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
<domain>	Unique name for the domain.
<data_type>	SQL data type
DEFAULT	Specifies a default column value that is entered when no other entry is made; possible values are: <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.
NOTNULL	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 not 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*** or ***VARCHAR*** 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 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 ***NCHAR*** 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 Character Sets and Collation Orders.

Columns based on a domain definition inherit all characteristics of the domain. The domain default, collation clause, and ***NOTNULL*** setting can 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(67) [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;
```

CREATE ENCRYPTION

Creates encryption keys for use during the encryption process.

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

Argument	Description
Key-name	Name associated with the encryption key. Name must be unique.
For AES|DES	Indicates the level of encryption InterBase will apply to the encrypted data. Advanced Encryption Standard (AES) is considered a strong encryption scheme and requires a license to use with InterBase. Data Encryption Standard (DES) is considered a weak encryption scheme that requires no special license.

Description: ***CREATE ENCRYPTION*** creates an encryption key. Only a SYSDSO (Data Security Owner) can create an encryption key. An encryption key is used to encrypt pages and/or columns of a database. The database owner uses an encryption key to perform encryption on a specific database or column. InterBase stores encryption keys in the ***RDB$ENCRYPTIONS*** system table.

Three new columns have been added to the ***RDB$RELATIONS_FIELDS*** table: ***RDB$ENCRYPTION_ID***, ***RDB $DECRYPT_DEFAULT_VALUE*** and ***RDB$DECRYPT_DEFAULT_SOURCE*** to support the database page and column-level encryption as well.

Example: The following ***isql*** statement creates an encryption key called revenue_key using the AES encryption scheme and a length of 192 bits:

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

CREATE EXCEPTION

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

```
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;
. . .
```

CREATE GENERATOR

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

```
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.

You can use ***SET GENERATOR*** to set or change the value of an existing generator when writing triggers, procedures, or SQL statements that call ***GEN_ID()***.

CREATE INDEX

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

```
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 the speed of data access. Using an index for columns that appear in a ***WHERE*** clause may improve search performance.

IMPORTANT

You cannot index Blob columns or arrays.

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 s ***ORDER BY*** clause of the query. 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);
```

CREATE JOURNAL

Creates a journal file and activates journaling.

```
CREATE JOURNAL [<journal-file-specification>] [LENGTH <number-of-pages>]
[CHECKPOINT LENGTH <number-of-pages> [PAGES]]
[CHECKPOINT INTERVAL <number-of-seconds> [SECONDS]]
[PAGE SIZE <number-of-bytes> [BYTES]]
[PAGE CACHE <number-of-buffers> [BUFFERS]]
[[NO] TIMESTAMP NAME]
```

Argument	Description
journal-file-specification	Specifies a quoted string containing the full path and base file name of the journal file. The base journal file name is used as a template for the actual journal file names as they are created. The default is the full database path and file name.
LENGTH	This clause specifies the number of pages that can be written to the journal file before rolling over to a new journal file. The maximum length is 2GB or 4000 pages.
CHECKPOINT LENGTH	This clause specifies the number of pages that can be written to the journal file before checkpoint occurs. The default is 500.
CHECKPOINT INTERVAL	Determines the number of seconds between database checkpoints. The checkpoint interval determines how long it will take to recover after a server crash. The default is 0. **Note**: If both CHECKPOINT LENGTH and CHECKPOINT INTERVAL are specified, whichever event occurs first will initiate a database checkpoint.
PAGE SIZE	Determines the size of a journal page in bytes. A journal page size must be at least twice the size of a database page size. If a journal page size of less is specified, it will

Argument	Description
	be rounded up to twice the database page size and a warning will be returned. The journal page size needs not be a power of 2. The default is twice the database size.
PAGE CACHE	Determines the number of journal pages that are cached to memory. This number must be large enough to provide buffers for worker threads to write to when the cache writer is writing other buffers. If the number is too small, the worker threads wait and performance suffers.The default is 100 buffers.
[NO]TIMESTAMP NAME	Determines whether or not to append the file creation timestamp to the base journal file name. The default is enabled. If used, the base journal file name will be appended with a timestamp in the following format: ***YYYY_MM_DDTHH_MM_SSZ.sequence_number.journal***
[NO] PREALLOCATE	Determines journal file space requirements while simultaneously guaranteeing that the space is allocated in advance. The default is twice the database size.

Description: A journal consists of one or more journal files. A journal file records each database transaction as it occurs. To save changed journal pages in the database cache to the hard disk, you set up journaling checkpoints to occur automatically. A checkpoint specifies the time at which InterBase must save all the changed pages in the database cache to the database file.

The ***CREATE JOURNAL*** statement causes all subsequent write operations on a database to be done asynchronously. The journal file I/O is always synchronous and cannot be altered. All transaction changes are safely recorded on durable storage before the transaction is committed.

Journaling can be used with journal archiving to provide more complete disaster recovery.

Example: In the following example:

```
CREATE JOURNAL 'e:\database\test'
LENGTH 4000
CHECKPOINT LENGTH 10000
PAGE CACHE 2500;
```

The ***LENGTH*** parameter of 65000 will cause rollover to a new journal file every 1GB (65000 x 16KB). A ***CHECKPOINT LENGTH*** parameter of 10000 means the database checkpoint will occur every 160MB (10000 x 16KB). The 2500 journal buffer configuration will leave 2000 spare buffers for the worker threads to dump their journal changes. At the built-in ***PAGE CACHE*** default of 100, the worker threads can stall due to a high rate of journal buffer wait states.

CREATE JOURNAL ARCHIVE

Activities journal archiving and performs the initial database dump to the archive directory.

```
CREATE JOURNAL ARCHIVE <journal archive directory>
```

Argument	Description

journal archive directory	The location in which InterBase stores the journal archive. If the directory does not exist or is not accessible, InterBase returns an error message. The directory path can be a local drive, a mapped drive, or an UNC path (as long as the underlying file APIs can open the file using that specification). If you do not specify a journal archive directory in the ***CREATE JOURNAL ARCHIVE*** statement, InterBase uses the journal directory created with the ***CREATE JOURNAL*** statement.

Description: The ***CREATE JOURNAL ARCHIVE*** command performs two functions: it activates journal archiving in an InterBase database, and it automatically performs the initial full, physical dump of the database. InterBase stores the dump in the journal archive directory you specify in the ***CREATE*** statement. A journal archive enables you to recover to the last committed transaction in the most recently archived and completed journal file.

IMPORTANT

CREATE JOURNAL ARCHIVE creates the archive and performs an initial dump. However, you must issue a specific ***gbak*** command to copy completed journal files to the journal archive. You use another ***gbak*** command to perform subsquent dumps to the archive. For information about the ***gbak*** archive commands, and about how to implement journaling and journal archiving, see the InterBase Operations Guide.

Journal Archive Management

You can manage the Journal Archive feature of InterBase V8. The archive is a directory that holds journal files, which have been archived from the local journal directory associated with a database. In addition, to storing copies of the local journal files, the archive also stores database dumps that are periodically backed up to the archive.

Description: Archived database dumps represent the starting point from which long-term database recovery is initiated. A set of archive journal files are applied to a copy of the archive database in the same way that local journal files are applied to a production database during short-term recovery. Also, an InterBase timestamp can be specified to indicate a point-in-time until which the journal files will be applied.

When the archive is used to recover a database, the resulting database is not a journaled database. This means that RDBLOG_FILES, RDBJOURNAL_FILES and the log page of the database are empty. This prevents the database from accidently using the journal and journal archive of an existing database. Database recovery is usually used when the original database is corrupted or unavailable due to hardware failures. However, it could be possible to recover a database on the same machine as the working production database or on a different machine where the journal and journal archive directories have no similarly-named directories. Therefore, if journaling and/or journal archiving is desired for the recovered database, it is necessary to execute the appropriate DDL commands to do so.

Examples: gbak is used to archive databases and journal files to the archive, and is also used to recover a database from the archive back to a specified local directory of the user's choice.

```
To archive a database:
gbak -archive_database <dbname>
To archive local journal files:
gbak -archive_journals <dbname>
```

To recover a database (optionally to a point-in-time)

```
gbak -archive_recover [-until <timestamp>] <archive_dbname> <local_dbname>
```

If the -until command line switch is not given, the database recover applies as many journal files as possible to recover a database to the most recent point-in-time. If possible, the database recovery attempts to "jump" from the archive to the local journal directory to apply the journal files that were never copied to the archive. In this way, a database may be recovered to the most recently committed transaction of the original database.

If allowed, the archive grows in storage size infinitely as the database and the most current journal files are continually archived. gfix is used to manage and garbage collect archived items that are no longer required. As the number of journal files grows in the archive without have created more recent archived database dumps, so does the time that will be needed to recover the database from the archive. Therefore, it is desirable to periodically create additional database dumps in the archive. At some point, you may decided that older database dumps and the journal files on which they depend on are no longer necessary, as the basis of recovery will be on more recent database dumps and journal files.

All archive items are denoted by an archive sequence number that corresponds to the order in which the items were created in the archive.

To garbage collect archive items less than an archive sequence number.

```
gfix -archive_sweep [-force] <archive_sequence_no>
```

If an archive item cannot be swept for some reason, the sweep stops and returns an error status. In some cases, this could stop the command from ever succeeding. For example, if an archive is manually deleted with a shell OS command, the sweep always fails because it cannot find the file to drop. The -force option continues regardless of errors to delete as much as possible. The -force switch logs errors to the InterBase error log instead of returning an error status.

To specify how many database dumps to allow in the archive:

```
gfix -archive_dumps <number>
```

Once the number of database dumps in the archive exceeds the <number> given, all lower sequenced archive items are deleted from the archive. Sometimes all lower sequenced items cannot be deleted. For example, a database dump may depend on a lower sequenced journal file with which to start recovery. In that case, InterBase automatically adjusts the given sequence number lower so that this dependency is not lost.

To track that state of the archive, a new system table, RDB$JOURNAL ARCHIVES, has been added for ODS 12 databases. The gbak and gfix commands listed above used this system table to decide which archive items are targets for the commands.

IMPORTANT

Listed below are the requirements and constraints for managing the Journal Archive.

1. The archive is platform-specific. An archive created with InterBase for Windows cannot be directly used to recover on InterBase for Unix. Instead, an archived database dump could be logically backed up in transportable format and then logically restored on the other platform.
2. The journal and journal archive are restricted to a single directory. The number of items allowed to be archived will be limited to the number of files that are allowed in a directory for a give file system.

3. Only full database dumps are archived. In particular, it is not possible to archive incremental database dumps.
4. Journaling must be enabled for a database before the database can be configured for journal archiving.

CREATE PROCEDURE

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

```
CREATE PROCEDURE '' name
'' [(<param>'' ''data_type [, <param>'' ''data_type'' ''...])]
[RETURNS param data_type'' [, ''<param>'' ''data_type ...])]
AS ''procedure_body '';
procedure_body =

[variable_declaration_list]
block
variable_declaration_list =

DECLARE VARIABLE var data_type;
[DECLARE VARIABLE var data_type; ...]
block =
BEGIN
compound_statement

[compound_statement ...]
END
compound_statement = block | statement;
data_type = { SMALLINT | INTEGER | FLOAT | DOUBLE PRECISION}
| {DECIMAL | NUMERIC} [(PRECISION [, scale])]
| {DATE | TIME | TIMESTAMP)
| {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR}
[(INT)] [CHARACTER SET charname]
| {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR} [VARYING] [(INT)]
| BOOLEAN
```

Argument	Description
<name>	Name of the procedure. Must be unique among procedure, table, and view names in the database.
<param data_type>	Input parameters that the calling program uses to pass values to the procedure: <param>: Name of the input parameter, unique for variables in the procedure. <data_type>: An InterBase data type.
***RETURNS** <param data_type>*	Output parameters that the procedure uses to return values to the calling program: <param>: Name of the output parameter, unique for variables within the procedure. <data_type>: An InterBase data type. The procedure returns the values of output parameters when it reaches a SUSPEND statement in the procedure body.

Argument	Description
AS	Keyword that separates the procedure header and the procedure body.
DECLARE VARIABLE	Declares local variables used only in the procedure; must be preceded by DECLARE VARIABLE and followed by a semicolon (;). is the name of the local variable, unique for variables in the procedure.
<statement>	Any single statement in InterBase procedure and trigger language; must be followed by a semicolon (;) except for BEGIN and END statements.

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 metadata of a database. 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***, F***OR 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.

InterBase does not allow database changes that affect the behavior of an existing stored procedure (for example, ***DROP TABLE*** or ***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 generators and UDFs that are linked with the database.

- 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 Procedures and Triggers.

Language extensions for stored procedures	
Statement	**Description**
BEGIN ... END	Defines a block of statements that executes as one. • The ***BEGIN*** keyword starts the block; the ***END*** keyword terminates it. • Neither should end with a semicolon.
variable = expression	Assignment statement: 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 that returns an error message to the calling application unless handled by a ***WHEN*** statement.
EXECUTE PROCEDURE <proc_name> [[, ...]] [RETURNING_VALUES[, ...]]	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> is like a normal ***SELECT*** statement.
<compound_statement>	Either a single statement in procedure and trigger language or a block of statements bracketed by ***BEGIN*** and ***END***.
IF (<condition>) THEN <compound_statement> [ELSE <compound_statement>]	Tests <condition>, and if it is ***TRUE***, performs the statement or block following ***THEN***; otherwise, performs the statement or block following ***ELSE***, if present. <condition>: a Boolean expression (***TRUE***, ***FALSE***, or ***UNKNOWN***), generally two expressions as operands of a comparison operator.
NEW.<column>	New context variable that indicates a new column value in an ***INSERT*** or ***UPDATE*** operation.
OLD.<column>	Old context variable that indicates a column value before an ***UPDATE*** or ***DELETE*** operation.
POST_EVENT <event_name> | <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>: • Tests <condition>, and performs <compound_statement> if condition is ***TRUE.*** • Repeats this sequence until <condition> is no longer ***TRUE.***

Language extensions for stored procedures	
Statement	**Description**
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*** errcode. • ***ANY:*** Handles any errors.

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

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

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

```
CREATE PROCEDURE SUB_TOT_BUDGET (HEAD_DEPT CHAR(3))
RETURNS (tot_bw1udget 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 ;
```

The following ***SELECT*** procedure, ORG_CHART, displays an organizational chart that shows the department name, the parent department, the department manager, the manager's job title, and the 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	MGR_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.

CREATE ROLE

Creates a role.

```
CREATE ROLE <rolename>;
```

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
<rolename>	Name associated with the role; must be unique among role names in the database

Description: Roles created with ***CREATE ROLE*** can be granted privileges just as users can. These roles can be granted to users, who then inherit the privilege list that has been granted to the role. Users must specify the role at connect time. Use ***GRANT*** to grant privileges (***ALL***, ***SELECT***, ***INSERT***, ***UPDATE***, ***DELETE***, ***EXECUTE***, ***REFERENCES***) to a role and to grant a role to users. Use ***REVOKE*** to revoke them.

Example: The following statement creates a role called "administrator."

```
CREATE ROLE administrator;
```

CREATE SHADOW

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

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

[fileinfo]

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 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 no shadow is available. • All attachments and accesses succeed. • Deletes all references to the shadow and detaches the shadow file.
MANUAL	Specifies that database attachments and accesses fail until a shadow becomes available, *or* until 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; must be a local file system and must not include a node name or be on a networked file system.
LENGTH [=] <int> [PAGE[S]]	Length in database pages of an additional shadow file; page size is determined by the 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.ib***:

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

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

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

The following ***isql*** statements create a multiple-file shadow set for the ***employee.ib*** 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';
```

CREATE SUBSCRIPTION

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

```
CREATE SUBSCRIPTION <subscription_name> ON
<table>[(column_comma-list)]:[FOR ROW ({INSERT, UPDATE, DELETE})
], <table>[(column-comma_list)][FOR ROW ({INSERT, UPDATE, DELETE})] ...]
[DESCRIPTION user-description];
```

Argument	Description
FOR ROW	Determines what types of row modification causes column-level changes.
<table>	If a table is specified, all table columns are tracked.
column_comma-list	Specifies a subset of columns to be tracked.
user-description	

Description: The ***FOR*** clause tailors what types of row modifications causes column-level changes to be tracked for the subscription. If the ***FOR*** clause is omitted then all data changing row operations cause column data to be tracked for the subscription. If a table alone is specified then all columns of the table are tracked. If only a subset of columns is desired to be tracked, then an optional list of columns can be specified by the subscription.

An optional list of columns is specified for the "Employees" table so that only changes on those columns are tracked. Since no FOR clause is specified for "Employees" the default of ***FOR*** assumes that all insert,

update, and delete changes are tracked by the subscription. The "Customer" table clause specifies that only row deletions are tracked.

- If you no longer want to observe a set of changed views, the subscription must be dropped.
- If ***RESTRICT*** is specified then a check of existing subscribers is performed. If there are subscribers then an error is returned without dropping the subscription.
- If ***CASCADE*** is specified then all subscribers of this subscription are also dropped.
- If neither ***RESTRICT*** nor ***CASCADE*** is specified then ***RESTRICT*** is assumed.

Example: If only a subset of columns is desired to be tracked, then an optional list of columns can be specified by the subscription.

```
CREATE SUBSCRIPTION "Subscribed_Changes" ON "Employees" (NAME, DEPARTMENT, SALARY),
"Customers" FOR ROW (DELETE).
```

To create your subscriptions (the first line shows new employees, the second shows customer records that were deleted).

```
CREATE SUBSCRIPTION "Subscribed_Inserts" ON "Employees" (FULL_NAME, DEP_NO, SALARY) FOR
ROW (INSERT)
CREATE SUBSCRIPTION "Customer_Deletes" ON "Customer" FOR ROW (DELETE)
```

CREATE TABLE

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

IMPORTANT

To create a global Temporary table, see: "global Temporary Tables" in the Data Definition Guide.

```
CREATE TABLE <table> [EXTERNAL [FILE] '<filespec>']
(col_def [, col_def | tconstraint ...]) [ON COMMIT {PRESERVE | DELETE} ROWS] [[NO] RESERVE
SPACE];
col_def = col {data_type | COMPUTED [BY] (expr) | DOMAIN}
[DEFAULT {literal | NULL | USER}]
[NOT NULL]
[col_constraint]
[COLLATE collation]
data_type =
{SMALLINT | INTEGER | FLOAT | DOUBLE PRECISION}[array_dim]
| {DATE | TIME | TIMESTAMP} [array_dim]
| {DECIMAL | NUMERIC} [(PRECISION [, scale])] [array_dim]
| {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR} [(INT)]
[array_dim] [CHARACTER SET charname]
| {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
[VARYING] [(INT)] [array_dim]
| BLOB [SUB_TYPE {INT | subtype_name}] [SEGMENT SIZE INT]
[CHARACTER SET charname]
```

```
| BLOB [(seglen [, subtype])]
| BOOLEAN
array_dim = [[x:]y [, [x:]y ...]]
expr = A valid SQL expression that results IN a single VALUE.
col_constraint = [CONSTRAINT CONSTRAINT]
{ UNIQUE
| PRIMARY KEY
| REFERENCES other_table [(other_col [, other_col ...])]
[ON DELETE {RESTRICT|NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
[ON UPDATE {RESTRICT|NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
| CHECK (search_condition)}
tconstraint = [CONSTRAINT CONSTRAINT]
{{PRIMARY KEY | UNIQUE} (col [, col ...])
| FOREIGN KEY (col [, col ...])
REFERENCES other_table [(other_col [, other_col ...])]
[ON DELETE {RESTRICT|NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
[ON UPDATE {RESTRICT|NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
| CHECK (search_condition)}
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 {>= | <=}
| 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]
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 data_type)
| UPPER (val)
| GEN_ID (generator, val)
operator = {= | < | > | <= | >= | !< | !> | <> | !=}
select_one = SELECT ON a single COLUMN; RETURNS exactly one VALUE.
select_list = SELECT ON a single COLUMN; RETURNS zero OR more VALUES.
select_expr = SELECT ON a list OF VALUES; RETURNS zero OR more VALUES.
```

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.

Notes on the ***CREATE TABLE*** statement:

- When declaring arrays, you must include the outermost brackets, shown below in bold. For example, the following statement creates a 5 by 5 two-dimensional array of strings, each of which is 6 characters long:

```
my_array VARCHAR(6)[5,5]
```

- Use the colon (:) to specify an array with a starting point other than 1. The following example creates an array of integers that begins at 10 and ends at 20:

```
my_array INTEGER[10:20]
```

- In SQL and ***isql,*** you cannot use val as a parameter placeholder (like "?").
- In DSQL and ***isql,*** val cannot be a variable.
- You cannot specify a ***COLLATE*** clause for Blob columns.
- expr is any complex SQL statement or equation that produces a single value.

Argument	Description
<table>	Name for the table; must be unique among table and procedure names in the database.
EXTERNAL [FILE]'<filespec>'	Declares that data for the table under creation 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. You can also encrypt/decrypt a column when you create a table. For instructions on how to encrypt and decrypt a column or database see "Encrypting Your Data" in the Data Definition Guide.
<data_type>	SQL data type for the column; see Data Types.
COMPUTED [BY](<expr>)	Specifies that the value of the data of the coulmn is calculated from <expr> at runtime and is therefore not allocated storage space in the database. • <expr> can be any arithmetic expression valid for the data types in the expression. • Any columns referenced in <expr> must exist before they can be used in <expr>. • <expr> cannot reference Blob columns. • <expr> must return a single value, and cannot return an array.
<domain>	Name of an existing domain
DEFAULT	Specifies a default column value that is entered when no other entry is made; possible values are: • <literal>: Inserts a specified string, numeric value, or date value. • ***NULL***: Enters a ***NULL*** value.

Argument	Description
	• ***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>	Name of a column or table constraint; the constraint name must be unique within the table.
<constraint_def>	Specifies the kind of column constraint; valid options are ***UNIQUE***, ***PRIMARY KEY***, ***CHECK***, and ***REFERENCES***.
REFERENCES	Specifies that the column values are derived from column values in another table; if you do not specify column names, InterBase looks for a column with the same name as the referencing column in the referenced table.
ON DELETE |ON UPDATE	Used with ***REFERENCES***: Changes a foreign key whenever the referenced primary key changes; valid options are: • ***[Default] NO ACTION***: Does not change the foreign key; may cause the primary key update to fail due to referential integrity checks. • ***CASCADE***: For ***ON DELETE***, deletes the corresponding foreign key; for ***ON UPDATE***, updates the corresponding foreign key to the new value of the primary key. • ***SET NULL***: Sets all the columns of the corresponding foreign key to ***NULL***. • ***SET DEFAULT***: Sets every column of the corresponding foreign key is set to its default value in effect when the referential integrity constraint is defined. When the default for a foreign column changes after the referential integrity constraint is defined, the change does not have an effect on the default value used in the referential integrity constraint.
CHECK <search_condition>	An attempt to enter a new value in the column fails if the value does not meet the <search_condition>.
COLLATE <collation>	Establishes a default sorting behavior for the column; see Character Sets and Collation Orders for more information.

Description: ***CREATE TABLE*** establishes a new table, its columns, and integrity constraints in an existing database. The user who creates a table is the owner of the table 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 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 given character set of the column, 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 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.

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.

External files must either be placed in ***<InterBase_home>/ext*** or their location must be specified in the ***ibconfig*** configuration file using the ***EXTERNAL_FILE_DIRECTORY*** entry.

Referential integrity constraints:

- You can define integrity constraints at the time you create a table. These constraints are rules that validate data entries by enforcing column-to-table and table-to-table relationships. They span all transactions that access the database and are automatically maintained by the system. ***CREATE TABLE*** supports the following integrity constraints:
- A ***PRIMARY KEY*** is one or more columns whose collective contents are guaranteed to be unique. A ***PRIMARY KEY*** column must also define the ***NOT NULL*** attribute. A table can have only one primary key.
- ***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 (***REFERENCES***) ensure that values in the specified columns (known as the foreign key) are the same as values in the referenced ***UNIQUE*** or ***PRIMARY KEY*** columns in another table. The ***UNIQUE*** or P***RIMARY KEY*** columns in the referenced table must be defined before the ***REFERENCES*** constraint is added to the secondary table. ***REFERENCES*** has ***ON DELETE*** and ***ON UPDATE*** clauses that define the action on the foreign key when the referenced primary key is updated or deleted. The values for ***ON UPDATE*** and ***ON DELETE*** are as follows:

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

Action specified	Effect on foreign key
	The default value is the one in effect when the referential integrity constraint was defined. When the default for a foreign key column is changed after the referential integrity constraint is set up, the change does not have an effect on the default value used in the referential integrity constraint.
SET NULL	Every column of the corresponding foreign key is set to ***NULL***.

- You can create a ***FOREIGN KEY*** reference to a table that is owned by someone else only if that owner has explicitly granted you ***REFERENCES*** privilege on that table. Any user who updates your foreign key table must have ***REFERENCES*** or ***SELECT*** privileges on the referenced primary key table.
- ***CHECK*** constraints enforce a <search_condition> that must be true for inserts or updates to the specified table. <search_condition> can require a combination or range of values or can compare the value entered with data in other columns.

NOTE

Specifying ***USER*** as the value for a <search_condition> references the login of the user who is attempting to write to the referenced table.

- 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 ***RDB$RELATION_CONSTRAINTS*** system table.

NOTE

Constraints are not enforced on expressions.

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));
```

In the next example, the F2 column in table T2 is a foreign key that references table T1 through the primary key P1 of T1. When a row in T1 changes, that change propagates to all affected rows in table T2. When a row in T1 is deleted, all affected rows in the F2 column of table T2 are set to ***NULL***.

```
CREATE  TABLE  T1  (P1  INTEGER  NOT  NULL  PRIMARY  KEY);
CREATE  TABLE  T2  (F2  INTEGER  FOREIGN  KEY  (F2)  REFERENCES  T1  (P1)
ON  UPDATE  CASCADE
ON  DELETE  SET  NULL);
```

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 default character set of the dataset. 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);
```

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

NO RESERVE SPACE for Database and User Tables

This feature is useful if you have very, large databases (VLDB) with tables that are archival in nature. An archival table means that the rows of a table are infrequently or never UPDATED or DELETED; have complex queries, such as aggregates and analytics that process a high percentage of rows; and where indexes are rebuilt and the database is backed and/or restored frequently. These database operations could see a performance improve of 20% or more with a savings in storage space.

By default, InterBase reserves a small amount of space in each data page of a table to optimize UPDATE and DELETE operations on resident rows. This reserve space can amount to 20% or more of the total space occupied by all of the rows of the table. Some tables archive historical data or data that are UPDATED infrequently or not at all and their rows may never be deleted. Database operations that process most or all of the rows, such as backup, restore, index creation, aggregate computation have always suffered performance penalties proportional to this reservation overhead.

For this reason, a CREATE/ALTER TABLE clause is introduced that prevents space reservation and maximizes row packing for the most efficient fill ratio. At the database level, it has been possible to restore a database with the -USE_ALL_SPACE switch so that no space is reserved for any table. To change the storage behavior in a like manner for new or existing databases, the same clause is introduced for CREATE/ALTER DATABASE.

User Interface To effect the new storage behavior, a non-standard SQL clause is added:

Clause is presented before the secondary file specification.

```
CREATE DATABASE <file name> ... [NO] RESERVE SPACE
```

Clause is presented after the column list specification and optional ON COMMIT clause for temporary tables.

```
CREATE TABLE <TABLE name> ... [NO] RESERVE SPACE
```

This causes newly INSERTED rows to not reserve space on their data page for a DELETE record version stub, as would normally be the case. Over many row insertions, a decrease in storage size should be observed relative to what the table size would be in the absence of this feature. The optional NO keyword when used with ALTER TABLE toggles the behavior to the alternate state of the current storage behavior for the table.

The NO RESERVE storage modifier is preserved across database backup and restore. This state is stored as flag bit 64 (0x100) of RDB$RELATIONS.RDB$FLAGS for the user's table entry in the system table RDB $RELATIONS.

The clause is displayed by ISQL's SHOW TABLE command following the enumeration of a table's column definitions. It is also visible using ISQL's Extract (-x) command in a syntax-correct manner for the CREATE TABLE output of the respective table listing. The state for database-wide storage behavior is stored in a like manner for the RDB$DATABASE entry in RDB$RELATIONS.

ON COMMIT

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

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

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

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

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

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

```
CREATE TABLE <table> [EXTERNAL [FILE] '<filespec>']
(col_def [, col_def | tconstraint ...]) [ON COMMIT {PRESERVE | DELETE} ROWS] [[NO] RESERVE
SPACE];
```

CREATE TRIGGER

```
CREATE TRIGGER name FOR TABLE
[ACTIVE | INACTIVE]
{BEFORE | AFTER}
{DELETE | INSERT | UPDATE}
[POSITION NUMBER]
AS trigger_body ;
trigger_body = [variable_declaration_list] block
variable_declaration_list =
DECLARE VARIABLE variable data_type;
[DECLARE VARIABLE variable data_type; ...]
block =
BEGIN
compound_statement
[compound_statement ...]
END
data_type = SMALLINT
| INTEGER
| FLOAT
| DOUBLE PRECISION
| {DECIMAL | NUMERIC} [(PRECISION [, scale])]
| {DATE | TIME | TIMESTAMP)
| {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR}
[(INT)] [CHARACTER SET charname]
| {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR} [VARYING] [(INT)]
| BOOLEAN
compound_statement = block | statement;
```

Argument	Description
<name>	Name of the trigger; 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 the associated operation. • AFTER: After the 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 the 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	Declares local variables used only in the trigger. Each declaration must be preceded by ***DECLARE VARIABLE*** and followed by a semicolon (;). • : Local variable name, unique in the trigger. • <data_type>: The data type of the local variable.
<statement>	Any single statement in InterBase procedure and trigger language; each statement except ***BEGIN*** and ***END*** must be followed by a semicolon (;).

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.

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_nam>e. 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 generators and UDFs that are linked with the calling application.
- 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 Procedures and Triggers.

Language extensions for triggers	
Statement	**Description**
BEGIN ...END	Defines a block of statements that executes as one. • The ***BEGIN*** keyword starts the block; the ***END*** keyword terminates it. • Neither should it be followed by a semicolon.
<variable> = <expression>	Assignment statement that 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 that returns an error message to the calling application unless handled by a ***WHEN*** statement.
EXECUTE PROCEDURE < proc_name> [[, ...]] [RETURNING_VALUES[, ...]]	Executes the stored procedure, <proc_name>, with the listed input arguments. • Returns 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.

Language extensions for triggers	
Statement	**Description**
FOR <select_statement> DO <compound_statement>	Repeats the statement or block following ***DO*** for every qualifying row retrieved by <select_statement>.
<select_statement>	A normal ***SELECT*** statement.
<compound_statement>	Either a single statement in procedure and trigger language or a block of statements bracketed by ***BEGIN*** and ***END***.
IF (condition) THEN compound_statement [ELSE compound_statement]	Tests <condition>, and if it is ***TRUE***, performs the statement or block following ***THEN***; otherwise, performs the statement or block following ELSE, if present.
<condition>	A Boolean expression (***TRUE***, ***FALSE***, or ***UNKNOWN***), generally two expressions as operands of a comparison operator.
NEW.<column>	New context variable that indicates a new column value in an ***INSERT*** or ***UPDATE*** operation.
OLD.<column>	Old context variable that indicates a column value before an ***UPDATE*** or ***DELETE*** operation.
POST_EVENT <event_name> | <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>. • Tests <condition>, and performs <compound_statement> if <condition> is ***TRUE***. • Repeats this sequence 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***. • ANY: Handles any errors
<error>	***EXCEPTION <exception_name>, SQLCODE <errcode> or GDSCODE errcode***

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

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

- *OPEN*
- *FETCH*

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:

```
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 ;
```

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.

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

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

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

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.

CREATE USER

Create a new user. Available in ***DSQL*** and ***isql***.

```
CREATE USER <name> SET
[PASSWORD <password>]
[[NO] DEFAULT ROLE <name>]
[[NO] SYSTEM USER NAME <name>]
[[NO] GROUP NAME <name>]
[[NO] UID <number>]
[[NO] GID <number>]
[[NO] DESCRIPTION <string>]
[[NO] FIRST NAME <string>]
[[NO] MIDDLE NAME <string>]
[[NO] LAST NAME <string>]
[ACTIVE]
[INACTIVE];
```

Argument	Description
PASSWORD	Password of user
[NO]DEFAULT ROLE	Default role
[NO] SYSTEM USER NAME	System user name for target user
[NO]GROUP NAME	Group name for target user
[NO] UID	Target user ID
[NO] GID	Group ID for target user
[NO] DESCRIPTION	Description
[NO]FIRST NAME	First name for target user
[NO] MIDDLE NAME	Middle name for target user
[NO]LAST NAME	Last name for target user
ACTIVE	Default. After inactive, reinstates selected user.
INACTIVE	Prevents a user from logging into database.

Description: ***CREATE USER*** creates a new user. Only used with database under embedded user authentication. If you choose to set more than one property value for the user, include a comma between each property value pair.

NOTE

When ***NO*** is specified, an argument to the option must not be supplied. ***NO*** sets the option to a ***NULL*** state.

Examples: The following statement creates the user, JDOE and set password, jdoe:

```
CREATE USER JDOE SET PASSWORD 'jdoe';
```

The next statement creates the user, JDOE, and set password, first name and last name:

```
CREATE USER JDOE SET PASSWORD 'jdoe', FIRST NAME 'Jane', LAST NAME 'Doe';
```

CREATE VIEW

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

```
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 ***SELECT*** clause of the view.

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, roles, triggers, views, 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*** clause, 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

You cannot select from a view that is based on the result set of a stored procedure.

An updatable view cannot have ***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 ***SELECT*** statement of the view 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;
```

DECLARE CURSOR

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

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 ***DECLA RECURSOR***; the resulting rows become the *active set* of the cursor.
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 the 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
```

DECLARE CURSOR (BLOB)

Declares a Blob cursor for read or insert. Available in ***gpre***.

```
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;
```

DECLARE EXTERNAL FUNCTION

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

```
DECLARE EXTERNAL FUNCTION name [data_type
| CSTRING (<int>) [, data_type | CSTRING (<int>) ...]]
RETURNS {data_type [BY VALUE] | CSTRING (<int>) | PARAMETER <n>} [FREE_IT]
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.

NOTE

Whenever a UDF returns a value by reference to dynamically allocated memory, you must declare it using the FREE_IT keyword in order to free the allocated memory.

Argument	Description
<name>	Name of the UDF to use in SQL statements; can be different from the name of the function specified after the ENTRY_POINT keyword.
<data_type>	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.
CSTRING (<int>)	Specifies a UDF that returns a null-terminated string <int> bytes in length.
RETURNS	Specifies the return value of a function.
BYVALUE	Specifies that a return value should be passed by value rather than by reference.
PARAMETER <n>	• Specifies that the <n>th input parameter is to be returned. • Used when the return data type is BLOB.
FREE_IT	Frees memory of the return value after the UDF finishes running. • Use only if the memory is allocated dynamically in the UDF • See also Error Codes and Messages.
'*<entryname>*'	Quoted string that contains the function name as it is stored in the library that is referenced by the UDF.
'*<modulename>*'	Quoted specification identifying the library that contains the UDF. • The library must reside on the same machine as the InterBase server. • On any platform, the module can be referenced with no path name if it is in <<InterBase_home>>/UDF or <<InterBase_home>>/intl • If the library is in a directory other than <<InterBase_home>>/UDF or <<InterBase_home>>/intl, you *must* specify its location in configuration file (ibconfig) of InterBase using the ***EXTERNAL_FUNCTION_DIRECTORY*** parameter. • It is not necessary to supply the extension to the module name.

Description: ***DECLARE EXTERNAL FUNCTION*** provides information about a UDF to a database: where to find it, its name, the input parameters it requires, and the single value it returns. Each UDF in a library must be declared once to each database where it will be used. As long as the entry point and module name do not change, there is no need to redeclare a UDF, even if the function itself is modified.

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

The module name does not need to include a path. However, the module must either be placed in <<InterBase_home>>/UDF or be listed in the InterBase configuration file using the ***EXTERNAL_FUNCTION_DIRECTORY*** parameter.

To specify a location for UDF libraries in the InterBase configuration file, enter a line of the following form for Windows platforms:

```
EXTERNAL_FUNCTION_DIRECTORY  D:\Mylibraries\InterBase
```

For UNIX, the line does not include a drive letter:

```
EXTERNAL_FUNCTION_DIRECTORY  \Mylibraries\InterBase
```

The InterBase configuration file is called ***ibconfig*** on all platforms.

Examples: The following ***isql*** statement declares the TOPS() UDF to a database:

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

This example does not need the ***FREE_IT*** keyword because only cstrings, ***CHAR***, and ***VARCHAR*** return types require memory allocation.

The next example declares the ***LOWERS()*** UDF and frees the memory allocated for the return value:

```
DECLARE EXTERNAL FUNCTION LOWERS VARCHAR(256)
RETURNS CSTRING(256) FREE_IT
ENTRY POINT 'fn_lower' MODULE_NAME 'udflib';
```

DECLARE FILTER

Declares an existing Blob filter to a database. Available in ***gpre***, ***DSQL***, and ***isql***.

```
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.

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';
```

DECLARE STATEMENT

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

```
DECLARE statement STATEMENT;
```

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

Description: *DECLARE STATEMENT* names a 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;
```

DECLARE TABLE

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

```
DECLARE TABLE TABLE (table_def);
```

Argument	Description
<table>	Name of the table; 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. You must use it if you both create and populate a table 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.

Example: 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));
```

DELETE

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

SQL and DSQL form:

IMPORTANT

Omit the terminating semicolon for DSQL.

```
DELETE [TRANSACTION TRANSACTION] FROM TABLE
{[WHERE search_condition] | WHERE CURRENT OF cursor}
[ORDER BY order_list]
[ROWS VALUE [TO upper_value] [BY step_value][PERCENT][WITH TIES]];
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
WWHERE CURRENT OF <cursor>	Specifies that the current row in the active set of <cursor> is to be deleted
ORDER BY <order_list>	Specifies columns to order, either by column name or ordinal number in the query, and the sort order (ASC or DESC) for the returned rows
ROWS <value> [TO <upper_value>] [BY <step_value>] [PERCENT][WITH TIES]	• <value> is the total number of rows to return if used by itself • <value> is the starting row number to return if used with ***TO*** • <value> is the percent if used with ***PERCENT*** • <upper_value> is the last row or highest percent to return • If <step_value> = <n>, returns every <n>th row, or <n> percent rows • ***PERCENT*** causes all previous ROWS values to be interpreted as percents • ***WITH TIES*** returns additional duplicate rows when the *last* value in the ordered sequence is the same as values in subsequent rows of the result set; must be used in conjunction with ***ORDER BY***

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.

IMPORTANT

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 CURRENTOF*** 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;
```

DESCRIBE

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

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

Argument	Description
OUTPUT	[Default] Indicates that column information should be returned in the XSQLDA.
INPUT	Indicates that dynamic parameter information should be stored in the XSQLDA.

Argument	Description
<statement>	• A previously defined alias for the statement to DESCRIBE. • Use PREPARE to define aliases.
{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;
```

DISCONNECT

Detaches an application from a database. Available in ***gpre***.

```
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;
```

DROP DATABASE

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

```
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;
```

DROP DOMAIN

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

```
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;
```

DROP ENCRYPTION

Used to delete an encryption key from a database.

```
DROP ENCRYPTION key-name [RESTRICT | cascade]
```

Argument	Description
key-name	Specifies the name of the encryption key to drop from the database.
restrict	This is the sub-command which is the default drop behavior.
cascade	Decrypts all fields in all relations encrypted by it.

Description: An encryption key can be dropped (deleted) from the database. Only the SYSDSO can execute this command. The command fails if the encryption key is still being used to encrypt the database. If any table columns are encrypted when "restrict" is specified, which is the default drop behavior, the command also fails. If "cascade" is specified, then all columns using that encryption are decrypted and the encryption is dropped “Restrict” and “Cascade” are the only options available for this command.

In the case of Column-level Encryption use, although ***DROP ENCRYPTION CASCADE*** decrypts all fields in all relations encrypted by it, that decryption process makes back versions of the decrypted records, which remain dependent on the existence of the encryption. The encryption is only marked for deletion.

The next time the database is swept, database sweep completion checks for any record formats that still depend on a “marked for deletion” encryption. If there are none, the encryption is fully deleted at that time.

If you are trying to completely remove all encryption from your database and are presented with an "unsuccessful metadata update encryptions still exist", you need to sweep the database after the ***DROP ENCRYPTION CASCADE*** and before ***ALTER DATABASE SET NO SYSTEM PASSWORD***.

Example: The following example uses the cascade option to decrypt all columns using the revenue_key and to delete the key:

drop encryption revenue_key cascade

DROP EXCEPTION

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

```
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;
```

DROP EXTERNAL FUNCTION

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

```
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.

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

```
DROP EXTERNAL FUNCTION TOPS;
```

DROP FILTER

Removes a Blob filter declaration from a database. Available in ***gpre***, ***DSQL***, and ***isql***.

```
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.

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

```
DROP FILTER DESC_FILTER;
```

DROP GENERATOR

Drops a generator from the database. Available in DSQL, and ***isql.***

```
DROP GENERATOR generator_name
```

Argument	Description
generator_name	Name of the generator.

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

NOTE

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

DROP INDEX

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

```
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

You cannot drop system-defined indexes, such as those for ***UNIQUE***, ***PRIMARY KEY***, and ***FOREIGN KEY***.

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;
```

DROP JOURNAL

Discontinues the use of journaling and deletes existing journal files in the database.

```
DROP JOURNAL
```

Description: The ***DROP JOURNAL*** statement discontinues the use of write-ahead logging and deletes all journal files. This operation does not delete any journal files in the journal archive but does discontinue maintenance of the journal archive. Dropping journal files requires exclusive access to the database.

DROP JOURNAL ARCHIVE

Discontinues journal archiving on the database.

```
DROP JOURNAL ARCHIVE
```

Description: ***DROP JOURNAL ARCHIVE*** disables journal archiving for the database. It causes all journal files and database file dumps to be deleted in all journal archive directories. The file system directories themselves are not deleted.

IMPORTANT

This command does not discontinue journaling and the creation of journal files.

DROP PROCEDURE

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

```
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;
```

DROP ROLE

Deletes a role from a database. Available in ***gpre***, DSQL, and ***isql***.

```
DROP ROLE <rolename>;
```

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
<rolename>	Name of an existing role

Description: ***DROP ROLE*** deletes a role that was previously created using ***CREATE ROLE***. Any privileges that users acquired or granted through their membership in the role are revoked.

A role can be dropped by its creator, the SYSDBA user, or any user with superuser privileges.

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

```
DROP ROLE administrator;
```

DROP SHADOW

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

```
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;
```

DROP SUBSCRIPTION

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

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

IMPORTANT

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

Argument	Description
<***RESTRICT***>	Checks existing subscribers.
CASCADE	All subscribers of the subscription are dropped.

DROP TABLE

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

```
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 the data, metadata, and indexes of a table from a database. It also drops any triggers that reference the table.

A table referenced in a 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;
```

DROP TRIGGER

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

```
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;
```

DROP USER

Deletes an existing user from an embedded user authentication database. Available in DSQL, and ***isql***.

```
DROP USER <name>
```

DROP VIEW

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

```
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;
```

END DECLARE SECTION

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

```
END DECLARE SECTION;
```

Description: ***END DECLARE SECTION*** is used in embedded SQL applications to identify the end of host-language variable declarations for variables 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;
```

EVENT INIT

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

```
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 interest in an event.
2. ***EVENT WAIT*** causes the application to wait until notified of the event occurrence.

EVENT INIT registers an application 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');
```

EVENT WAIT

Causes an application to wait until notified of an event occurrence. Available in ***gpre.***

```
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;
```

EXECUTE

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

```
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 parameters should be taken from the specified XSQLDA
INTOSQL 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 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;
```

EXECUTE IMMEDIATE

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

```
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 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 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;
```

EXECUTE PROCEDURE

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

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 ...]]
```

isqlform:

```
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 a 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;
```

EXTRACT( )

Extracts date and time information from ***DATE***, ***TIME***, and ***TIMESTAMP*** values. Available in ***gpre***, ***DSQL***, and ***isql***.

```
EXTRACT (part FROM VALUE)
```

Argument	Description
<part>	YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, WEEKDAY, or YEARDAY; see the table below for data types and ranges of values
<value>	DATE, TIME, or TIMESTAMP value

Description: The value passed to the EXTRACT() expression must be a DATE, a TIME, or a TIMESTAMP. Extracting a part that does not exist in a data type results in an error. For example, a statement such as tEXTRACT (YEAR from aTime) would fail.

NOTE

The data type of part depends on which part is extracted.

Part extracted	Data type	Range
YEAR	**SMALLINT**	0–5400
MONTH	**SMALLINT**	1–12
DAY	**SMALLINT**	1–31
HOUR	**SMALLINT**	0–23
MINUTE	**SMALLINT**	0–59
SECOND	**DECIMAL(6,4)**	0–59.9999
WEEKDAY	**SMALLINT**	0–6 (0 = Sunday, 1 = Monday, etc.)
YEARDAY	**SMALLINT**	0–365

Example:

```
EXTRACT(HOUR FROM StartTime);
```

FETCH

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

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***.

Argument	Description
<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 *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;
```

FETCH (BLOB)

Retrieves the next available segment of a Blob column and places it in the specified local buffer. Available in ***gpre.***

```
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 the 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;
}
. . .
}
```

GEN ID( )

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

```
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 $-(2^{63})$ to $2^{63} - 1$

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():

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

The first time the trigger fires, NEW.EMPNO is set to 1. Each subsequent firing increments NEW.EMPNO by 1.

GRANT

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

```
GRANT <privileges> ON [TABLE] {<tablename> | <viewname}>
TO {object|userlist [WITH GRANT OPTION]|GROUP <UNIX_group}>
| EXECUTE ON PROCEDURE procname TO {object | userlist}
| <role_granted> TO {PUBLIC | <role_grantee_list}>[WITH ADMIN OPTION];
privileges = ALL [PRIVILEGES] | privilege_list
privilege_list = {
   SELECT
| DELETE
| INSERT
| ENCRYPT ON ENCRYPTION
| DECRYPT
| UPDATE [(col [, col ...])]
| REFERENCES [(col [, col ...])]
}[, privilege_list ...]
object = {
   PROCEDURE procname
| TRIGGER trigname
| VIEW viewname
| PUBLIC
}[, object ...]
userlist = {
   [USER] username
| rolename
| UNIX_user
}[,userlist ...]
role_granted = rolename [, rolename ...]
role_grantee_list = [USER] username [, [USER] username ...]
```

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
<***privilege_list***>	Name of privilege to be granted; valid options are ***SELECT***, ***DELETE***, ***INSERT***, ***UPDATE***, ***ENCRYPT ON ENCRYPTION***, ***DECRYPT***, and ***REFERENCES***.
<***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
GROUP <***unix_group***>	On a UNIX system, the name of a group defined in /etc/group
<***object***>	Name of an existing procedure, trigger, or view; ***PUBLIC*** is also a permitted value.
<***userlist***>	A user in the InterBase security database (admin.ib by default) or a rolename created with ***CREATE ROLE***
WITH GRANT OPTION	Passes ***GRANT*** authority for privileges listed in the ***GRANT*** statement to userlist.
<***rolename***>	An existing role created with the ***CREATE ROLE*** statement
<***role_grantee_list***>	A list of users to whom <rolename> is granted; users must be in the InterBase security database.
WITH ADMIN OPTION	Passes grant authority for roles listed to <role_grantee_list>.

Description: ***GRANT*** assigns privileges and roles for database objects to users, roles, 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.

The following table summarizes available privileges:

Privilege	Enables users to ...
ALL	Perform ***SELECT***, ***DELETE***, ***INSERT***, ***UPDATE***, and ***REFERENCES***
SELECT	Retrieve rows from a table or view
DELETE	Remove rows from a table or view
DECRYPT	After encrypting a column, the database owner or the individual table owner can grant decrypt permission to users who need to access the values in an encrypted column.
ENCRYPT ON ENCRYPTION	Enables the database owner or individual table owner to use a specific encryption key to encrypt a database or column. Only the SYSDSO (Data Security Owner) can grant encrypt permission.
INSERT	Store new rows in a table or view
UPDATE	Change the current value in one or more columns in a table or view; can be restricted to a specified subset of columns.
EXECUTE	Execute a stored procedure
REFERENCES	Reference the specified columns with a foreign key; at a minimum, this must be granted to all the columns of the primary key if it is granted at all.

NOTE

ALL does not include ***REFERENCES*** in code written for InterBase 4.0 or earlier.

- To access a table or view, a user or object needs the appropriate ***SELECT***, ***INSERT***, ***UPDATE***, ***DELETE***, or ***REFERENCES*** privileges for that table or view. ***SELECT***, ***INSERT***, ***UPDATE***, ***DELETE***, and ***REFERENCES*** privileges can be assigned as a unit with ALL.
- A user or object must have ***EXECUTE*** privilege to call a stored procedure in an application.
- For more information about the ***GRANT ENCRYPT ON ENCRYPTION*** and ***GRANT DECRYPT*** permissions, see "Encrypting Your Data" in the Data Definition Guide.
- To grant privileges to a group of users, create a role using ***CREATE ROLE***. Then use ***GRANT*** <privilege> ***TO*** <rolename> to assign the desired privileges to that role and use ***GRANT*** <rolename> ***TO*** <user> to assign that role to users. Users can be added or removed from a role on a case-by-case basis using ***GRANT*** and ***REVOKE***. A user must specify the role at connection time to actually have those privileges. See "ANSI SQL 3 roles" in the Operations Guide for more information about invoking a role when connecting to a database.
- On UNIX systems, privileges can be granted to groups listed in /etc/groups and to any UNIX user listed in /etc/passwd on both the client and server, as well as to individual users and to roles.
- To allow another user to reference a column from a foreign key, grant ***REFERENCES*** privileges on the primary key table or on the primary key columns of the table to the owner of the foreign key table. You must also grant ***REFERENCES*** or ***SELECT*** privileges on the primary key table to any user who needs to write to the foreign key table.

TIP

Make it easy, if read security is not an issue, ***GRANT REFERENCES*** on the primary key table to ***PUBLIC***.

- If you grant the REFERENCES privilege, it must, at a minimum, be granted to all columns of the primary key. When REFERENCES is granted to the entire table, columns that are not part of the primary key are not affected in any way.
- When a user defines a foreign key constraint on a table owned by someone else, InterBase checks that the user has REFERENCES privileges on the referenced table.
- The privilege is used at run time to verify that a value entered in a foreign key field is contained in the primary key table.
- You can grant REFERENCES privileges to roles.
- To give users permission to grant privileges to other users, provide a userlist that includes the ***WITH GRANT OPTION***. Users can grant to others only the privileges that they themselves possess.
- To grant privileges to all users, specify ***PUBLIC*** in place of a list of user names. Specifying ***PUBLIC*** grants privileges only to users, not to database objects.

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

The following example creates a role called "administrator", grants ***UPDATE*** privileges on table1 to that role, and then grants the role to user1, user2, and user3. These users then have ***UPDATE*** and REFERENCES privileges on table1.

```
CREATE ROLE administrator;
GRANT UPDATE ON table1 TO administrator;
GRANT administrator TO user1, user2, user3;
```

GRANT SUBSCRIBE

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

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

IMPORTANT

To set a subscription as active, an application issues a SET SUBSCRIPTION statement.

Argument	Description
<subscription_name>	Implied by the user identity of the database
<user_name>	User identify of the database connection

Description: This ***SET SUBSCRIPTION*** statement allows multiple subscriptions to be activated and includes an ***AT*** clause to denote a destination or device name as a recipient of the subscribed changes. The subscriber user name is implied by the user identity of the database connection. Multiple subscriptions against the same schema object for a user, via the ***AT*** clause, are available for two reasons:

First, each subscription for a user may connote a separate device among many that have a disconnected interest in a change set that is queried independently at different times for different purposes.

Second, some multiuser applications use pooled database connections under the umbrella of a single user name (for example CRM_User or even SYSDBA). In these cases, an alternate identifier must be provided to distinguish which subscription should be used to query a change set. That additional identifier can be thought of as a destination or a "device name".

Example: This is to grant subscribe privileges to that user:

```
GRANT SUBSCRIBE ON SUBSCRIPTION Subscribed_Inserts TO smartphone_user;
```

```
GRANT SUBSCRIBE ON SUBSCRIPTION Customer_Deletes TO smartphone_user;
```

GRANT TEMPORARY SUBSCRIBE

```
GRANT TEMPORARY SUBSCRIBE[(<column_comma-list>)] ON <table_name> TO <user_name>;
REVOKE TEMPORARY SUBSCRIBE[(<column_comma-list>)]ON <table_name> FROM <user_name>;TO
SET a subscription AS active, an application issues a SET SUBSCRIPTION statement.
```

IMPORTANT

The user issues a ***SET SUBSCRIPTION*** command as usual giving the name of the base table instead of a subscription name.

Argument	Description
<column_comma-list>	
<table_name>	
user_name	

Description:

Example: Retrieving Changed Views from ISQL

```
SET SUBSCRIPTION ":Employees" ACTIVE;
SELECT NAME, DEPARTMENT, SALARY :FROM "Employees";
COMMIT;
<Another USER reassigns an existing employee TO another department AND gives another
employee a raise>
SELECT NAME, DEPARTMENT, SALARY FROM "Employees";
<CHANGE> NAME DEPARTMENT SALARY
UPDATE joe sales 50000
UPDATE mary finance 75000
SET SAME;
SELECT NAME, DEPARTMENT, SALARY FROM "Employees";
<CHANGE> NAME DEPARTMENT SALARY
UPDATE <same> sales <same>
UPDATE <same> <same> 75000
COMMIT;
SET SUBSCRIPTION "Employees" INACTIVE;
```

ISQL has a collection of ***SET*** statements that toggle a display set. The ***SET SAME*** display toggle alternates between showing the column data value or its changes state of <same> or the changed data value. The ***CHANGE*** column is a pseudo column that shows the type of DML statement that modified the column value(s). All of this change state is returned by the ***XSQLVAR.SQLIND*** member of the new XSQLDA structure.

Minimal support for changed data views is provided by InterBase SQL with the addition of a ***IS SAME*** or ***IS NOT SAME*** clause as the following example illustrates:

IMPORTANT

Using ***IS NOT SAME*** in ***SELECT*** queries

```
SELECT NAME, DEPARTMENT, SALARY FROM "Employees" WHERE SALARY IS NOT SAME;
<CHANGE> NAME DEPARTMENT SALARY
UPDATE mary finance 75000
```

We see that Joe's department reassignment is not returned since he received no compensation adjustment for a lateral move.

INSERT

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

```
INSERT [TRANSACTION TRANSACTION] INTO object [(<col> [, <col> ...])]
{VALUES (val [, val ...]) | select_expr};
object = tablename | viewname
val = {:variable | constant | expr
| FUNCTION | udf ([val [, val ...]])
| NULL | USER | RDB$DB_KEY | ?
} [COLLATE collation]
constant = num | 'string' | charsetname 'string'
FUNCTION = CAST (val AS data_type)
| UPPER (val)
| GEN_ID (generator, val)
```

Argument	Description
<expr>	A valid SQL expression that results in a single column value
<select_expr>	A ***SELECT*** that returns zero or more rows and where the number of columns in each row is the same as the number of items to be inserted

Notes on the ***INSERT*** statement:

- In SQL and ***isql***, you cannot use val as a parameter placeholder (like "?").
- In DSQL and ***isql***, val cannot be a variable.
- You cannot specify a ***COLLATE*** clause for Blob columns.

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 <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.

IMPORTANT

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';
```

INSERT CURSOR (BLOB)

Inserts data into a Blob cursor in units of a Blob segment-length or less in size. Available in ***gpre.***

```
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);
```

MAX( )

Retrieves the maximum value in a column. Available in ***gpre***, ***DSQL***, and ***isql***.

```
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;
```

MIN( )

Retrieves the minimum value in a column. Available in ***gpre***, ***DSQL***, and ***isql***.

```
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. Use the ***DEFAULT CHARACTER SET*** clause in ***CREATE DATABASE TO*** specify a default character set for an entire database, or the ***COLLATE*** clause in ***CREATE TABLE*** to specify a character set at the column level.

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;
```

NULLIF( )

The ***NULLIF*** function returns a null value if the arguments are equal, otherwise it returns the value of the first argument.

```
NULLIF (<expression1>, <expression2>)
```

Description: The ***COALESCE*** and ***NULLIF*** expressions are common, shorthand forms of use for the ***CASE*** expression involving the ***NULL*** state. A ***COALESCE*** expression consists of a list of value expressions. It evaluates to the first value expression in the list that evaluates to ***non-NULL.*** If none of the value expressions in the list evaluates to ***non-NULL*** then the ***COALESCE*** expression evaluates to ***NULL***.

The ***NULLIF*** expression consists of a list of two value expressions. If the two expressions are unequal then the ***NULLIF*** expression evaluates to the first value expression in the list. Otherwise, it evaluates to ***NULL***.

Example: The following example demonstrates the use of ***CASE*** using the sample employee.ib database:

```
SELECT NULLIF(department, head_dept) FROM department
```

OPEN

Retrieve specified rows from a cursor declaration. Available in ***gpre*** and DSQL.

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***	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 release system resources

Examples: The following embedded SQL statement opens a cursor:

```
EXEC SQL
OPEN C;
```

OPEN (BLOB)

Opens a previously declared Blob cursor and prepares it for reading or inserting. Available in ***gpre***.

```
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;
```

PREPARE

Prepares a dynamic SQL (DSQL) statement for execution. Available in ***gpre***.

```
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;
```

RELEASE SAVEPOINT

```
RELEASE SAVEPOINT <savepoint_name>
```

Description: Releasing a savepoint destroys savepoint named by the identifier without affecting any work that has been performed subsequent to its creation.

REVOKE

Withdraws privileges from users for specified database objects. Available in ***-either_case***, ***DSQL***, and ***isql***.

```
REVOKE [GRANT OPTION FOR] privilege ON [TABLE] {tablename | viewname}
FROM {object | userlist | rolelist | GROUP UNIX_group}
| EXECUTE ON PROCEDURE procname FROM {object | userlist}
| role_granted FROM {PUBLIC | role_grantee_list}};
privileges = ALL [PRIVILEGES] | privilege_list
privilege_list = {
SELECT
| DELETE
| INSERT
| ENCRYPT ON ENCRYPTION
| DECRYPT
| UPDATE [(col [, col ...])]
| REFERENCES [(col [, col ...])]
}[, privilege_list ...]
object = {
   PROCEDURE procname
| TRIGGER trigname
| VIEW viewname
| PUBLIC
}[, object ...]
userlist = [USER] username [, [USER] username ...]
rolelist = rolename [, rolename ...]
role_granted = rolename [, rolename ...]
role_grantee_list = [USER] username [, [USER] username ...]
```

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
<***privilege_list***>	Name of privilege to be granted; valid options are ***SELECT***, ***DELETE***, ***INSERT***, ***ENCRYPT ON ENCRYPTION***, ***DECRYPT***, ***UPDATE***, and ***REFERENCES***.
GRANT OPTION FOR	Removes grant authority for privileges listed in the REVOKE statement from <userlist>; cannot be used with <object>.
<***col***>	Column for which the privilege is revoked.
<***tablename***>	Name of an existing table for which privileges are revoked.
<***viewname***>	Name of an existing view for which privileges are revoked.
GROUP <***unix_group***>	On a ***UNIX*** system, the name of a group defined in /etc/group.
<***object***>	Name of an existing database object from which privileges are to be revoked.
<***userlist***>	A list of users from whom privileges are to be revoked.
<***rolename***>	An existing role created with the ***CREATE ROLE*** statement.

Argument	Description
<role_grantee_list>	A list of users to whom <rolename> is granted; users must be in the InterBase security database (admin.ib by default).

Description: ***REVOKE*** removes privileges from users or other database objects. Privileges are operations for which a user has authority.

The following table lists SQL privileges:

SQL privileges	
Privilege	**Removes a user's privilege to ...**
ALL	Perform ***SELECT***, ***DELETE***, ***INSERT***, ***UPDATE***, ***REFERENCES***, and ***EXECUTE***.
SELECT	Retrieve rows from a table or view.
DELETE	Remove rows from a table or view.
DECRYPT	After encrypting a column, the database owner or the individual table owner can grant decrypt permission to users who need to access the values in an encrypted column.
ENCRYPT ON ENCRYPTION	Enables the database owner or individual table owner to use a specific encryption key to encrypt a database or column. Only the SYSDSO (Data Security Owner) can grant encrypt permission.
INSERT	Store new rows in a table or view.
UPDATE	Change the current value in one or more columns in a table or view; can be restricted to a specified subset of columns.
REFERENCES	Reference the specified columns with a foreign key; at a minimum, this must be granted to all the columns of the primary key if it is granted at all.
EXECUTE	Execute a stored procedure.

GRANT OPTION FOR revokes a user 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 can 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.
- When a role is revoked from a user, all privileges that granted by that user to others because of authority gained from membership in the role are also revoked.
- For more information about the ***REVOKE ENCRYPT ON ENCRYPTION*** and ***REVOKE DECRYPT*** permissions, see "Encrypting Your Data" in the Data Definition Guide.

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;
```

ROLLBACK

Restores the database to its state prior to the start of the current transaction or savepoint. Available in *gpre*, *DSQL*, and *isql*.

```
ROLLBACK [TRANSACTION name] [TO SAVEPOINT <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].
TOSAVEPOINT <name>	Specifies the savepoint to roll back to.
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 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;
```

SAVEPOINT

```
SAVEPOINT <savepoint_name>
```

Description: A savepoint allows a transaction to be partially rolled back. Updates that are made after a named savepoint is established can be rolled back by issuing a ***ROLLBACK*** command of the following form:

```
ROLLBACK [TRANSACTION transaction_name] TO SAVEPOINT savepoint_name;
```

If no transaction name is specified, the default transaction is used.

A savepoint name can be any valid SQL identifier. Savepoint names must be unique within their atomic execution context. If you assign a name that is already in use, the existing savepoint is released and the name is applied to the current savepoint. An application, for example, is an execution context, as is each trigger and stored procedure. Thus, if you have an application with several triggers, you can have a savepoint named SV1 within the application and also within each trigger and stored procedure.

SELECT

Retrieves data from one or more tables. Available in ***gpre***, ***DSQL***, and ***isql***.

Syntax

```
SELECT [TRANSACTION transact] [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 [ALL] select_expr][PLAN <plan_expr>]
[ORDER BY <order_list>]
[ROWS VALUE [TO upper_value] [BY step_value][PERCENT][WITH TIES]]
[FOR UPDATE [OF col [, col ...]]];
val = {col [array_dim]
   | :variable | constant | expr
   | funct | udf ([val [, val ...]])
   | NULL | USER | RDB$DB_KEY | ? }
   [COLLATE collation] [AS alias]

array_dim = [[x:]y [, [x:]y ...]]

constant = num | 'string' | charsetname 'string'

funct = 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 data_type)
   | UPPER (val)
   | GEN_ID (generator, val)

tableref = <joined_table> | <table_primary>

joined_table = tableref join_type JOIN tableref
   ON search_condition | (joined_table)

join_type = [INNER] JOIN
```

```
   | {LEFT | RIGHT | FULL } [OUTER]}

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 {>= | <=} val
   | val [NOT] {= | < | >} val
   | {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 = {= | < | > | <= | >= | !< | !> | <> | !=}

table_primary = [{TABLE | VIEW | PROCEDURE} [[AS] alias]] | <derived_table>

derived_table = query_expression [AS] alias

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
<expr>	A valid SQL expression that results in a single value.
<select_one>	A ***SELECT*** on a single column that returns exactly one value.
<select_list>	A ***SELECT*** on a single column that returns zero or more rows.
<select_expr>	A ***SELECT*** on a list of values that returns zero or more rows.

Argument	Description
TRANSACTION transact	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.

Argument	Description
{|<val>[,<val> ...]}*	The asterisk (*) retrieves all columns for the specified tables. <val>[,<val> ...] retrieves a list of specified columns, values, and expressions.
***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, stored procedures or derived tables from which to retrieve data; list can include joins and joins can be nested.
<joined_table>	A table reference consisting of a ***JOIN***.
<join_type>	Type of join to perform. Default: ***INNER***.
<table_primary>	Name of an existing table, view, stored procedure or a derived table.
alias	Alternate name for a table, view, or column.
<derived_table>	A result set of a ***SELECT*** query that you can use in the ***FROM*** clause. See Derived Tables (SELECT FROM SELECT) for more information and examples.
***WHERE**<search_condition>*	• Specifies a condition that limits rows retrieved to a subset of all available rows. • A WHERE clause can contain its own ***SELECT*** statement, referred to as a subquery.
***GROUP BY** col [, col ...]*	Groups related rows based on common column values; used in conjunction with ***HAVING***.
***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 the grouped rows returned.
UNION[ALL]	• Combines the results of two or more ***SELECT*** statements to produce a single, dynamic table without duplicate rows. • The ***ALL*** option keeps identical rows separate instead of folding them together into one.
***PLAN** <plan_expr>*	Specifies the query plan that should be used by the query optimizer instead of one it would normally choose.
<plan_item>	Specifies a table and index method for a plan.
***ORDER BY**<order_list>*	Specifies columns to order, either by column name or ordinal number in the query, and the sort order (ASC or DESC) for the returned rows.
***ROWS**<value>* *[TO<upper_value>]* *[BY<step_value>]* *[PERCENT][WITH TIES]*	• <value> is the total number of rows to return if used by itself. • <value> is the starting row number to return if used with ***TO***. • <value> is the percent if used with ***PERCENT***. • <upper_value> is the last row or highest percent to return. • If <step_value>=<n>, returns every <n>th row, or <n> percent rows. • ***PERCENT*** causes all previous ***ROWS*** values to be interpreted as percents. • ***WITH TIES*** returns additional duplicate rows when the *last* value in the ordered sequence is the same as values in subsequent rows of the result set; must be used in conjunction with ***ORDER BY***.
FOR UPDATE	Specifies columns listed after the ***SELECT*** clause of a ***DECLARE CURSOR*** statement that can be updated using a ***WHERE CURRENT OF*** clause.

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.

NOTE

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.
- Return portions or sequential portions of a larger result set; useful for Web developers, among others.

All SELECT statements consist of two required clauses (SELECT, FROM), and possibly others (INTO, WHERE, GROUP BY, HAVING, UNION, PLAN, ORDER BY, ROWS).

For more information on how to use SELECT in isql, see the Operations Guide. For a complete explanation of SELECT and its clauses, see the Embedded SQL Guide.

Derived Tables (SELECT FROM SELECT)

A derived table is the result set of a SELECT query that you can use in the FROM clause. You may find it useful to think of a derived table as a view with statement-level scope. This allows you the expressive flexibility to use a view-like structure without defining a database schema view, or allows the user to obtain the same benefit in an ad-hoc query without requiring a database administer to create a view definition.

You can use derived tables in triggers and stored procedures as well as user applications, but you must have proper access privileges on the underlying base tables and views accessed by a derived table.

Dynamic SQL and ***isql*** support derived table syntax, Embedded SQL does not support derived table syntax. For further info on Derived Tables refer to SQL Derived Table Support

Examples With Derived Tables

1. The following simple example shows how you can use derived tables:

```
SELECT  elj.job_code,
        elj.job_title
FROM    ( SELECT  job_code,
                  job_title
          FROM    job
          WHERE   max_salary  <  50000 )  AS  elj;
```

The statement queries the EMPLOYEE table for entry-level jobs.

2. The following is a more complex statement using derived tables:

```
SELECT  emp.emp_no,
          emp.full_name,
          emp.job_code,
          job.job_grade,
          job.job_title
FROM     ( SELECT  emp_no,
                     full_name,
                     job_code,
                     job_grade,
                     job_country
             FROM     employee ) AS  emp,
          ( SELECT  job_code,
                     job_grade,
                     job_country,
                     job_title
             FROM     job ) AS  job
WHERE    ( emp.job_code  =  job.job_code ) AND
          ( emp.job_grade  =  job.job_grade ) AND
          ( emp.job_country  =  job.job_country ) AND
          ( emp.job_country  =  'USA' );
```

3. The following example shows a derived table with a subquery:

```
SELECT  eid,
          ename
FROM     ( SELECT  e.emp_no,
                     e.full_name
             FROM     employee  e
             WHERE    e.job_country  =
                     ( SELECT  e1.job_country
                        FROM     employee  e1
                        WHERE    emp_no  =  144 ) ) AS  emp  (eid,  ename);
```

Additional Notes on Derived Tables

- Derived tables can be nested.
- Derived tables can be unions and can be used in unions. They can contain aggregate functions, subselects and joins, and can themselves be used in aggregate functions, subselects and joins. They can also be or contain queries on selectable stored procedures.

Additional Notes on SELECT

- When declaring arrays, you must include the outermost brackets, shown below in bold. For example, the following statement creates a 5 by 5 two-dimensional array of strings, each of which is 6 characters long:

```
my_array  =  VARCHAR(6)[5,5]
```

- Use the colon (:) to specify an array with a starting point other than 1. The following example creates an array of INTEGER that begins at 10 and ends at 20:

```
my_array = INTEGER[20:30]
```

- In SQL and ***isql***, you cannot use val as a parameter placeholder (like ?).
- In DSQL and ***isql***, val cannot be a variable.
- You cannot specify a COLLATE clause for Blob columns.
- You cannot specify a GROUP BY clause for Blob and array columns.

Examples

1. The following ***isql*** statement selects columns from a table:

```
SELECT job_grade,
       job_code,
       job_country,
       max_salary
FROM   project;
```

2. The next ***isql*** statement uses the * wildcard to select all columns and rows from a table:

```
SELECT *
FROM   countries;
```

3. 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;
```

4. 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;
```

5. 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;
```

6. 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';
```

7. 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;
```

8. 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);
```

9. The next example queries two of the system tables, RDB$CHARACTER_SETS and RDB$COLLATIONS to display all the available character sets, their ID numbers, number of bytes per character, and collations. Note the use of ordinal column numbers in the ORDER BY clause.

```
SELECT  rdb$character_set_name,
          rdb$character_set_id,
          rdb$bytes_per_character,
          rdb$collation_name
FROM     rdb$character_sets
          JOIN  rdb$collations
             ON  rdb$character_sets.rdb$character_set_id  =  rdb$collations.rdb$character_set_iefd
ORDER    BY  1,4;
```

10. The following examples reward the best performing sales people and terminate the least performing members of the sales team. The examples show how a Web developer, for example, could split the result set in half for display purposes.

```
SELECT     salesman,
             sales_dollars,
             sales_region
```

```
FROM        salespeople
ORDER BY sales_dollars DESC
ROWS 1 TO 50;
SELECT     salesman,
             sales_dollars,
             sales_region
FROM        salespeople
ORDER BY sales_dollars DESC
ROWS 50 TO 100 WITH ties;
```

11. Reward the best 100 performing salesmen with a 15 percent bonus:

```
UPDATE salespeople
SET         sales_bonus = 0.15 * sales_dollars
ORDER BY sales_dollars DESC
ROWS 100 WITH ties;
```

12. Eliminate the worst five percent of the sales force:

```
DELETE
FROM        salespeople
ORDER BY sales_dollars
ROWS 5 percent WITH ties;
```

SET DATABASE

Declares a database handle for database access. Available in ***gpre***.

```
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 • Must be unique within the program. • 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 at run time and compile time.

Argument	Description
'<dbname>'	Location and path name of the database associated with <dbhandle>; platform-specific.
RUNTIME	Specifies a database to use at run time if different than the one specified for use during preprocessing.
<var>	Host-language variable containing a database specification, user name, or password.
USER *'<name>'*	A valid user name on the server where the database resides • Used with ***PASSWORD*** to gain database access on the server. • Required for PC client attachments, optional for all others.
PASSWORD*'<string>'*	A valid password on the server where the database resides • Used with ***USER*** to gain database access on the server. • Required for PC client attachments, optional for all others.

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.ib';
```

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.ib' RUNTIME :db_name;
```

SET GENERATOR

Sets a new value for an existing generator. Available in ***gpre***, DSQL, and ***isql***.

```
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^{63} to $2^{63} - 1$

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. Any value that can be stored in a ***DECIMAL(18,0)*** can be specified as the value in a ***SET GENERATOR*** statement.

Generators return a 64-bit value, and wrap around only after 2^{64} invocations (assuming an increment of 1). Use an ISC-INT64 variable to hold the value returned by a generator.

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.

SET NAMES

Specifies an active character set to use for subsequent database attachments. Available in ***gpre***, and ***isql***.

```
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 are affected.

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 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.ib';
EXEC SQL
CONNECT;
```

The next statements demonstrate the use of *SET NAMES* in *isql*:

```
SET NAMES LATIN1;
```

```
CONNECT 'employee.ib';
```

SET SQL DIALECT

Declares the SQL Dialect for database access. Available in ***gpre*** and ***isql.***

```
SET SQL DIALECT n;
```

Argument	Description
<n>	The SQL Dialect type, either 1, 2, or 3

Description: ***SET SQL DIALECT*** declares the SQL Dialect for database access.

n is the SQL Dialect type 1, 2, or 3. If no dialect is specified, the default dialect is set to that of the specified compile-time database. If the default dialect is different than the one specified by the user, a warning is generated and the default dialect is set to the user-specified value.

Set SQL Dialects for Database Access	
SQL Dialect	**Used for**
1	InterBase 5 and earlier compatibility.
2	Transitional dialect used to flag changes when migrating from dialect 1 to dialect 3.
3	Current InterBase; allows you to use delimited identifiers, exact numerics, and DATE, TIME, and TIMESTAMP data types.

Examples: The following embedded SQL statement sets the SQL Dialect to 3:

```
EXEC SQL
SET SQL DIALECT 3;
```

SET STATISTICS

Recomputes the selectivity of a specified index. Available in ***gpre***, ***DSQL***, and ***isql***.

```
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;
```

SET SUBSCRIPTION

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

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

Argument	Description
<subscription_name>	Implied by the user identity of the database
<user_name>	User identify of the database connection

Description: The following example activates two subscriptions and returns changed data sets from the subscribed tables.

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

Example: ***SET SUBSCRIPTION*** "Employee_Changes", "Customer_Deletes" AT 'smartphone_123'

```
ACTIVE;
SELECT NAME, DEPARTMENT, SALARY FROM "Employees";
SELECT * FROM "Customers";
COMMIT or COMMIT RETAIN;
SET SUBSCRIPTION "Employee_Changes", "Customer_Deletes" AT 'smartphone_123'
INACTIVE;
```

SET TRANSACTION

Starts a transaction and optionally specifies its behavior. Available in ESQL ***(GPRE),*** DSQL, and ***ISQL.***

```
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 ...]]
[[NO] SAVEPOINT];
reserving_clause = TABLE [, TABLE ...]
[FOR [SHARED | PROTECTED] {READ | WRITE}] [, reserving_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.

Argument	Description
NAME <transaction>	Specifies the name for this transaction. • <transaction> is a previously declared and initialized host-language variable. • SQL only.
READ WRITE	[Default] Specifies that the transaction can read and write to tables.
READ ONLY	Specifies that the transaction can only read tables.
WAIT	[Default] Specifies that a transaction wait for access if it encounters a lock conflict with another transaction.
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.
NO SAVEPOINT	If ***NO SAVEPOINT*** is mentioned, the transaction is executed without starting an implicit savepoint for any SQL statements that execute within the context of that transaction. By default, InterBase starts an implicit savepoint to guarantee the atomicity of an SQL statement. For more information, see Chapter 5, "Working with Transactions" section on "Working with the ***NO SAVEPOINT*** Option" in the API Guide.

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 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;
```

Exclusive Isolation Level

Introduction

A Tool performing online reorganization of tables may need temporary exclusive table access to perform its functions. Transactions use exclusive table access to acquire an exclusive lock on a target table, and they are the only ones able to execute ***SELECT***, ***INSERT***, ***UPDATE***, and ***DELETE*** on a table. When a transaction acquires an exclusive lock, other transactions with lock requests must wait until the lock is released or downgraded to a compatible level. Transactions that maintain exclusive table access can modify data on a table without interference from other transactions. This isolation level is different from ***TABLE STABILITY*** and ***PROTECTED*** access because it does not allow other transactions to select from the table.

Usage

Use the ***SET TRANSACTION*** statement to specify the ***TABLE EXCLUSIVITY*** clause, or use the existing ***RESERVING*** clause to request exclusive access to one or more tables. ***TABLE EXCLUSIVITY*** acquires exclusive access to every table that a transaction accesses during statement execution. The RESERVING clause acquires exclusive access to a list of tables at transaction startup. To use the ***RESERVING*** clause, specify F***OR <table_list> EXCLUSIVE [READ | WRITE]***. Note that there is no difference between ***READ*** and ***WRITE*** because both modes do not allow other transactions to access the table. As with ***TABLE STABILITY***, there is an increased likelihood of lock conflicts and waits when this isolation level is used. In addition to ***isc_tpb_shared*** and ***isc_tpb_protected,*** you can use ***isc_tpb_exclusive*** in a transaction parameter block (TPB) to specify exclusive table access when calling ***isc_start_transaction()*** at the API level.

Requirements and Constraints

- It is possible to acquire exclusive table access even if one or more statements or requests that access the table have been prepared.
- It is possible to acquire exclusive table access even if one or more statements or requests that access the table have been executed as long as they have not yet accessed the table.

Migration issues

Prior to InterBase 2017, ***isc_tpb_exclusive*** could be used, but it allowed select access by concurrent transactions. Under InterBase 2017, a transaction has to wait until those readers terminate and subsequent readers block until the transaction with exclusive access terminates or downgrades the exclusive lock.

ALTER TABLE ... ALTER COLUMN for encryption and ***TRUNCATE TABLE*** acquire exclusive table access to perform their function.

InterBase 2017 introduces the InterBase-specific SQL reserved keywords ***EXCLUSIVITY*** and ***EXCLUSIVE***.

Wait time

Introduction

To acquire lockable resources, InterBase transaction lock can wait indefinitely, wait an specified period of time, or do not wait and return an error immediately. When a transaction holds a lock on a resource at a level incompatible with the requested lock level, this resource is inaccessible to other transactions. lockable resources can be tables, rows, or transaction entities.

Usage

This is the SQL syntax to specify a lock resolution mode:

```
SET TRANSACTION {[NO] WAIT};
```

WAIT implies wait indefinitely until a resource lock is acquired.

To specify a wait period use an optional WAIT clause in seconds. This is the time a transaction waits for a lock on a resource:

```
SET TRANSACTION WAIT [<number> [SECONDS]];
```

An ***isc_lock_timeout*** error code returns if the lock on the resource cannot be acquired during the wait period.

For example, consider attempting to Truncate Table. Table truncate attempts to acquire an exclusive lock on the target table and referencing tables that have a foreign key constraint on the target table. It is desirable to specify a wait time for the transaction if other transactions are using the table actively.

```
SQL> set transaction wait 10 seconds;
SQL> truncate table salary_history;
Statement failed, SQLCODE = -901

lock time-out on wait transaction
-unsuccessful metadata update
-object SALARY_HISTORY is in use
SQL>
```

There is a new transaction parameter block (TPB) parameter called ***isc_tpb_wait_time*** for use with InterBase transaction APIs: ***isc_start_transaction()***, ***isc_reconnect_transaction()***, and ***isc_start_multiple()***. It is followed by the literal "4" denoting a byte count and four bytes in little endian format denoting the wait period in seconds. Here are two examples specifying a 30 second and 300 second (5 minute) wait period, respectively:

```
isc_tpb_wait_time, 4, 30, 0, 0, 0
isc_tpb_wait_time, 4, 44, 1, 0, 0
```

There is an InterClient/JDBC extension API method for class ***interbase.interclient.Connection: setLockResolution(int mode, int waitTime)*** The existing method ***setLockResolution(int mode)*** is equivalent to ***setLockResolution(int mode, 0)***.

```
/* Set transaction timeout to 1 minute */

Driver driver = interbase.interclient.Driver();
Connection connection = driver.connect(url, properties);
(interbase.interclient.Connection connection).setLockResolution(LOCK_RESOLUTION_WAIT, 60);
```

It is expected that FireDAC, IBX and ODBC frameworks will provide low-level integrated support for the feature.

Requirements and Constraints

- The WAIT period is a positive integer between 1 and 32,767, inclusive. This is the equivalent of about 9 hours.
- Underlying remote and local protocols pass a 32-bit integer so that this limit can be increased without modifying the protocols.
- The feature is available through Dynamic SQL but not Static (Embedded) SQL.
- The feature is available through InterClient/JDBC API.
- The feature may not be visible as a transaction property by FireDAC, IBX or ODBC frameworks, but should be available as pass-through DSQL.

Migration issues

- The WAIT optional clause is not recognized by SQL parsers in InterBase versions older than 2017.
- The ***isc_tpb_wait_time*** TPB parameter is not recognized at the API level by InterBase versions older than 2017.

SHOW SQL DIALECT

Returns the current client SQL Dialect setting and the database SQL Dialect value. Available in ***gpre*** and ***isql.***

```
SHOW SQL DIALECT;
```

Description: ***SHOW SQL DIALECT*** returns the current client SQL Dialect setting and the database SQL Dialect value, either 1, 2, or 3.

SQL Dialect	Used for
1	InterBase 5 and earlier compatibility
2	Transitional dialect used to flag changes when migrating from dialect 1 to dialect 3.
3	Current InterBase; allows you to use delimited identifiers, exact numerics, and DATE, TIME, and TIMESTAMP data types.

Examples: The following embedded SQL statement returns the SQL Dialect:

```
EXEC SQL
SHOW SQL DIALECT;
```

SHOW SUBSCRIPTION

Syntax

```
SHOW {SUBSCRIPTION [<subscription_name>] | SUBSCRIPTIONS};
```

Argument	Description
<subscription_name>	The name of the subscription that you want to display.

Description

To display a list of all subscriptions, use the ***SHOW SUPSCRIPTIONS*** command. If you only want to display one supscription, use the ***SHOW SUPSCRIPTION*** <subscription_name> command.

Example

```
SHOW SUBSCRIPTIONS;

Subscription Name
============================================================================
SUB_CUSTOMER_DELETES
SUB_EMPLOYEE_CHANGES
SUB_VARIOUS_CHANGES

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

SHOW SUBSCRIPTION sub_customer_deletes;
Subscription name: SUB_CUSTOMER_DELETES

Owner: SYSDBA
Description: Subscribe TO deletes IN CUSTOMER TABLE
             CUSTOMER FOR ROW (DELETE)

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

SUM( )

Totals the numeric values in a specified column. Available in ***gpre***, ***DSQL***, and ***isql***.

```
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;
```

TRUNCATE TABLE

Introduction

InterBase 2017 introduces the SQL reserved keyword *TRUNCATE*. The Truncate Table command allows users and applications to empty the contents of a database table. This feature is useful for tables where rows require frequent deletion. The Truncate Table command performs faster, requires less I/O, and journals and archives much less information than an equivalent *DELETE FROM* table command. ETL applications or other applications can benefit from the combination of *TRUNCATE TABLE* with the NO *RESERVE SPACE* table allocation option when they stage large amounts of data that are deleted after use or moved to a more permanent location such as a history table.

Requirements and Constraints

1. The Truncate Table command obtains exclusive and protected write locks, this can have a visible effect on concurrent transactions that try to access tables being truncated. Although the table is being truncated anyways, all layers of the dependent table tree hierarchy in a *TRUNCATE CASCADE* are locked, and in a *TRUNCATE DEFERRED* these locks are held until the transaction terminates.
2. As a consequence of the previous point, users may run Truncate Table commands using a transaction with *NO WAIT* or a *WAIT TIME* limit.This way the transaction could rollback the operation if a timeout occurs or retry a limited number of times.
3. It is not possible to truncate system tables, temporary tables, and views. For optimization and performance reasons the engine truncates physically some of these tables types, users don't have access to this functionality. However, users might perceive better performance.
4. External tables can be truncated.
5. It's not possible to track who executed a Truncate Table command. *FOR EACH STATEMENT* triggers that enable users to write a triggered action for a Truncate Table command are not supported in InterBase.

How it works

Truncate Table operates at table level rather than at row level, it acts on the stored data inside a table instead of the metadata. Truncate Table deletes all the rows of a table in similar way to a ***DELETE FROM <table>*** command, but it doesn't perform row level actions like DELETE triggers, check constrains, and index maintenance. Truncate Table is usually faster than row-level deletion.

The Truncate Table command is not under transaction control. When you empty a table, it is not possible to undo the action even if you roll back the transaction that executed it. Only a point-in-time recovery can recover the data from InterBase journal archives. Truncated tables don't have storage allocated for row data, indexes, or blobs.

The Truncate Table command is sensitive to other tables' foreign key constraints that reference the table being truncated. In it's simplest form, foreign key constrains disallow table truncation. InterBase Truncate Table provides several non-SQL and run-time extensions to override this restriction. This enables a more liberal interpretation of the command enable execution in situations that do not compromise existing foreign key constraints. Although Truncate Table is not under transaction control, it is possible to make it behave as if it were by deferring its execution until after the effects of the transaction in which it is contained have been committed or rolled back.

Truncate Table syntax

```
TRUNCATE TABLE <table_name> [IMMEDIATE | DEFERRED] [RESTRICT | CASCADE]
```

When using the truncate Table command, please consider these points:

- ***IMMEDIATE*** is implicit if neither ***IMMEDIATE*** nor ***DEFERRED*** are specified. ***IMMEDIATE*** and ***DEFERRED*** are unreserved keywords.
- ***RESTRICT*** is implicit if neither ***RESTRICT*** nor ***CASCADE*** are specified.

For example:

```
TRUNCATE TABLE <table_name>
```

is the same as:

```
TRUNCATE TABLE <table_name> IMMEDIATE RESTRICT
```

- Use the ***IMMEDIATE*** qualifier to execute the Truncate Table command immediately and to empty the content of the table.
- Use the ***DEFERRED*** qualifier to execute the Truncate Table command when the transaction terminates with ***COMMIT*** or ***ROLLBACK***. ***COMMIT*** guarantees all the transactional work before emptying the target table. ***ROLLBACK*** cancels the Truncate Table Command.
- When you specify the ***RESTRICT*** qualifier the Truncate Table command only succeeds if no foreign key constrains reference the target table. The Truncate Table command only executes if the table has self-referencing foreign key constrains.
- When you specify the ***CASCADE*** qualifier, declare all the foreign key constrains of referencing tables with the ***ON CASCADE DELETE*** action, or the foreign key constrains not declared must reference currently "empty" tables. This condition applies recursively to referencing tables, if any table violates this condition the Truncate Table command fails with a foreign key violation error.

NOTE

In this context "empty" means the table has no data storage allocated to it. A table with no rows still has storage allocated to it. This can happen when all rows have been deleted with one or more ***DELETE*** statements, but concurrent transactions still have earlier versions of the row in their snapshots, or the rows and their earlier versions are not in any transactions' snapshots but have not yet been garbage collected. To immediately make those foreign key dependent tables empty, Truncate Table can be run against such tables if logic dictates. A Truncate Table statement is allowed to be called from InterBase triggers and stored procedures assuming they have been granted the TRUNCATE privilege.

Truncate Table privilege

Execution of a Truncate Table command requires a ***TRUNCATE*** privilege. By default, this privilege is granted only to the table owner and SYSDBA initially. The TRUNCATE privilege must be specifically granted to any other authorization identifier as it is not considered a member of ***ALL*** privileges.

```
{GRANT | REVOKE} TRUNCATE ON <table_name> {TO | FROM} <grantee> [WITH GRANT OPTION]
```

NOTE

The ***TRUNCATE*** privilege is not required on referencing tables with non-***ON CASCADE DELETE*** foreign key constraints when checking if those tables are empty. [a]

[a] In this context "empty" means the table has no data storage allocated to it. A table with no rows still has storage allocated to it.

Truncate Table operation

The Truncate Table command is executed in two phases:

1. A locking phase.
2. A truncation phase.

Upon command, the returned target tables are always locked for exclusive access. If the ***DEFERRED*** qualifier is specified, the truncation phase of the operation does not occur until transaction ***COMMIT***. Specifically, foreign dependent tables with non-***ON CASCADE DELETE*** reference constraints are only locked for protected write.

Because these tables are not being physically dismantled, reads can be allowed without blocking on the empty table. The protected write lock prevents insertion of new rows that might have a valid reference on a table with an imminent truncation.

If the ***CASCADE*** qualifier is specified, then the target table is locked as well as referencing tables with foreign key constraints that depend on the target table. The locking protocol works in a top-down fashion, locking the target table first followed by the referencing tables and recursively applied to those referencing tables with ***ON CASCADE DELETE*** foreign key constraints. This is referred to as a dependent table tree hierarchy.

The truncation protocol works in a bottom-up fashion. First, it truncates foreign dependent tables, this prevents dangling foreign key references if the total execution fails unexpectedly before completion. During this phase, all table data, index and blob storage is released back to the database for reuse. Once the tables have been truncated, the table locks are downgraded to the level they would have acquired for normal write access. For a consistency mode transaction this is protected write. For a concurrency mode transaction this is shared write.

Truncate Table errors

A lock error returns if an exclusive table lock cannot be acquired during the locking phase. The error returned can be a ***isc_deadlock*** error or a transaction wait error depending on the transaction's wait mode. If a transaction waits indefinitely for lock acquisition, it can only return a ***isc_deadlock*** error due to a real deadlock with a concurrent transaction.

If the transaction is ***NO WAIT***, it returns an ***isc_lock_conflict*** error immediately. If the transaction requests a ***WAIT TIME***, it returns ***isc_lock_timeout*** when waiting the specified time for table lock acquisition.

It is also an error to execute a Truncate Table command from a ***READ_ONLY*** transaction or database. During the truncation phase there is no expected way for an error to occur. However, unexpected errors can occur due to extraneous circumstances.

If a transaction executing a Truncate Table command has open cursors on one or more of the truncated tables, attempting to perform an ***UPDATE*** on those open cursors can return an ***isc_table_truncated.*** Otherwise, if the fetch from the cursor is only for retrieval purposes, the fetch operation returns as if there were no more remaining rows to fetch.

Truncate Table effect on Change Views

When a client database connection activates subscriptions containing one or more truncated tables, the client receives two indications of the underlying truncate activity.

First, when a cursor opens (the ***SELECT*** operation is executed), a warning status vector indicating ***isc_table_truncated*** returns with the name of the truncated table. A warning status vector can chain together five separate ***isc_table_truncated*** status codes of truncated tables in a ***SELECT*** statement. Clients can use this form of table notification to truncate one or more corresponding tables on the client. For example, after executing the query:

```
if (isc_dsql_execute(status_vector, ...) == 0)      /* after successful execution check for warnings */
      {
      if (status_vector[2] == isc_arg_warning)

// A warning status vector for one or more truncated tables shall have the following format.

status_vector[0] = isc_arg_gds
status_vector[1] = 0
status_vector[2] = isc_arg_warning

// The following sequence can be repeated up to 5 times

status_vector[3] = isc_table_truncated
status_vector[4] = isc_arg_string
status_vector[5] = name of table truncated

// status_vector terminator

status_vector[last element] = isc_arg_end
```

Second, on every fetch from the cursor, a ***SQLIND_TRUNCATE*** flag is set in the SQL indicator member of a SQLVAR element for a column of a truncated table. Clients can use this form of column notification to

delete one or more rows in corresponding tables before using the other SQLIND flags to decide on the appropriate row modification operation.

```
/* Bit flag definitions for SQLVAR.sqlind output variable */

#define SQLIND_NULL              (short) (1 << 15)
#define SQLIND_INSERT            (1 << 0)
#define SQLIND_UPDATE            (1 << 1)
#define SQLIND_DELETE            (1 << 2)
#define SQLIND_CHANGE            (1 << 3)
#define SQLIND_TRUNCATE          (1 << 4)
#define SQLIND_CHANGE_VIEW       (1 << 5)
```

If the query returns no rows because there were no changes to the subscribed tables subsequent to table truncation, then only the first method can be used. The second method will not work since the SQLDA/SQLVAR element will not be populated because no rows have been returned.

Higher level database frameworks such as FireDAC may surface these truncate notifications with supporting APIs (e.g., ***isTruncated()***.)

Truncate Table examples

Consider a lottery drawing example:

```
TRUNCATE TABLE PENDING_LOTTERY_TICKETS DEFERRED;
INSERT INTO CURRENT_LOTTERY_DRAWING ... SELECT FROM PENDING_LOTTERY TICKETS;
COMMIT;
```

The day of the lottery drawing at 9:00 PM the ***TRUNCATE TABLE*** command is executed with a ***DEFERRED*** status. Because the Truncate Table command obtains an exclusive lock, any attempts to insert new lottery tickets at 9:00 PM have to wait. The ***CURRENT_LOTTERY_DRAWING*** table is then populated with ***PENDING_LOTTERY_TICKETS***. The ***PENDING_LOTTERY_TICKETS*** table is truncated only after a successful COMMIT, this ensures the tickets are not lost before moving them for the current lottery drawing. Once truncation completes, the ***PENDING_LOTTERY_TICKETS*** exclusive lock is released, allowing pending lottery ticket ***INSERT*** commands to complete and be eligible for the next lottery drawing.

Conversely, a bulk load operation would want to ensure a table is immediately emptied before the load:

```
TRUNCATE TABLE CUSTOMER_ORDERS;  /* IMMEDIATE is implied */
EXECUTE LOAD_CUSTOMER_ORDERS;
COMMIT;
```

A set of tables may form a composition hierarchy to represent the semantic notion of containment:

```
INVOICE_HEADER <-- INVOICE_DETAILS <-- {RAIN_CHECK_TICKET, DROP_SHIP_ADDRESS}
```

The dependent tables are all declared with ***ON CASCADE DELETE*** foreign key constraints. All the invoices can be quickly dropped by executing:

```
TRUNCATE TABLE INVOICE_HEADERS CASCADE;
```

On the other hand, there may exist a lookup table of two-letter US State postal codes named ***POSTAL_CODES*** that every document in an organization depends on. None of these dependent tables register an ***ON CASCADE DELETE*** foreign key constraint with the lookup table.

POSTAL CODE	STATE
CA	California
MA	Massachusetts
NC	North Carolina
TX	Texas
...	

```
TRUNCATE TABLE POSTAL_CODES CASCADE;
```

Assuming that one or more of the foreign dependent tables are not empty, this command fails with a ***FOREIGN KEY CONSTRAINT*** violation error. The foreign dependent tables are not ***ON CASCADE DELETE*** and have storage allocated for their existing rows.

Truncate Table Tutorial

This section guides you in the use of the Truncate Table command and its qualifiers.

Creating a test database and tables

1. Create a Database.

```
CREATE DATABASE "truncate.ib";
COMIT;
```

2. Create a table named 'SOLO' that has no references from any other table.

```
CREATE TABLE SOLO (F1 INTEGER);
INSERT INTO SOLO VALUES (1);
COMMIT;
```

3. Create a table named 'SOLO_SELF_REF' and populate it with data, this table references itself.

```
CREATE TABLE SOLO_SELF_REF (EMP_NO INTEGER NOT NULL, MNGR_NO INTEGER,
      PRIMARY KEY (EMP_NO));
ALTER TABLE SOLO_SELF_REF ADD FOREIGN KEY (MNGR_NO) REFERENCES SOLO_SELF_REF
(EMP_NO);
INSERT INTO SOLO_SELF_REF VALUES (1, 1);
INSERT INTO SOLO_SELF_REF VALUES (2, 1);
INSERT INTO SOLO_SELF_REF VALUES (3, 2);
INSERT INTO SOLO_SELF_REF VALUES (4, 2);
COMMIT;
```

4. Next, create the primary table called PT, and add a primary key on EMP_NO.

```
CREATE TABLE PT (EMP_NO INTEGER NOT NULL, SSN_NO INTEGER NOT NULL);
ALTER TABLE PT ADD PRIMARY KEY (EMP_NO);
INSERT INTO PT VALUES (1, 100);
INSERT INTO PT VALUES (2, 200);
INSERT INTO PT VALUES (3, 300);
INSERT INTO PT VALUES (4, 400);
COMMIT;
```

5. Create a table named "FT1" and add a foreign key reference, this references to PT with ***ON DELETE CASCADE***.

```
CREATE TABLE FT1 (MNGR_NO INTEGER NOT NULL, EMP_COUNT INTEGER, PRIMARY KEY
(MNGR_NO));
ALTER TABLE FT1 ADD FOREIGN KEY (MNGR_NO) REFERENCES PT (EMP_NO) ON DELETE
CASCADE;
INSERT INTO FT1 VALUES (1, 1);
INSERT INTO FT1 VALUES (2, 2);
COMMIT;
```

Truncate a table with no references from other tables

1. First check the number of records on each table.

```
SELECT COUNT(*) FROM SOLO;
SELECT COUNT(*) FROM SOLO_SELF_REF;
```

2. Next, truncate the SOLO table.

```
TRUNCATE TABLE SOLO;
COMMIT;
```

3. Next, truncate the SOLO_SELF_REF table with reference to self.

```
TRUNCATE TABLE SOLO_SELF_REF;
COMMIT;
```

4. Finally, check count of records on each table.

```
SELECT COUNT(*) FROM SOLO;
SELECT COUNT(*) FROM SOLO_SELF_REF;
```

Truncate a table with no references from other tables using the DEFERRED qualifier

1. Check count of records

```
SELECT COUNT(*) FROM SOLO_SELF_REF;
```

2. Truncate the SOLO_SELF_REF table with ***DEFERRED*** qualifier. We now have exclusive access to the table. No other requests allowed to read/write to the table.

```
TRUNCATE TABLE SOLO_SELF_REF DEFERRED;
```

3. We still have access to the table. Do new DML requests.

```
SELECT * FROM SOLO_SELF_REF;
INSERT INTO SOLO_SELF_REF VALUES (5, 2);
SELECT * FROM SOLO_SELF_REF;
```

4. Check count of records

```
SELECT COUNT(*) FROM SOLO_SELF_REF;
```

5. ***COMMIT*** will truncate now due to ***DEFERRED*** action

```
COMMIT;
```

6. Check count of records

```
SELECT COUNT(*) FROM SOLO_SELF_REF;
```

Truncate on a primary table cascades to table references with *ON CASCADE DELETE* definition

1. Check count of records

```
SELECT COUNT(*) FROM PT;
SELECT COUNT(*) FROM FT1;
```

2. ***TRUNCATE PT*** table with default ***RESTRICT*** qualifier

```
TRUNCATE TABLE PT;
COMMIT;
```

3. Check count of records

```
SELECT COUNT(*) FROM PT;
SELECT COUNT(*) FROM FT1;
```

4. TRUNCATE PT table with ***CASCADE*** qualifier

```
TRUNCATE TABLE PT CASCADE;
COMMIT;
```

5. Check count of records

```
SELECT COUNT(*) FROM PT;
SELECT COUNT(*) FROM FT1;
```

UPDATE

Changes the data in all or part of an existing row in a table, view, or active set of a cursor. Available in ***gpre***, ***DSQL***, and ***isql***.

SQL form:

```
UPDATE [TRANSACTION <transaction>] {TABLE | VIEW}
SET col = val [, col = val ...]
[WHERE search_condition | WHERE CURRENT OF cursor]
[ORDER BY order_list]
[ROWS VALUE [TO upper_value] [BY step_value][PERCENT][WITH TIES]];
```

DSQL and ***isql*** form:

```
UPDATE {TABLE | VIEW}
SET col = val [, col = val ...]
[WHERE search_condition
[ORDER BY order_list]
[ROWS VALUE [TO upper_value] [BY step_value][PERCENT][WITH TIES]]
val = {
col [array_dim]
| :variable
| constant
| expr
| FUNCTION
| udf ([val [, val ...]])
| NULL
| USER
| ?}
[COLLATE collation]
array_dim = [[x:]y [, [x:]y ...]]
constant = num | 'string' | charsetname 'string'
FUNCTION = CAST (val AS data_type)
| UPPER (val)
| GEN_ID (generator, val)
```

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

<search_condition> = See CREATE TABLE. for a full description.

Notes on the ***UPDATE*** statement:

- In SQL and ***isql***, you cannot use <val> as a parameter placeholder (like "?").
- In DSQL and ***isql***, <val> cannot be a variable.
- You cannot specify a ***COLLATE*** clause for Blob columns.

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 active set is to be modified • Not available in DSQL and ***isql***
ORDER BY <order_list>	Specifies columns to order, either by column name or ordinal number in the query, and the sort order (ASC or DESC) for the returned rows
ROWS <value> *[TO <upper_value>]* *[BY <step_value>]* *[PERCENT][WITH TIES]*	• <value> is the total number of rows to return if used by itself • <value> is the starting row number to return if used with ***TO*** • <value> is the percent if used with ***PERCENT*** • <upper_value> is the last row or highest percent to return • If <step_value> = <n>, returns every <n>th row, or <n> percent rows • ***PERCENT*** causes all previous ***ROWS*** values to be interpreted as percents • ***WITH TIES*** returns additional duplicate rows when the *last* value in the ordered sequence is the same as values in subsequent rows of the result set; must be used in conjunction with ***ORDER BY***

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.

IMPORTANT

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;
```

UPPER( )

Converts a string to all uppercase. Available in ***gpre***, ***DSQL***, and ***isql***.

```
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));
```

WHENEVER

Traps SQLCODE errors and warnings. Available in ***gpre***.

```
WHENEVER {NOT FOUND | SQLERROR | SQLWARNING}
{GOTO label | CONTINUE};
```

Argument	Description
NOT FOUND	Traps SQLCODE = 100, no qualifying rows found for the executed statement
SQLERROR	Traps SQLCODE < 0, failed statement
SQLWARNING	Traps SQLCODE > 0 AND < 100, system warning or informational message
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 a 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. */
```

For a complete discussion of error-handling methods and programming, see the Embedded SQL Guide.

RECONNECT

Reconnects to the latest successfully connected database. ***RECONNECT*** is only available in isql and in SQL scripts that you run in ***isql.***

Syntax

isql:

```
RECONNECT [USER <username>] [PASSWORD <password>] [ROLE <rolename>] [CACHE
<number>];
```

Argument	Description
USER <username>	String or host-language variable that specifies a user name for the database. The server checks the user name against the security database. User names are case-insensitive.
PASSWORD <password>	String or host-language variable, that specifies a password for the database. The server checks the password against the security database. Passwords are case-sensitive.
ROLE <rolename>	String or host-language variable up to 67 characters in size, that specifies the role that the user adopts for this connection to the database. The user can adopt at most one role per connection, and cannot switch roles (except by reconnecting).
CACHE <number>	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 <number>:

	• Default: 256 • Maximum value: system-dependent

Description

The ***RECONNECT*** statement connects to the last successfully connected database. All parameters for the ***RECONNECT*** statement are optional. If you do not specify a parameter, ***RECONNECT*** uses the value that you pass via Command-line Options.

Examples

```
RECONNECT;
RECONNECT USER 'sysdba' PASSWORD 'masterkey';
RECONNECT USER 'sysdba' PASSWORD 'masterkey' ROLE 'DBA';
```

Procedures and Triggers

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, 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.

Statement Types Not Supported

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

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

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.

```
<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);
```

BEGIN ... END

Defines a block of statements executed as one. Available in triggers and stored procedures.

```
<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 semicolon.

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.

```
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
;
```

Comment

Comment syntax allows programmers to add comments to procedure and trigger code or SQL scripts.

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

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

NOTE

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

```
-- comment text
```

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

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

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

For example:

- A multi-line bracketed comment:

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

- A multi-line simple comment:

```
-- my multi-line

-- comment is this

-- text
```

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

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

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

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

/* This script sets up Change Views Subscriptions
    on the EMPLOYEE table.
*/
CONNECT "emp.ib" USER 'SYSDBA' password 'masterkey';
COMMIT;
CREATE SUBSCRIPTION sub ON EMPLOYEE FOR ROW (INSERT, UPDATE, DELETE);
COMMIT;

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

- Simple comment followed by another SLC

```
-- One more comment

CREATE SUBSCRIPTION sub2 ON EMPLOYEE FOR ROW (INSERT);
COMMIT;
```

- Simple comment followed by another SLC with leading whitespace

```
    -- One more comment followed by leading whitespace before CREATE below

CREATE SUBSCRIPTION sub3 ON EMPLOYEE FOR ROW (INSERT, UPDATE, DELETE);
COMMIT;

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

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

```
COMMIT;

SET TERM ^;
```

- Create a stored procedure with inline comments

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

DECLARE VARIABLE

Declares a local variable. Available in triggers and stored procedures.

```
DECLARE VARIABLE var data_type;
```

Argument	Description
<var>	Name of the local variable, unique within the trigger or procedure
<data_type>	Data type of the local variable; can be any InterBase data type except 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
```

```
. . .
```

EXCEPTION

Raises the specified exception. Available in triggers and stored procedures.

```
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;
```

EXECUTE PROCEDURE

Executes a stored procedure. Available in triggers and stored procedures.

```
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 example illustrates a recursive procedure, ***FACTORIAL***, which calculates factorials. The procedure calls itself recursively to calculate the factorial of NUM, the input parameter.

```
CREATE PROCEDURE FACTORIAL (NUM INT)
RETURNS (N_FACTORIAL DOUBLE PRECISION)
AS
DECLARE VARIABLE NUM_LESS_ONE INT;
BEGIN
IF (NUM = 1) THEN
BEGIN /**** Base case: 1 factorial is 1 ****/
N_FACTORIAL = 1;
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;
```

EXECUTE STATEMENT

Embedding a variation of ***EXECUTE STATEMENTS*** within a Stored Procedure.

Description: Store procedure developers can now embed three variations of ***EXECUTE STATEMENT*** within their Stored Procedures. The variations depend on the number of rows returned from the ***EXECUTE STATEMENT*** command. The variations are: No rows or data returned, One row of data returned, and Any number of data rows returned.

No Rows or Data Returned

```
EXECUTE STATEMENT <statement>
```

Argument	Description
<statement>	A SQL statement returning no rows of data.

Examples:

```
CREATE PROCEDURE EXEC_STMT_NO_RET (proc_name VARCHAR(20))
AS
DECLARE VARIABLE EMPNO INTEGER;
DECLARE VARIABLE EXECSTMT VARCHAR(150);
BEGIN
SELECT MAX(EMP_NO) FROM EMPLOYEE INTO EMPNO;
EXECSTMT = 'EXECUTE PROCEDURE' || proc_name || '( ' || CAST (EMPNO AS VARCHAR(10)) || ')';
EXECUTE STATEMENT EXECSTMT;
END
```

One Row of Data Returned

```
EXECUTE STATEMENT <select-statement> INTO :<var> [, :<var> ..]
```

Argument	Description
<select-statement>	SQL statement returning one or no rows of data.
<var>	Valid procedure variable, the ":" is optional.

Example:

```
CREATE PROCEDURE EXEC_STMT_SINGLETON (TABLE_NAME VARCHAR(50))
AS
DECLARE VARIABLE MAXEMPNO INTEGER;
BEGIN
EXECUTE STATEMENT 'SELECT MAX(EMP_NO) FROM ' || TABLE_NAME INTO :MAXEMPNO;
END
```

Any Number of Data Rows Returned

```
FOR EXECUTE STATEMENT <select-statement> INTO :<var> [, :<var> ..]
DO <compound-statement>
```

Argument	Description
<select-statement>	SQL statement returning one or zero rows of data.
<var>	Valid procedure variable. The : is optional.

Example:

```
CREATE PROCEDURE EXEC_STMT_ANY (TABLE_NAME VARCHAR(50), INT_FIELD INTEGER)
RETURNS
 (INT_RETVAR INTEGER)
 AS
 DECLARE VARIABLE IFIELD INTEGER;
 BEGIN
  FOR EXECUTE STATEMENT
  'SELECT ' || INT_FIELD || ' FROM ' || TABLE_NAME INTO :IFIELD
  DO
    IF (IFIELD = 0) THEN
      INT_RETVAR = 0;
    ELSE
      INT_RETVAR = INT_RETVAR + IFIELD;
  SUSPEND;
  END
```

Requirements and Constraints

There are constrains and peculiarities with using ***EXECUTE STATEMENT***:

- Starting with InterBase XE7 Update 1, there is a new requirement on ***FOR EXECUTE STATEMENT*** to match every item in the ***SELECT*** list with a corresponding item in the ***INTO*** list.
- The Statement is "prepared" every time it is executed, which affects the performance of the Stored Procedure.
- No checks are done on the statement when the procedure is created; dependency checks are not done when the procedure is created, also the checks for existence of tables or column names referred in the execute statement are not performed. All these checks are done at execute time and results in errors if an error condition occurs.
- The feature can be used to perform DDL operations.
- All statements are executed based on the privileges of the user executing the Stored Procedure.
- SQL statements, "***COMMIT***:", "***COMMIT RETAIN***", "***ROLLBACK***", "***ROLLBACK RETAIN***", and "***CREATE DATABASE***" are not supported with "***EXECUTE STATEMENT***". These statements return the code isc_exec_stmt_disallow error.

FOR SELECT...DO

Repeats a block or statement for each row retrieved by the ***SELECT*** statement. Available in triggers and stored procedures.

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
```

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.

```
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;
. . .
```

Input Parameters

Used to pass values from an application to a stored procedure. Available in stored procedures only.

```
CREATE PROCEDURE <|name> [(<param data_type> [, <param data_type ...>])]
```

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. However, arrays of data types are not supported.

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
. . .
```

NEW Context Variables

Indicates a new column value in an ***INSERT*** or ***UPDATE*** operation. Available only in triggers.

```
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 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.

```
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 ;
```

OLD Context Variables

Indicates a current column value in an ***UPDATE*** or ***DELETE*** operation. Available in triggers only.

```
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 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.

```
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 ;
```

Output Parameters

Used to return values from a stored procedure to the calling application. Available in stored procedures only.

```
CREATE PROCEDURE <name> [(<param DATA type> [, <param DATA TYPE ...>])]
[RETURNS (<param DATA type> [, <param DATA type> ...])]
```

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. Arrays of data types are not supported.

If output parameters are declared in the header of a procedure, 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 output parameters of the procedure. In an ***EXECUTE PROCEDURE*** statement, the output parameters need not match the names of the output parameters of the procedure, 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)
```

POST EVENT

Posts an event. Available in triggers and stored procedures.

```
POST_EVENT 'event_name' | <col | variable>;
```

Argument	Description
<event_name>	Name of the event being posted; must be enclosed in quotes
col	Name of a column whose value the posting will be based on
variable	Name of a string variable in the stored procedure or trigger

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;
```

The next statement posts an event based on a string variable previously declared:

```
myval = 'new_order:' || NEW.COMPANY;
POST_EVENT myval;
```

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.

```
<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;
```

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.

```
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***.

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 a SQLCODE error, GDSCODE error, or exception—the statements executed since the last ***SUSPEND*** are undone. Statements before the last ***SUSPEND*** are never undone, unless the transaction comprising the procedure is rolled back.

Example: The following procedure illustrates the use of ***SUSPEND*** and ***EXIT***:

```
CREATE PROCEDURE P RETURNS (R INTEGER)
AS
BEGIN
R = 0;
WHILE (R < 5) DO
BEGIN
R = R + 1;
SUSPEND;
IF (R = 3) THEN
EXIT;
END
END;
```

If this procedure is used as a select procedure 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.

WHEN ... DO

Error-handling statement that performs the statements following DO when the specified error occurs. Available in triggers and stored procedures.

```
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>	A SQLCODE error code number
GDSCODE <errcode>	An InterBase error code. Use Table 5.5 and strip isc_ before mentioning the error code with GDSCODE usage. For example: GDSCODE lock_conflict.
ANY	Keyword that handles any of the above types of errors.
<compound_statement>	Statement or block executed when any of the specified errors occur.

IMPORTANT

If used, ***WHEN*** must be the last statement in a ***BEGIN...END*** block. It should come after ***SUSPEND***, if present.

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.

```
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;
```

WHILE ... DO

Performs the statement or block following ***DO*** as long as the specified condition is ***TRUE***. Available in triggers and stored procedures.

```
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:

```
CREATE PROCEDURE SUM_INT (I INTEGER) RETURNS (S INTEGER)
AS
BEGIN
S = 0;
WHILE (I > 0) DO
BEGIN
S = S + I;
I = I - 1;
END
END;
```

If this procedure is called from ***isql*** with the command:

```
EXECUTE PROCEDURE SUM_INT 4;
```

then the results will be:

```
S
==========
10
```

Keywords

The table in this chapter lists keywords, words reserved from use in SQL programs and isql (Interactive SQL). The list includes ***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, unless it is enclosed in double quotes. Keywords are:

- Part of statements
- Used as statements
- Names of standard data structures or data types

InterBase Keywords

These keywords are reserved words in all dialects.

- Beginning with InterBase 6, you cannot create objects in a dialect 1 database that have any of these keywords as object names (identifiers).
- You can migrate a version 5 database that contains these keywords used as identifiers to version 6 or later dialect 1 without changing the object names: a column could be named "YEAR", for instance.
- Version 5 clients can access these keyword identifiers without error.
- Version 6 and later clients cannot access keywords that are used as identifiers. In a dialect 1 database, you must change the names so that they are not keywords.
- If you migrate directly to dialect 3, you can retain the names, but you must delimit them with double quotes. To retain accessibility for older clients, put the names in all upper case. Delimited identifiers are case sensitive.
- Although TIME is a reserved word in version 6 and later dialect 1, you cannot use it as a data type because such databases guarantee data type compatibility with version 5 clients.
- In dialect 3 databases and clients, any reserved word can be used as an identifier as long as it is delimited with double quotes.

A

ACTION	***ACTIVE***	***ADD***	***ADMIN***
AFTER	***ALL***	***ALTER***	***AND***
ANY	***AS***	***ASC***	***ASCENDING***
AT	***AUTO***	***AUTODDL***	***AVG***

B

BASED	***BASENAME***	***BASE_NAME***	***BEFORE***
BEGIN	***BETWEEN***	***BLOB***	***BLOBEDIT***
BOOLEAN	***BUFFER***	***BY***	

C

CACHE *CASCADE* *CASE* *CAST*
CHAR *CHARACTER* *CHARACTER_LENGTH* *CHAR_LENGTH*
CHECK *CHECK_POINT_LEN* *CHECK_POINT_LENGTH* *COALESCE*
COLLATE *COLLATION* *COLUMN* *COMMIT*
COMMITTED *COMPILETIME* *COMPUTED* *CLOSE*
CONDITIONAL *CONNECT* *CONSTRAINT* *CONTAINING*
CONTINUE *COUNT* *CREATE* *CSTRING*
CURRENT *CURRENT_DATE* *CURRENT_TIME* *CURRENT_TIMESTAMP*
CURSOR

D

DATABASE *DATE* *DAY* *DB_KEY*
DEBUG *DEC* *DECIMAL* *DECLARE*
DECRYPT *DEFAULT* *DELETE* *DESC*
DESCENDING *DESCRIBE* *DESCRIPTOR* *DISCONNECT*
DISPLAY *DISTINCT* *DO* *DOMAIN*
DOUBLE *DROP*

E

ECHO *EDIT* *ELSE* *ENCRYPT*
ENCRYPTION *END* *ENTRY_POINT* *ESCAPE*
EVENT *EXCEPTION* *EXECUTE* *EXISTS*
EXIT *EXTERN* *EXTERNAL* *EXTRACT*

F

FALSE *FETCH* *FILE* *FILTER*
FLOAT *FOR* *FOREIGN* *FOUND*
FREE_IT *FROM* *FULL* *FUNCTION*

G

GDSCODE *GENERATOR* *GEN_ID* *GLOBAL*
GOTO *GRANT* *GROUP* *GROUP_COMMIT_WAIT*
GROUP_COMMIT_WAIT_TIME

H

HAVING *HELP* *HOUR*

I

IF *IMMEDIATE* *IN* *INACTIVE*
INDEX *INDICATOR* *INIT* *INNER*
INPUT *INPUT_TYPE* *INSERT* *INT*
INTEGER *INTO* *IS* *ISOLATION*
ISQL

J

JOIN

K

KEY

L

LC_MESSAGES *LC_TYPE* *LEFT* *LENGTH*
LEV *LEVEL* *LIKE* *LOGFILE*
LOG_BUFFER_SIZE *LOG_BUF_SIZE* *LONG*

M

MANUAL *MAX* *MAXIMUM* *MAXIMUM_SEGMENT*
MAX_SEGMENT *MERGE* *MESSAGE* *MIN*
MINIMUM *MINUTE* *MODULE_NAME* *MONTH*

N

NAMES *NATIONAL* *NATURAL* *NCHAR*
NO *NOAUTO* *NOT* *NULL*
NULLIF *NUMERIC* *NUM_LOG_BUFS* *NUM_LOG_BUFFERS*

O

OCTET_LENGTH *OF* *ON* *ONLY*
OPEN *OPTION* *OR* *ORDER*
OUTER *OUTPUT* *OUTPUT_TYPE* *OVERFLOW*

P

PAGE *PAGELENGTH* *PAGES* *PAGE_SIZE*
PARAMETERS *PASSWORD* *PERCENT* *PLAN*
POSITION *POST_EVENT* *PRECISION* *PREPARE*

PRESERVE	*PROCEDURE*	*PROTECTED*	*PRIMARY*
PRIVILEGES	*PUBLIC*		

Q

QUIT			

R

RAW_PARTITIONS	*RDB$DB_KEY*	*READ*	*REAL*
RECORD_VERSION	*REFERENCES*	*RELEASE*	*RESERV*
RESERVING	*RESTRICT*	*RETAIN*	*RETURN*
RETURNING_VALUES	*RETURNS*	*REVOKE*	*RIGHT*
ROLE	*ROLLBACK*	*ROW*	*ROWS*
RUNTIME			

S

SCHEMA	*SECOND*	*SEGMENT*	*SELECT*
SET	*SHADOW*	*SHARED*	*SHELL*
SHOW	*SINGULAR*	*SIZE*	*SMALLINT*
SNAPSHOT	*SOME*	*SORT*	*SQLCODE*
SQLERROR	*SQLWARNING*	*STABILITY*	*STARTING*
STARTS	*STATEMENT*	*STATIC*	*SUSPEND*

T

TABLE	*TEMPORARY*	*TERMINATOR*	*THEN*
TIES	*TIME*	*TIMESTAMP*	*TO*
TRANSACTION	*TRANSLATE*	*TRANSLATION*	*TRIGGER*
TRIM	*TRUE*	*TYPE*	

U

UNCOMMITTED	*UNION*	*UNIQUE*	*UNKNOWN*
UPDATE	*UPPER*	*USER*	*USING*

V

VALUE	*VALUES*	*VARCHAR*	*VARIABLE*
VARYING	*VERSION*	*VIEW*	

W

WAIT	***WEEKDAY***	***WHEN***	***WHENEVER***
WHERE	***WHILE***	***WITH***	***WORK***
WRITE			

Y

YEAR	YEARDAY

NOTE

The following keywords are specific to InterBase and are not part of the SQL standard.

WEEKDAY YEARDAY

Error Codes and Messages

This chapter summarizes InterBase error-handling options and error codes. Tables in this chapter 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 Embedded SQL 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 InterBase Software Corp.

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 Embedded SQL Guide:

1. Override the ***WHENEVER*** branching by inserting a ***WHENEVER SQL ERROR CONTINUE*** statement. The program now ignores SQLCODE.
2. Use a 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 chapter lists all status array codes in SQLCODE 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 the following library functions for retrieving and printing 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.

```
isc_print_sqlerror  (short  SQLCODE,  ISC_STATUS  *status_vector);
```

isc_sql_interprete( )

This function retrieves a 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.

```
isc_sql_interprete(short SQLCODE, char *buffer, short length);
```

Responding to 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:

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:

Topic	To find...	See...
SQLCODE and error handling	Complete discussion and programming instructions	Embedded SQL Guide
List of SQLCODE codes and messages	SQLCODE codes and messages and associated messages for embedded SQL, DSQL, isql	This chapter: SQLCODE Codes and Messages.
WHENEVER syntax	Usage and syntax	SQL Statement and Function Reference.
Programming WHENEVER	Using and programming error-handling routines	Embedded SQL Guide
InterBase status array and functions	Complete programming instructions	Embedded SQL Guide
List of status array codes	Status array error codes and associated messages for embedded SQL, DSQL, isql	This chapter: InterBase Status Array.

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:

SQLCODE and messages summary		
SQLCODE	**Message**	**Meaning**
<0	SQLERROR	Error: The statement did not complete; table 5.4 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.

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

SQLCODE codes and messages		
SQLCODE	**SQLCODE text**	**InterBase number.**
-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
-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	Encryption <string> has bad length of <string> bits.	336003096L
-204	Reference to invalid stream number.	335544502L
-204	CHARACTER SET <string> is not defined.	335544509L

SQLCODE codes and messages		
SQLCODE	**SQLCODE text**	**InterBase number.**
-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
-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
-204	EXECUTE STATEMENT could not prepare statement : <*string*>.	335544850
-204	SQL statement invalid as it returns no records. SQL : <*string*>.	335544851
-204	Parameter mis-match for the statement : <*string*>.	335544852
-204	Could not execute statement : <*string*>.	335544853
-204	EXECUTE STATEMENT fetch error.	335544854
-204	EXECUTE STATEMENT in this form must return single row, not multiple rows.	335544855
-204	SQL statement not allowed in EXECUTE STATEMENT : <*string*>.	335544857
-204	Statement evaluated to a NULL statement. EXECUTE STATEMENT cannot execute a NULL statement.	335544858
-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
-239	Cache length too small.	335544691L
-260	Cache redefined.	335544690L

SQLCODE codes and messages		
SQLCODE	**SQLCODE text**	**InterBase number.**
-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 NOTNULL.	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
-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

SQLCODE codes and messages		
SQLCODE	**SQLCODE text**	**InterBase number.**
-552	Only the owner of a table can 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
-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/TIME/TIMESTAMP 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	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
-663	Too many keys defined for index <string>.	335544631L

SQLCODE codes and messages		
SQLCODE	**SQLCODE text**	**InterBase number.**
-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 version, 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
-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

SQLCODE codes and messages		
SQLCODE	**SQLCODE text**	**InterBase number.**
-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
-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

SQLCODE codes and messages		
SQLCODE	**SQLCODE text**	**InterBase number.**
-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
-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

SQLCODE codes and messages		
SQLCODE	**SQLCODE text**	**InterBase number.**
-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
-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

SQLCODE codes and messages		
SQLCODE	**SQLCODE text**	**InterBase number.**
-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
-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	Maximum number of online dump files that can be specified is 16	335544483L
-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

SQLCODE codes and messages		
SQLCODE	**SQLCODE text**	**InterBase number.**
-904	Request depth exceeded. (Recursive definition?)	335544683L
-904	Size of optimizer block exceeded.	335544762L
-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
-924	Bad parameters on attach or create database.	335544325L
-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.***

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

Error code	Number	Message
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
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

Error code	Number	Message
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
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

Error code	Number	Message
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
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>
isc_wrodynver	335544437L	wrong DYN version

Error code	Number	Message
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>
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>

Error code	Number	Message
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_num_old_files	335544483L	maximum number of online dump files that can be specified is 16
isc_bad_stmt_handle	335544485L	invalid statement handle
isc_stream_not_defined	335544502L	reference to invalid stream number
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_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_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 NOTNULL.
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

Error code	Number	Message
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).
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_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_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	Datatype for constant unknown
isc_dsql_cursor_err	335544572L	Cursor unknown
isc_dsql_datatype_err	335544573L	Datatype unknown
isc_dsql_decl_err	335544574L	Declared cursor already exists
isc_dsql_cursor_update_err	335544575L	Cursor not updatable
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

Error code	Number	Message
isc_dsql_request_err	335544582L	Request unknown
isc_dsql_sqlda_err	335544583L	SQLDA missing or incorrect version, 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
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

Error code	Number	Message
isc_return_mode_err	335544618L	Return mode by value not allowed for this datatype
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>
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

Error code	Number	Message
isc_invalid_direction	335544655L	invalid direction for find operation
isc_dsql_var_conflict	335544656L	variable <string> conflicts with parameter in same procedure
isc_dsql_no_blob_array	335544657L	Array/Blob/DATE / TIME/TIMESTAMP 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?)
isc_no_field_access	335544684L	cannot access column <string> in view <string>
isc_no_dbkey	335544685L	dbkey not available for multi-table views
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

Error code	Number	Message
isc_cache_too_small	335544691L	Cache length too small
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.
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>
isc_svc_in_use	335544716L	service is currently busy: <<string>>
isc_err_stack_limit	335544717L	stack size insufficient to execute current request
isc_invalid_key	335544718L	invalid key for find operation
isc_net_init_error	335544719L	error initializing the network software
isc_loadlib_failure	335544720L	unable to load required library <<string>>
isc_network_error	335544721L	unable to complete network request to host "<<string>>"
isc_net_connect_err	335544722L	failed to establish a connection
isc_net_connect_listen_err	335544723L	error while listening for an incoming connection
isc_net_event_connect_err	335544724L	failed to establish a secondary connection for event processing
isc_net_event_listen_err	335544725L	error while listening for an incoming event connection request
isc_net_read_err	335544726L	error reading data from the connection
isc_net_write_err	335544727L	error writing data to the connection
isc_integ_index_deactivate	335544728L	cannot deactivate index used by an Integrity Constraint
isc_integ_deactivate_primary	335544729L	cannot deactivate primary index
isc_unsupported_network_drive	335544732L	access to databases on file servers is not supported
isc_io_create_err	335544733L	error while trying to create file
isc_io_open_err	335544734L	error while trying to open file

Error code	Number	Message
isc_io_close_err	335544735L	error while trying to close file
isc_io_read_err	335544736L	error while trying to read from file
isc_io_write_err	335544737L	error while trying to write to file
isc_io_delete_err	335544738L	error while trying to delete file
isc_io_access_err	335544739L	error while trying to access file
isc_udf_exception	335544740L	exception <<integer>> detected in blob filter or user defined function
isc_lost_db_connection	335544741L	connection lost to database
isc_no_write_user_priv	335544742L	user cannot write to RDB$USER_PRIVILEGES
isc_token_too_long	335544743L	token size exceeds limit
isc_max_att_exceeded	335544744L	maximum user count exceeded; contact your database administrator
isc_login_same_as_role_name	335544745L	your login <<string>> is same as one of the SQL role names; ask your database administrator to set up a valid InterBase login
isc_reftable_requires_pk	335544746L	“REFERENCES table” without “(column)”; requires PRIMARY KEY on referenced table
isc_usrname_too_long	335544747L	the username entered is too long. Maximum length is 31 bytes.
isc_password_too_long	335544748L	the password specified is too long. Maximum length is 8 bytes.
isc_usrname_required	335544749L	a username is required for this operation.
isc_password_required	335544750L	a password is required for this operation
isc_bad_protocol	335544751L	the network protocol specified is invalid
isc_dup_usrname_found	335544752L	a duplicate user name was found in the security database
isc_usrname_not_found	335544753L	the user name specified was not found in the security database
isc_error_adding_sec_record	335544754L	error while attempting to add the user
isc_error_modifying_sec_record	335544755L	error while attempting to modify the user record
isc_error_deleting_sec_record	335544756L	error while attempting to delete the user record
eisc_rror_updating_sec_db	335544757L	error while updating the security database
isc_sort_rec_size_err	335544758L	sort record size is too big
isc_bad_default_value	335544759L	cannot assign a NULL default value to a column with a NOTNULL constraint
isc_invalid_clause	335544760L	the specified user-entered string is not valid
isc_too_many_handles	335544761L	too many open handles to database
isc_optimizer_blk_exc	335544762L	optimizer implementation limits are exceeded; for example, only 256 conjuncts (AND and OR) are allowed

System Tables, Temporary Tables, and Views

This chapter describes the InterBase system tables, SQL system views, and Change Views.

IMPORTANT

Only InterBase system object names can begin with the characters "RDB$" or "TMP$". No other object name in InterBase can begin with these character sequences, including tables, views, triggers, stored procedures, indexes, generators, domains, and roles.

Overview of System Tables, Temporary Tables, and Views

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 temporary system tables allow access to information about the database and its connections and a degree of control over transactions. By default, all users can select from permanent system tables, but only the database owner and the SYSDBA user can write to them. To gain access to temporary system tables, explicit access has to be granted to them by the database owner or the SYSDBA. These users can grant write access to others if they wish. See the Operations Guide for details about system table security.

SQL system views provide information about existing integrity constraints for a database. You must create system views yourself by creating and 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

The following table lists all InterBase system tables. The names of system tables and the names of the columns of system tables start with RDB$.

System table	Description
RDB$CHARACTER SETS	Describes the valid character sets available in InterBase.
RDB$JOURNAL ARCHIVES	Stores information about the repository of database and journal archive files.
RDB$CHECK CONSTRAINTS	Stores database integrity constraint information for CHECK constraints. In addition, the table stores information for constraints implemented with NOTNULL.
RDB$LOG_FILES	RDB$LOG_FILES is deprecated.
RDB$COLLATIONS	Records the valid collating sequences available for use in InterBase.
RDB$PAGES	Keeps track of each page allocated to the database.

System table	Description
RDB$DATABASE	Defines a database.
RDB$PROCEDURE PARAMETERS	Stores information about each parameter for each of the procedures of a database.
RDB$DEPENDENCIES	Keeps track of the tables and columns upon which other system objects depend. These objects include views, triggers, and computed columns.
RDB$PROCEDURES	Stores information about a stored procedures of a database.
RDB$ENCRYPTIONS	Describes the characteristics of encryptions stored in the database.
RDB$REF CONSTRAINTS	Stores information about referential integrity constraints.
RDB$EXCEPTIONS	Describes error conditions related to stored procedures, including user-defined exceptions.
RDB$RELATION CONSTRAINTS	Stores information about integrity constraints for tables.
RDB$FIELD DIMENSIONS	Describes each dimension of an array column.
RDB$RELATION FIELDS	For database tables, lists columns and describes column characteristics for domains.
RDB$FIELDS	Defines the characteristics of a column.
RDB$RELATIONS	Defines some of the characteristics of tables and views.
RDB$FILES	Lists the secondary files and shadow files for a database.
RDB$ROLES	Lists roles that have been defined in the database and the owner of each role.
RDB$FILTERS	Tracks information about a blob filter.
RDB$SECURITY CLASSES	Defines access control lists and associates them with databases, tables, views, and columns in tables and views.
RDB$FORMATS	Keeps track of the format versions of the columns in a table.
RDB$TRANSACTIONS	Keeps track of all multi-database transactions.
RDB$FUNCTION ARGUMENTS	Defines the attributes of a function argument.
RDB$TRIGGER MESSAGES	Defines a trigger message and associates the message with a particular trigger.
RDB$FUNCTIONS	Defines a user-defined function.
RDB$TRIGGERS	Defines triggers.
RDB$GENERATORS	Stores information about generators, which provide the ability to generate a unique identifier for a table.
RDB$TYPES	Records enumerated data types and alias names for InterBase character sets and collation orders.
RDB$INDEX SEGMENTS	Specifies the columns that comprise an index for a table.
RDB$USER PRIVILEGES	Keeps track of the privileges assigned to a user through a SQL GRANT statement.
RDB$INDICES	Defines the index structures that allow InterBase to locate rows in the database more quickly.
RDB$USERS	Only permits users in that system table access to the database.
RDB$VIEW RELATIONS	Not used by SQL objects.
RDB$SUBSCRIBERS	Stores subscribers information.
RDB$SUBSCRIPTIONS	Stores subscription information.

RDB$CHARACTER SETS

RDB$CHARACTER_SETS describes the valid character sets available in InterBase.

Column name	Data type	Length	Description
RDB$CHARACTER_SET_NAME	CHAR	67	Name of a character set that InterBase recognizes.
RDB$FORM_OF_USE	CHAR	67	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	67	Subtype 2: default collation sequence for the character set.
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 or NULL). • System-defined (value of 1).
RDB$DESCRIPTION	BLOB		Subtype text: Contains a user-written description of the character set.
RDB$FUNCTION_NAME	CHAR	67	Reserved for internal use; subtype 2.
RDB$BYTES_PER_CHARACTER	SMALLINT		Size of character in bytes.

RDB$JOURNAL ARCHIVES

RDB$JOURNAL_ARCHIVES stores information about the repository of database and journal archive files.

RDB$JOURNAL_ARCHIVES			
Column name	**Data type**	**Length**	**Description**
RDB$ARCHIVE_ NAME	VARCHAR	1024	The name of the archived item.
RDB$ARCHIVE_ TYPE	CHAR	1	The type of the archived item. 'D' indicates a database dump. 'S' indicates a secondary database file of a database dump. 'J' indicates a journal file.
RDB$ARCHIVE_ LENGTH	INT64	8	Length of the archived item as stored in bytes.
RDB$ARCHIVE_ SEQUENCE	INTEGER	4	Sequence number of archive item.
RDB$ARCHIVE_ TIMESTAMP	TIMESTAMP	8	Timestamp when item was stored in the archive.
RDB$DEPENDED_ ON_SEQUENCE	INTEGER	4	Sequence of archived item that this item depends on. For 'S' archive types, it would be the sequence number of the 'D' primary database dump file. For 'D' archive types, it is the sequence number of the starting journal file for recovering from the archive.
RDB$DEPENDED_ ON_TIMESTAMP	TIMESTAMP	8	As above, but the archive timestamp for the depended on archive item.

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***.

Column name	Data type	Length	Description
RDB$CONSTRAINT_NAME	CHAR	67	Subtype 2: Name of a ***CHECK*** or ***NOT NULL*** constraint
RDB$TRIGGER_NAME	CHAR	67	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.

Column name	Data type	Length	Description
RDB$COLLATION_NAME	CHAR	67	Name of a valid collation sequence in InterBase.
RDB$COLLATION_ID	SMALLINT		Unique identifier for the collation sequence.
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		Subtype Text: Contains a user-written description of the collation sequence.
RDB$FUNCTION_NAME	CHAR	67	Reserved for internal use.

RDB$PAGES

RDB$PAGES keeps track of each page allocated to the database.

IMPORTANT

Modifying this table in any way corrupts a database.

RDB$PAGES			
Column name	**Data type**	**Length**	**Description**
RDB$PAGE_NUMBER	INTEGER		The physically allocated page number
RDB$RELATION_ID	SMALLINT		Identifier number of the table for which this page is allocated
RDB$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$DATABASE

RDB$DATABASE defines a database.

RDB$DATABASE			
Column name	**Data type**	**Length**	**Description**
RDB$DESCRIPTION	BLOB		Subtype Text: Contains a user-written description of the database; when a comment is included in a ***CREATE***, ***ALTER SCHEMA*** or ***ALTER DATABASE*** statement, isql writes to this column.
RDB$RELATION_ID	SMALLINT		For internal use by InterBase
RDB$SECURITY_CLASS	CHAR	67	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	67	Subtype 2; Name of character set
RDB$PAGE_CACHE	INTEGER		Sets database page buffer cache limit. Also, tries to expand cache to that limit.
RDB$PROCEDURE_CACHE	INTEGER		
RDB$TRIGGER_CACHE	INTEGER		
RDB$RELATION_CACHE	SMALLINT		
RDB$FLUSH_INTERVAL	INTEGER		Enables database flush. The interval <number> is interpreted in units of seconds.
RDB$LINGER_INTERVAL	INTEGER		Allows a database to remain in memory after the last user detaches. Interval is seconds.
RDB$RECLAIM_INTERVAL	INTEGER		Reclaim interval is in seconds. Determines how often the garbage collector thread will run to release memory from unused procedures, triggers, and internal system queries back to InterBase memory heap.
RDB$SWEEP_INTERVAL	INTEGER		
RDB$GROUP_COMMIT	CHAR(1)		
RDB$PASSWORD_DIGEST	VARCHAR(16)		

RDB$PROCEDURE PARAMETERS

RDB$PROCEDURE_PARAMETERS stores information about each parameter for each of a database's procedures.

RDB$PROCEDURE_PARAMETERS			
Column name	**Data type**	**Length**	**Description**
RDB$PARAMETER_NAME	CHAR	67	Parameter name
RDB$PROCEDURE_NAME	CHAR	67	Name of the procedure in which the parameter is used
RDB$PARAMETER_NUMBER	SMALLINT		Parameter sequence number
RDB$PARAMETER_TYPE	SMALLINT		Parameter data type Values are: • 0 = input • 1 = output
RDB$FIELD_SOURCE	CHAR	67	Global column name
RDB$DESCRIPTION	BLOB		Subtype Text: User-written description of the parameter
RDB$SYSTEM_FLAG	SMALLINT		Indicates whether the parameter is: • User-defined (value of 0) • System-defined (value greater than 0)

RDB$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.

RDB$DEPENDENCIES			
Column name	**Data type**	**Length**	**Description**
RDB$DEPENDENT_NAME	CHAR	67	Subtype 2; names the object this table tracks: a view, trigger, or computed column.
RDB$DEPENDED_ON_NAME	CHAR	67	Subtype 2; names the table referenced by the object named above.
RDB$FIELD_NAME	CHAR	67	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$TYPES): • 0 - table • 1 - view • 2 - trigger • 3 - computed_field • 4 - validation • 5 - procedure • 7 - exception • 8 - user • 9 - field • 10 - index All other values are reserved for future use.
RDB$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 • 7 - exception • 8 - user • 9 - field • 10 - index • 11 - generator • 14 - External Functions • 15 - Encryption

RDB$DEPENDENCIES			
Column name	**Data type**	**Length**	**Description**
			All other values are reserved for future use.

RDB$PROCEDURES

RDB$PROCEDURES stores information about a database's stored procedures.

RDB$PROCEDURES			
Column name	**Data type**	**Length**	**Description**
RDB$PROCEDURE_NAME	CHAR	67	Procedure name
RDB$PROCEDURE_ID	SMALLINT		Procedure number
RDB$PROCEDURE_INPUTS	SMALLINT		Number of input parameters
RDB$PROCEDURE_OUTPUTS	SMALLINT		Number of output parameters
RDB$DESCRIPTION	BLOB		Subtype Text: User-written description of the procedure
RDB$PROCEDURE_SOURCE	BLOB		Subtype Text: Source code for the procedure
RDB$PROCEDURE_BLR	BLOB		Subtype BLR: BLR (Binary Language Representation) of the procedure source
RDB$SECURITY_CLASS	CHAR	67	Security class of the procedure
RDB$OWNER_NAME	CHAR	67	User who created the procedure (the owner for SQL security purposes)
RDB$RUNTIME	BLOB		Subtype Summary: Describes procedure metadata; used for performance enhancement
RDB$SYSTEM_FLAG	SMALLINT		Indicates whether the procedure is: • User-defined (value of 0) • System-defined (value greater than 0)

RDB$ENCRYPTIONS

RDB$ENCRYPTIONS describes the characteristics of encryptions stored in the database.

RDB$ENCRYPTIONS			
Column name	**Data type**	**Length**	**Description**
RDB$ENCRYPTION_NAME	CHAR	67	A unique name for the encryption.
RDB$ENCRYPTION_TYPE	CHAR	16	BASE: Defines a base encryption that has its own encryption value. COPY: Copy of a BASE encryption that shares the same encryption value. BACKUP: Defines an encryption used to encrypt database backup files. RECOVERY: Defines an encryption that can be used to recover a password-protected encryption when the password has been lost or forgotten. This encryption cannot be used to perform database encryption
RDB$ENCRYPTION_CIPHER	CHAR	16	Encryption cipher algorithm. This is either AES (Advanced Encryption Standard) or DES (Data Encryption Standard).
RDB$ENCRYPTION_ LENGTH	SMALLINT		Encryption key length (bits) must be one of these values for AES: 128, 192 or 256. The default is 128. For DES the default is 56.
RDB$ENCRYPTION_INIT_ VECTOR	CHAR	6	RANDOM: specifies that random bytes should be used with cipher block chaining (CBC) encryption mode. <null>: default, specifies electronic cookbook (ECB) encryption mode used.
RDB$ENCRYPTION_PAD	CHAR	6	RANDOM: pads value to be encrypted with random bytes.
RDB$ENCRYPTION_VALUE	CHAR	68	Encrypted value of the actual encryption key value.
RDB$ENCRYPTION_SALT	CHAR	68	Hash to verify decrypted value of actual encryption key value is correct.
RDB$ENCRYPTION_ TIMESTAMP	TIMESTAMP		Timestamp when encryption key value was created or refreshed.
RDB$ENCRYPTION_ID	SMALLINT		Unique identifier for Encryption key.
RDB$SECURITY_CLASS	CHAR	67	Names a security class stored in RDB $SECURITY_CLASSES.
RDB$OWNER_NAME	CHAR	67	Owner of the encryption
RDB$PASSWORD2	VARCHAR	68	Password hash used to allow access to the encryption.
RDB$SYSTEM_FLAG	SMALLINT		0: User-defined 1: System-defined.
RDB$FLAGS	SMALLINT	2	1: random initialization vector defined for cipher block chaining encryption mode. 2: random padding of plaintext 4: encryption is marked for deletion.

RDB$ENCRYPTIONS			
Column name	**Data type**	**Length**	**Description**
			32: indicates one or more subscriptions on the relation
RDB$DESCRIPTION	BLOB		Subtype Text: User-written description of encryption.

RDB$REF CONSTRAINTS

RDB$REF_CONSTRAINTS stores referential integrity constraint information.

RDB$REF_CONSTRAINTS			
Column name	**Data type**	**Length**	**Description**
RDB$3CONSTRAINT_NAME	CHAR	67	Name of a referential constraint
RDB$CONST_NAME_UQ	CHAR	67	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	Specifies the type of action on the foreign key when the primary key is updated; values are ***RESTRICT***, ***NO ACTION***, ***CASCADE***, ***SET NULL***, or ***SET DEFAULT***
RDB$DELETE_RULE	CHAR	11	Specifies the type of action on the foreign key when the primary key is ***DELETED***; values are ***RESTRICT***, ***NO ACTION***, ***CASCADE***, ***SET NULL***, or ***SET DEFAULT***

RDB$EXCEPTIONS

RDB$EXCEPTIONS describes error conditions related to stored procedures, including user-defined exceptions.

RDB$EXCEPTIONS			
Column name	**Data type**	**Length**	**Description**
RDB$EXCEPTION_NAME	CHAR	67	Subtype 2; exception name
RDB$EXCEPTION_NUMBER	INTEGER		Number for the exception
RDB$MESSAGE	VARCHAR	78	Text of exception message
RDB$DESCRIPTION	BLOB		Subtype Text: Text description of the exception
RDB$SYSTEM_FLAG	SMALLINT		Displays null

RDB$RELATION CONSTRAINTS

RDB$RELATION_CONSTRAINTS stores information about integrity constraints for tables.

RDB$RELATION_CONSTRAINTS				
Column name	**Data type**	**Length**	**Description**	
RDB$CONSTRAINT_NAME	CHAR	67	Name of a table constraint	
RDB$CONSTRAINT_TYPE	CHAR	11	Type of table constraint Constraint types are:	
			• ***PRIMARY KEY*** • ***UNIQUE*** • ***FOREIGN KEY***	• ***CHECK*** • ***NOTNULL***
RDB$RELATION_NAME	CHAR	67	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	67	Name of the index used by ***UNIQUE***, ***PRIMARY KEY***, or ***FOREIGN KEY*** constraints	

RDB$FIELD DIMENSIONS

RDB$FIELD_DIMENSIONS describes each dimension of an array column.

RDB$FIELD_DIMENSIONS			
Column name	**Data type**	**Length**	**Description**
RDB$FIELD_NAME	CHAR	67	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$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 such as column type and length. For both domains and simple columns, this table may contain default and nullability information.

Column name	Data Type	Length	Description
RDB$FIELD_NAME	CHAR	67	Column name defined by the user.
RDB$RELATION_NAME	CHAR	67	Table name defined by the user.
RDB$FIELD_SOURCE	CHAR	67	Internal Column name that matches up with RDB $FIELDS.RDB$FIELD_NAME.
RDB$QUERY_NAME	CHAR	67	Alternate column name for use in isql; supersedes the value in RDB$FIELDS.
RDB$BASE_FIELD	CHAR	67	Views only: The name of the column from RDB$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$QUERY_HEADER	BLOB		Not used in SQL.
RDB$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.

Column name	Data Type	Length	Description
RDB$DESCRIPTION	BLOB		Subtype Text: User-written description of the column being defined.
RDB$DEFAULT_VALUE	BLOB		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	67	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	67	Reserved for future use.
RDB$NULL_FLAG	SMALLINT		Indicates whether the column may contain NULL values.
RDB$DEFAULT_SOURCE	BLOB		Subtype Text: SQL source to define defaults.
RDB$COLLATION_ID	SMALLINT		Unique identifier for the collation sequence.
RDB$ENCRYPTION_ID	SMALLINT		Identifies encryption ID from RDB$ENCRYPTIONS used to encrypt this column.
RDB$DECRYPT_DEFAULT_VALUE	BLOB		Subtype BLR: BLR (Binary Language Representation) for decrypt default clause.
RDB$DECRYPT_DEFAULT_SOURCE	BLOB		Subtype Text: SQL to define decrypt default.
RDB$FLAGS	SMALLINT	2	1 = One or more subscriptions on the field

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 can contain default and null status information.

Column name	Data Type	Length	Description
RDB$FIELD_NAME	CHAR	67	Unique name of a domain or system-assigned name for a column, starting with SQL<nnn>; the actual column names are stored in the RDB$FIELD_SOURCE column of RDB$RELATION_FIELDS.
RDB$QUERY_NAME	CHAR	67	Not used for SQL objects.
RDB$VALIDATION_BLR	BLOB		Not used for SQL objects.
RDB$VALIDATION_SOURCE	BLOB		Not used for SQL objects.
RDB$COMPUTED_BLR	BLOB		Subtype BLR; for computed columns, contains the BLR (Binary Language Representation) of the expression the database evaluates at the time of execution.
RDB$COMPUTED_SOURCE	BLOB		Subtype Text: For computed columns, contains the original CHAR source expression for the column.
RDB$DEFAULT_VALUE	BLOB		Stores default rule; subtype BLR.
RDB$DEFAULT_SOURCE	BLOB		Subtype Text; SQL description of a default value.
RDB$FIELD_LENGTH	SMALLINT		Length in bytes of the field this row defines: For CHAR, VARCHAR, and NCHAR data types, this is the maximum length of the field, and InterBase uses this length when creating indexes on columns. For non-CHAR related data types, the column lengths are: • D_FLOAT - 8 • DOUBLE - 8 • DATE - 4 • BLOB - 8 • TIME - 4 • INT64 - 8 • SHORT - 2 • LONG - 4 • QUAD - 8 • FLOAT - 4 • TIMESTAMP - 8 • BOOLEAN - 2
RDB$FIELD_PRECISION	SMALLINT		Stores the precision for numeric and decimal types.
RDB$FIELD_SCALE	SMALLINT		Stores negative scale for numeric and decimal types.

Column name	Data Type	Length	Description
RDB$FIELD_TYPE	SMALLINT		Specifies the data type of the column being defined; changing the value of this column automatically changes the data type for all columns based on the column being defined. Valid values are: • BLOB - 261 • BLOB_ID - 45 • BOOLEAN - 17 • CHAR - 14 • CSTRING - 40 • D_FLOAT - 11 • DOUBLE - 27 • FLOAT - 10 • INT64 - 16 • INTEGER - 8 • QUAD - 9 • SMALLINT - 7 • DATE - 12 (dialect 3 DATE) • TIME - 13 • TIMESTAMP - 35 • VARCHAR - 37 Restrictions: • The value of this column cannot be changed to or from BLOB. • Non-numeric data causes a conversion error in a column changed from CHAR to numeric. Changing data from CHAR to numeric and back again adversely affects index performance; for best results, delete and re-create indexes when making this type of change.
RDB$FIELD_SUB_TYPE	SMALLINT		Used to distinguish types of Blobs, CHAR values, and integers. **1** If RDB$FIELD_TYPE is 261 (Blob), predefined subtypes can be: • 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

Column name	Data Type	Length	Description
			• 6 - description of multi-database transaction that finished irregularly **2** If RDB$FIELD_TYPE is 14 (CHAR), columns can be: • 0 - type is unspecified • 1 - fixed BINARY data Corresponds to the RDB$FIELD_SUB_TYPE column in the RDB$COLLATIONS table. **3** If RDB$FIELD_TYPE is 7 (SMALLINT), 8 (INTEGER), or 16 (INT64), the original declaration was: • 0 or NULL- RDB$FIELD_TYPE • 1 - NUMERIC • 2 - DECIMAL
RDB$MISSING_VALUE	BLOB		Not used for SQL objects.
RDB$MISSING_SOURCE	BLOB		Not used for SQL objects.
RDB$DESCRIPTION	BLOB		Subtype Text: Contains a user-written description of the column being defined.
RDB$SYSTEM_FLAG	SMALLINT		For system tables
RDB$QUERY_HEADER	BLOB		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: • BLOB - 261 • BLOB_ID - 45 • BOOLEAN - 17 • CHAR - 14 • CSTRING - 40 • D_FLOAT - 11 • DOUBLE - 27 • FLOAT - 10 • INT64 - 16 • INTEGER - 8 • QUAD - 9 • SMALLINT - 7 • DATE - 12

Column name	Data Type	Length	Description
			(dialect 3 DATE) • TIME - 13 • TIMESTAMP - 35 • VARCHAR - 37
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. Valid values are: • Empty: Can contain NULL values. • 1: Cannot contain NULL values.
RDB$CHARACTER_LENGTH	SMALLINT		Length in characters of the field this row defines: For CHAR, VARCHAR, and NCHAR data types, this is the quotient of RDB$FIELD_LENGTH divided by the number of bytes per character in the character set of the field. For other data types, this length value is not meaningful, and should be NULL.
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$CHARACTER_SETS system table.
RDB$SUBSCRIBE_FLAG	SMALLINT	2	Indicates one or more subscriptions of the field.

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.

RDB$RELATIONS			
Column name	**Data Type**	**Length**	**Description**
RDB$VIEW_BLR	BLOB		Subtype BLR: For a view, contains the BLR (Binary Language Representation) of the query InterBase evaluates at the time of execution.
RDB$VIEW_SOURCE	BLOB		Subtype Text: For a view, contains the original source query for the view definition.
RDB$DESCRIPTION	BLOB		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)

RDB$RELATIONS			
Column name	**Data Type**	**Length**	**Description**
			Do *not* set this column to 1 when creating tables.
RDB$DBKEY_LENGTH	SMALLINT		Length of the database key. Values are: • 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$RELATION_NAME	CHAR	67	The unique name of the table defined by this row
RDB$SECURITY_CLASS	CHAR	67	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; if this is blank, the table does not correspond to an external file.
RDB$RUNTIME	BLOB		Subtype Summary: Describes table metadata; used for performance enhancement.
RDB$EXTERNAL_DESCRIPTION	BLOB		Subtype EXTERNAL_FILE_DESCRIPTION; user-written description of the external file.
RDB$OWNER_NAME	CHAR	67	Identifies the creator of the table or view; the creator is considered the owner for SQL security (GRANT/ REVOKE) purposes.
RDB$DEFAULT_CLASS	CHAR	67	Default security class that InterBase applies to columns newly added to a table using the SQL security system.
RDB$FLAGS	SMALLINT		• 1 = SQL-defined table • 2 = Global temporary table • 4 = <reserved for future use> • 8 = Delete temporary rows on commit • 16 = Preserve temporary rows on commit; rows are deleted on database detach • 32 = Indicates one or more subscriptions on the relation
RDB$DATA_BLOCKING_FACTOR	SMALLINT		ODS 15 creates a new column which stores a table-specific record blocking factor. It is set during GBAK restore based on the characteristics of the restored data. If a table does not have a table-specific data blocking factor, this system column queries as NULL.
RDB$BLOB_BLOCKING_FACTOR	SMALLINT		ODS 15 creates a new column which stores a table-specific blob blocking factor. It is set during GBAK restore based on the characteristics of the restored blobs.

RDB$RELATIONS			
Column name	**Data Type**	**Length**	**Description**
			If a table does not have a table-specific blob blocking factor, this system column queries as NULL. Note: If a table has Blob columns and no indexes defined, then the table uses the database-wide blocking factor as before.

RDB$FILES

RDB$FILES lists the secondary files and shadow files for a database.

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: 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$ROLES

RDB$ROLES lists roles that have been defined in the database and the owner of each role.

RDB$ROLES			
Column name	**Data type**	**Length**	**Description**
RDB$ROLE_NAME	CHAR	67	Name of role being defined
RDB$OWNER_NAME	CHAR	67	Name of InterBase user who is creating the role

RDB$FILTERS

RDB$FILTERS tracks information about a Blob filter.

RDB$FILTERS			
Column name	**Data type**	**Length**	**Description**
RDB$FUNCTION_NAME	CHAR	67	Unique name for the filter defined by this row
RDB$DESCRIPTION	BLOB		Subtype Text: Contains a user-written description of the filter being defined
RDB$MODULE_NAME	VARCHAR	253	Names the library where the filter executable is stored
RDB$ENTRYPOINT	CHAR	67	The entry point within the filter library for the Blob filter being defined
RDB$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$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.

RDB$SECURITY_CLASSES			
Column name	**Data type**	**Length**	**Description**
RDB$SECURITY_CLASS	CHAR	67	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		Subtype ACL: Access control list that specifies users and the privileges granted to those users
RDB$DESCRIPTION	BLOB		Subtype Text: User-written description of the security class being defined

RDB$FORMATS

RDB$FORMATS keeps track of the format versions of the columns in a table. InterBase assigns the table a new format number at each change to a column definition. Direct metadata operations such as ***ALTER TABLE*** increment the format version; so do creating, dropping, activating, and deactivating triggers. This table allows existing application programs to access a changed table, without needing to be recompiled.

NOTE

InterBase allows only 255 changes to a metadata of a table. Once the limit is reached, the database must be backed up and restored before more metadata changes can be made. Only changes that affect a structure count of a row toward this limit. Changing a trigger from active to inactive, for example, does not count toward the limit.

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		Subtype Format: Lists each column in the table, along with its data type, length, and scale (if applicable).

RDB$TRANSACTIONS

RDB$TRANSACTIONS keeps track of all multi-database transactions.

RDB$TRANSACTIONS			
Column name	**Data type**	**Length**	**Description**
RDB$TRANSACTION_ID	INTEGER		Identifies the multi-database transaction being described • On ODS 15, it remains INTEGER • On ODS 16, dialect 1 get a "double precision" type since it cannot support dtype_int64 • On ODS 16, dialect 3 (the majority of users) gets NUMERIC(18,0) which is the native dtype_int64 type.
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		Subtype TRANSACTION_DESCRIPTION; describes a prepared multi-database transaction, available if the reconnect fails

RDB$FUNCTION ARGUMENTS

RDB$FUNCTION_ARGUMENTS defines the attributes of a function argument.

Column name	Data type	Length	Description
RDB$FUNCTION_NAME	CHAR	67	Unique name of the function with which the argument is associated; must correspond to a function name in RDB$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$FIELD_TYPE	SMALLINT		Data type of the argument being defined Valid values are: • BLOB - 261 • BLOB_ID - 45 • BOOLEAN - 17 • CHAR - 14 • CSTRING - 40 • D_FLOAT - 11 • DOUBLE - 27 • FLOAT - 10 • INT64 - 16 • INTEGER - 8 • QUAD - 9 • SMALLINT - 7 • DATE - 12 (dialect 3 DATE) • TIME - 13 • TIMESTAMP - 35 • VARCHAR - 37
RDB$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		The length of the argument defined in this row Valid column lengths are: • BLOB - 8 • BOOLEAN - 2 • D_FLOAT - 8 • DATE - 4 • DOUBLE - 8 • FLOAT - 4

Column name	Data type	Length	Description
			• INT64 - 8 • LONG - 4 • QUAD - 8 • SHORT - 2 • TIME - 4 • TIMESTAMP - 8
RDB$FIELD_SUB_TYPE	SMALLINT		If RDB$FIELD_TYPE is 7 (SMALLINT),8 (INTEGER), or 16 (INT64) the subtype can be: • 0 or NULL - RDB$FIELD_TYPE • 1 - NUMERIC • 2 - DECIMAL
RDB$CHARACTER_SET_ID	SMALLINT		Unique numeric identifier for a character set
RDB$FIELD_PRECISION	SMALLINT		The declared precision of the DECIMAL or NUMERIC function argument

RDB$TRIGGER MESSAGES

RDB$TRIGGER_MESSAGES defines a trigger message and associates the message with a particular trigger.

RDB$TRIGGER_MESSAGES			
Column name	**Data type**	**Length**	**Description**
RDB$TRIGGER_NAME	CHAR	67	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$FUNCTIONS

RDB$FUNCTIONS defines a user-defined function.

RDB$FUNCTIONS			
Column name	**Data type**	**Length**	**Description**
RDB$FUNCTION_NAME	CHAR	67	Unique name for a function
RDB$FUNCTION_TYPE	SMALLINT		Reserved for future use
RDB$QUERY_NAME	CHAR	67	Alternate name for the function that can be used in isql
RDB$DESCRIPTION	BLOB		Subtype Text: Contains a user-written description of the function being defined
RDB$MODULE_NAME	VARCHAR	253	Names the function library where the executable function is stored
RDB$ENTRYPOINT	CHAR	67	Entry point within the function library for the function being defined
RDB$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$TRIGGERS

RDB$TRIGGERS defines triggers.

RDB$TRIGGERS			
Column name	**Data type**	**Length**	**Description**
RDB$TRIGGER_NAME	CHAR	67	Names the trigger being defined.
RDB$RELATION_NAME	CHAR	67	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 alphabetic order by trigger name. • If this number is not assigned by the user, InterBase assigns a value of 0.
RDB$TRIGGER_TYPE	SMALLINT		The type of trigger being defined. Values are: • 1 - BEFORE INSERT • 2 - AFTER INSERT • 3 - BEFORE UPDATE • 4 - AFTER UPDATE • 5 - BEFORE DELETE • 6 - AFTER DELETE
RDB$TRIGGER_SOURCE	BLOB		Subtype Text: Original source of the trigger definition; the isqlSHOW TRIGGERS statement displays information from this column.
RDB$TRIGGER_BLR	BLOB		Subtype BLR: BLR (Binary Language Representation) of the trigger source.
RDB$DESCRIPTION	BLOB		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		1 = SQL-defined trigger 2 = ignore permission checking User-defined triggers require that the user executing them have underlying access permission

RDB$TRIGGERS			
Column name	**Data type**	**Length**	**Description**
			to the objects accessed by the trigger. However, internal, system-defined triggers occasionally need to bypass those permission checks to enforce database integrity.

RDB$GENERATORS

RDB$GENERATORS stores information about generators, which provide the ability to generate a unique identifier for a table.

RDB$GENERATORS			
Column name	**Data type**	**Length**	**Description**
RDB$GENERATOR_NAME	CHAR	67	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$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.

RDB$TYPES			
Column name	**Data Type**	**Length**	**Description**
RDB$FIELD_NAME	CHAR	67	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	67	Text that corresponds to the internal number.
RDB$DESCRIPTION	BLOB		Subtype Text: Contains a user-written description of the enumerated data type being defined.
RDB$SYSTEM_FLAG	SMALLINT		Indicates whether the data type 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.

RDB$INDEX_SEGMENTS			
Column name	**Data type**	**Length**	**Description**
RDB$INDEX_NAME	CHAR	67	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	67	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$FIELD_POSITION	SMALLINT		Position of the index segment being defined; corresponds to the sort order of the index
RDB$STATISTICS	DOUBLE PRECISION		OSD 16 - Segment-specific statistics for index selectivity

RDB$USER PRIVILEGES

RDB$USER_PRIVILEGES keeps track of the privileges assigned to a user through a SQL GRANT statement. There is one occurrence of this table for each user/privilege intersection.

Column name	Data type	Length	Description
RDB$USER	CHAR	67	Names the user who was granted the privilege listed in the RDB$PRIVILEGE column.
RDB$GRANTOR	CHAR	67	Names the user who granted the privilege.
RDB$PRIVILEGE	CHAR	6	Identifies the privilege granted to the user listed in the RDB$USER column, above. The character stored in the field corresponds to the valid values listed below. Valid values are: • SELECT(S) • DELETE(D) • INSERT(I) • UPDATE(U) • REFERENCE (R) • MEMBER OF(M) (for roles) • DECRYPT (T) • ENCRYPT (E) • SUBSCRIBE (B) • EXECUTE(X)
RDB$GRANT_OPTION	SMALLINT		Indicates whether the privilege was granted with the WITH GRANT OPTION (value of 1) or not (value of 0).
RDB$RELATION_NAME	CHAR	67	Identifies the table or role to which the privilege applies.
RDB$FIELD_NAME	CHAR	67	For update privileges, identifies the column to which the privilege applies.
RDB$USER_TYPE	SMALLINT		
RDB$OBJECT_TYPE	SMALLINT		

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.

RDB$INDICES			
Column name	**Data type**	**Length**	**Description**
RDB$INDEX_NAME	CHAR	67	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	67	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		Subtype Text: User-written description of the index.
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 1) This is not set for system tables.
RDB$INDEX_TYPE	SMALLINT		Contains an internal identifier for sort order, either ascending (ASC) or descending (DESC): • ASC (value of 0) • DESC (value of 1)
RDB$FOREIGN_KEY	CHAR	67	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		Subtype BLR: Contains the BLR (Binary Language Representation) for the expression, evaluated by the database at execution time; used for PC semantics.

RDB$INDICES			
Column name	**Data type**	**Length**	**Description**
RDB$EXPRESSION_SOURCE	BLOB		Subtype Text: Contains original text source for the column; used for PC semantics.
RDB$STATISTICS	DOUBLE PRECISION		Selectivity factor for the index; the optimizer uses index selectivity, a measure of uniqueness for indexed columns, to choose an access strategy for a query.
RDB$INDEX_SPLIT_NULL	SMALLINT		Indicates if index should store NULL keys in different buckets.

RDB$USERS

RDB$USERS only permits users in that system table access to the database.

RDB$USERS			
Column name	**Data type**	**Length**	**Description**
RDB$USER_NAME (RDB $USER_NAME)	VARCHAR(128)	CHARACTER SET UNICODE_FSS Nullable	
RDB$SYSTEM_USER_NAME (RDB$USER_NAME)	VARCHAR(128)	CHARACTER SET UNICODE_FSS Nullable	
RDB$GROUP_NAME (RDB $USER_NAME)	VARCHAR(128)	CHARACTER SET UNICODE_FSS Nullable	
RDB$UID (RDB$UID)	INTEGER	Nullable	
RDB$GID (RDB$GID)	INTEGER	Nullable	
RDB$PASSWORD (RDB $PASSWORD)	VARCHAR(32)	CHARACTER SET OCTETS Nullable	
RDB$USER_ACTIVE (RDB $USER_ACTIVE)	CHAR(2)	Nullable	
RDB$USER_PRIVILEGE (RDB $USER_PRIVILEGE)	INTEGER	Nullable	
RDB$DESCRIPTION (RDB $DESCRIPTION)	BLOB	segment 80, subtype TEXT CHARACTER SET UNICODE_FSS Nullable	
RDB$FIRST_NAME (RDB $NAME_PART)	VARCHAR(32)	CHARACTER SET UNICODE_FSS Nullable	
RDB$MIDDLE_NAME (RDB $NAME_PART)	VARCHAR(32)	CHARACTER SET UNICODE_FSS Nullable	
RDB$LAST_NAME (RDB $NAME_PART	VARCHAR(32)	CHARACTER SET UNICODE_FSS Nullable	
RDB$DEFAULT_ROLE (RDB $USER)	CHAR(67)	CHARACTER SET UNICODE_FSS Nullable	
RDB$PASSWORD_DIGEST (RDB $PASSWORD_DIGEST)	VARCHAR(16)	Nullable	

RDB$VIEW RELATIONS

RDB$VIEW_RELATIONS is not used by SQL objects.

RDB$VIEW_RELATIONS			
Column name	**Data type 7**	**Length**	**Description**
RDB$VIEW_NAME	CHAR	67	Name of a view: The combination of RDB$VIEW_NAME and RDB$VIEW_CONTEXT must be unique
RDB$RELATION_NAME	CHAR	67	Name of a table referenced in the view definition
RDB$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	67	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

RDB$SUBSCRIBERS

The required Subscriber information is stored in a new system relation RDB$SUBSCRIBERS

RDB$SUBSCRIBERS			
Column name	**Data type**	**Length**	**Description**
RDB$SUBSCRIBER_NAME	CHAR	31	Name of subscribing user
RDB$SUBSCRIPTION_NAME	CHAR	67	Name of subscription
RDB$DESTINATION	CHAR	32	Destination of subscriber
RDB$FLAGS	SMALLINT	2	
RDB$CHECK_OUT_TRANSACTION_ID	INT64	8	Transaction ID of last subscription check out
RDB$CHECK_OUT_TIMESTAMP	TIMESTAMP	8	Date and time of last subscription check out
RDB$CHECK_OUT_OLDEST_ TRANSACTION_ID	INT64	8	Transaction of oldest active transaction at check out
RDB$CHECK_OUT_TRANSACTIONS	BLOB		Set of active transaction IDs at last transaction check out
RDB$CHECK_IN_TRANSACTION_ID	INT64	8	Transaction ID of last subscription check in
RDB$CHECK_IN_TIMESTAMP	TIMESTAMP	8	Date and time of last subscription check in
RDB$CHECK_IN_TRANSACTIONS	BLOB		Set of check in transaction IDs by this subscription

RDB$SUBSCRIPTIONS

Subscription information is stored in a new system relation RDB$SUBSCRIPTIONS with a unique key on RDB$SUBSCRIPTION_NAME, RDB$SUBSCRIBER_NAME, RDB$DESTINATION. Additional fields store control information to facilitate "check in" and "check out" of changed data.

This includes transaction IDs, timestamps and transactional context of last observation of changed data on the schema object. The term "check out" denotes SELECT of changed columns of rows from subscribed tables when a subscritpion has been activated. The term "check in" refers to INSERT, UPDATE and DELETE of changed columns of rows from subscribed tables when a subscription has been activated. A subscription becomes activated during a database session with the execution of OPEN SUBSCRIPTION. It is deactivated with the execution of CLOSED SUBSCRIPTION.

RDB$SUBSCRIPTION

Column name	Data type	Length	Description
RDB$SUBSCRIPTION_NAME	CHAR	67	Name of subscription
RDB$RELATION_NAME	CHAR	67	Name of relation or view
RDB$FIELD_NAME	CHAR	67	Name of field
RDB$DESCRIPTION	BLOB		Subtype text: User-written description of subscription
RDB$SECURITY CLASS	CHAR	67	Security class of the subscription (the owner for SQL security purposes)
RDB$OWNER_NAME	CHAR	67	User who created the subscription
RDB$RUNTIME	BLOB		Runtime binary information to enhance performance
RDB$FLAGS	SMALLINT	2	
RDB$INSERT	BOOLEAN	2	Inserts are tracked
RDB$UPDATE	BOOLEAN	2	Updates are tracked
RDB$DELETE	BOOLEAN	2	Deletes are tracked
RDB$CHANGE	BOOLEAN	2	Tracks all operations, but returns as soon as any column changes

System Temporary Tables

The InterBase server keeps a massive collection of information about its databases, connections, transactions, and statements. This information is made available through the following system temporary tables. For more information about using these tables, see the InterBase Operations Guide.

ODS-16: In System Temporary Tables, Performance Monitoring data counters are updated to 64-bit Integer type for dialect 3. Dialect 1 cannot support 64-bit Integer type, so a 64-bit Integer type is internally convered to type "double" as it is same in size (8 bytes). It also accommodates the large values for 64-bit addresses and counter values.

ODS 15 remains the same as before and only supports 32-bit Integer counters.

- ODS <= 15 will continue to have 32-bit INTEGER counters as before (for both dialect 1 and dialect 3 databases).
- ODS >= 16 will have the counters defined as "double precision" data type for dialect 1 databases.
- ODS >= 16 will have the counters defined as "NUMERIC(18,0)" data type for dialect 3 databases. As you know, by default, any new database is created as ODS 16, dialect 3.

Temporary table names begin with TMP$. InterBase offers the following system temporary tables:

TMP$ATTACHMENTS	TMP$DATABASE	TMP$HEAPS
TMP$POOL BLOCKS	TMP$POOLS	TMP$PROCEDURES
TMP$RELATIONS	TMP$STATEMENTS	TMP$TRANSACTIONS
TMP$TRIGGERS		

TMP$ATTACHMENTS

The TMP$ATTACHMENTS table contains one row for each connection to a database.

TMP$ATTACHMENTS				
Column name	**Data type for <=ODS 15**	**Data type for >=ODS 16, dialect 1**	**Data type for >=ODS 16, dialect 3**	**Description**
TMP$ATTACHMENT_ID	INTEGER			Connection identifier
TMP$DATABASE_ID	INTEGER			Database identifier
TMP$POOL_ID	INTEGER			Reserved
TMP$POOL_MEMORY	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Reserved
TMP$STATEMENTS	SMALLINT			Number of compiled statements
TMP$TRANSACTIONS	SMALLINT			Number of active transactions
TMP$TIMESTAMP	TIMESTAMP			Connection create timestamp
TMP$QUANTUM	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Units of execution
TMP$USER	CHAR[67]			User name
TMP$USER_IP_ADDR	CHAR[31]			User IP address
TMP$USER_HOST	CHAR[31]			User host name
TMP$USER_PROCESS	CHAR[31]			User process ID
TMP$STATE	CHAR[31]			CONNECTED,ACTIVE
TMP$PRIORITY	CHAR[31]			Reserved
TMP$DBKEY_ID	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Transaction ID of dbkey
TMP$ACTIVE_SORTS	SMALLINT			Number of active sorts
TMP$PAGE_READS	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Page reads all database files
TMP$PAGE_WRITES	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Page writes all database files
TMP$PAGE_FETCHES	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Page fetches all database files
TMP$PAGE_MARKS	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Page marks all database files
TMP$RECORD_SELECTS	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Records selected by connection
TMP$RECORD_INSERTS	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Records inserted by connection
TMP$RECORD_UPDATES	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Records updated by connection
TMP$RECORD_DELETES	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Records deleted by connection
TMP$RECORD_PURGES	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Garbage collect record purges

TMP$ATTACHMENTS				
Column name	**Data type for <=ODS 15**	**Data type for >=ODS 16, dialect 1**	**Data type for >=ODS 16, dialect 3**	**Description**
TMP$RECORD_EXPUNGES	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Garbage collect record expunges
TMP$RECORD_BACKOUTS	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Garbage collect record backouts

TMP$DATABASE

TMP$DATABASE contains one row for each database you are attached to.

TMP$DATABASE				
Column name	**Data type for <=ODS 15**	**Data type for >=ODS 16, dialect 1**	**Data type for >=ODS 16, dialect 3**	**Description**
TMP$DATABASE_ID	INTEGER			Database identifier
TMP$DATABASE_PATH	VARCHAR[253]			Database pathname
TMP$ATTACHMENTS	SMALLINT			Number of active connections
TMP$STATEMENTS	SMALLINT			Number of compiled statements
TMP$STATE	CHAR[31]			FLUSH, SWEEP, RECLAIM
TMP$ALLOCATED_PAGES	INTEGER			Pages allocated to all database files
TMP$POOLS	INTEGER			Number of memory pools
TMP$PROCEDURES	SMALLINT			Number of procedures loaded
TMP$RELATIONS	SMALLINT			Number of relations loaded
TMP$TRIGGERS	SMALLINT			Number of triggers loaded
TMP$ACTIVE_THREADS	SMALLINT			Active threads in database
TMP$SORT_MEMORY	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Sort buffer allocated memory
TMP$CURRENT_MEMORY	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Current memory allocated database
TMP$MAXIMUM_MEMORY	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Maximum memory ever allocated
TMP$PERMANENT_POOL_MEMORY	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Permanent pool memory size
TMP$CACHE_POOL_MEMORY	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Buffer pool memory size
TMP$TRANSACTIONS	SMALLINT			Number of active transactions
TMP$TRANSACTION_COMMITS	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Number of transaction commits
TMP$TRANSACTION_ROLLBACKS	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Number of transaction rollbacks
TMP$TRANSACTION_PREPARES	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Number of transaction prepares
TMP$TRANSACTION_DEADLOCKS	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Number of transaction deadlocks
TMP$TRANSACTION_CONFLICTS	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Number of transaction update conflicts

TMP$DATABASE				
Column name	**Data type for <=ODS 15**	**Data type for >=ODS 16, dialect 1**	**Data type for >=ODS 16, dialect 3**	**Description**
TMP$TRANSACTION_WAITS	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Number of transaction wait for
TMP$NEXT_TRANSACTION	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Next transaction number
TMP$OLDEST_INTERESTING	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Oldest interesting transaction
TMP$OLDEST_ACTIVE	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Oldest active transaction
TMP$OLDEST_SNAPSHOT	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Oldest snapshot transaction
TMP$CACHE_BUFFERS	INTEGER			Number of cache buffers
TMP$CACHE_PRECEDENCE	INTEGER			Nodes in cache precedence graph
TMP$CACHE_LATCH_WAITS	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Buffer latch waits
TMP$CACHE_FREE_WAITS	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Number of waits for a free buffer
TMP$CACHE_FREE_WRITES	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Number of writes to free buffers
TMP$SWEEP_INTERVAL	INTEGER			Sweep trigger interval
TMP$SWEEP_ACTIVE	CHAR[1]			Y (active) N (not-active)
TMP$SWEEP_RELATION	CHAR[67]			Relation currently being swept
TMP$SWEEP_RECORDS	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Records swept in above relation
TMP$PAGE_READS	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Page reads all database files
TMP$PAGE_WRITES	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Page writes all database files
TMP$PAGE_FETCHES	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Page fetches all database files
TMP$PAGE_MARKS	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Page marks all database files
TMP$RECORD_SELECTS	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Records selected from database
TMP$RECORD_INSERTS	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Records inserted into database
TMP$RECORD_UPDATES	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Records updated to database
TMP$RECORD_DELETES	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Records deleted from database
TMP$RECORD_PURGES	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Garbage collect record purges

TMP$DATABASE				
Column name	**Data type for <=ODS 15**	**Data type for >=ODS 16, dialect 1**	**Data type for >=ODS 16, dialect 3**	**Description**
TMP$RECORD_EXPUNGES	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Garbage collect record expunges
TMP$RECORD_BACKOUTS	INTEGER	DOUBLE PRECISION	NUMERIC (18, 0)	Garbage collect record backouts

TMP$HEAPS

TMP$HEAPS contains one row for each entry in the InterBase Random and Block heap.

TMP$HEAPS				
Column name	**Data Type for <=ODS 15**	**Data Type for >=ODS 16, dialect 1**	**Data Type for >=ODS 16, dialect 3**	**Description**
TMP$HEAP_TYPE	CHAR[31]			RANDOM or BLOCK
TMP$HEX_ADDRESS	CHAR[31]			Memory address of a free block in hex
TMP$ADDRESS		DOUBLE PRECISION	NUMERIC (18,0)	Memory address of free block
TMP$FREE_MEMORY	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Amount of free memory in the block

TMP$POOL BLOCKS

The TMP$POOL_BLOCKS table contains one row for each block of memory in each pool.

TMP$POOL_BLOCKS				
Column name	**Data type for >=ODS 16, dialect 1**	**Data type for >=ODS 16, dialect 1**	**Data type for >=ODS 16, dialect 3**	**Description**
TMP$POOL_ID	INTEGER			
TMP$ACC	INTEGER			
TMP$ARR	INTEGER			
TMP$ATT	INTEGER			
TMP$BCB	INTEGER			Buffer control block
TMP$BDB	INTEGER			Buffer descriptor block
TMP$BLB	INTEGER			Blob block
TMP$BLF	INTEGER			
TMP$BMS	INTEGER			
TMP$BTB	INTEGER			
TMP$BTC	INTEGER			
TMP$CHARSET	INTEGER			
TMP$CSB	INTEGER			Compiler scratch block
TMP$CSCONVERT	INTEGER			
TMP$DBB	INTEGER			Database block
TMP$DCC	INTEGER			Data compression control block
TMP$DFW	INTEGER			Deferred work block
TMP$DLS	INTEGER			
TMP$EXT	INTEGER			
TMP$FIL	INTEGER			File block
TMP$FLD	INTEGER			
TMP$FMT	INTEGER			Format block
TMP$FRB	INTEGER			Free block
TMP$FUN	INTEGER			
TMP$HNK	INTEGER			Hunk block
TMP$IDB	INTEGER			
TMP$IDL	INTEGER			
TMP$IRB	INTEGER			
TMP$IRL	INTEGER			
TMP$LCK	INTEGER			Lock block
TMP$LWT	INTEGER			
TMP$MAP	INTEGER			

TMP$POOL_BLOCKS				
Column name	**Data type for >=ODS 16, dialect 1**	**Data type for >=ODS 16, dialect 1**	**Data type for >=ODS 16, dialect 3**	**Description**
TMP$MFB	INTEGER			
TMP$NOD	INTEGER			Node block
TMP$OPT	INTEGER			
TMP$PRC	INTEGER			
TMP$PRE	INTEGER			Precedence block
TMP$PRM	INTEGER			
TMP$REC	INTEGER			Record block
TMP$REL	INTEGER			Relation block
TMP$REQ	INTEGER			Request block
TMP$RIV	INTEGER			
TMP$RSB	INTEGER			Record source block
TMP$RSC	INTEGER			
TMP$SAV	INTEGER			
TMP$SBM	INTEGER			Sparse bitmap block
TMP$SCL	INTEGER			
TMP$SDW	INTEGER			
TMP$SMB	INTEGER			Sort map block
TMP$SRPB	INTEGER			
TMP$STR	INTEGER			String block
TMP$SVC	INTEGER			
TMP$SYM	INTEGER			
TMP$TEXTTYPE	INTEGER			
TMP$TFB	INTEGER			Temporary field block
TMP$TPC	INTEGER			
TMP$TRA	INTEGER			Transaction block
TMP$USR	INTEGER			
TMP$VCL	INTEGER			Vector long block
TMP$VCT	INTEGER			
TMP$VCX	INTEGER			
TMP$XCP	INTEGER			

TMP$POOLS

The TMP$POOLS table contains one row for each current memory pool. A pool is a collection of memory to support the allocation needs of an internal system object.

TMP$POOLS				
Column name	**Data type for <=ODS 15**	**Data type for >=ODS 16, dialect 1**	**Data type for >=ODS 16, dialect 3**	**Description**
TMP$POOL_ID	INTEGER			Pool identifier
TMP$TYPE	CHAR[31]			Pool type
TMP$POOL_MEMORY	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Total memory in pool
TMP$FREE_MEMORY	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Free memory in pool
TMP$EXTEND_MEMORY	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Memory by which pool extended
TMP$FREE_STACK_NODES	SMALLINT			Free linked list stack nodes
TMP$FREE_BITMAP_BUCKETS	SMALLINT			Free bitmap buckets
TMP$FREE_BITMAP_SEGMENTS	INTEGER			Free bitmap segments

TMP$PROCEDURES

The TMP$PROCEDURES table contains one row for each procedure executed since the current connection began.

TMP$PROCEDURES				
Column name	**Data type for <=ODS 15**	**Data type for >=ODS 16, dialect 1**	**Data type for >=ODS 16, dialect 3**	**Description**
TMP$PROCEDURE_ID	INTEGER			Procedure identifier
TMP$DATABASE_ID	INTEGER			Database identifier
TMP$PROCEDURE_NAME	CHAR[67]			Procedure name
TMP$POOL_ID	INTEGER			Pool identifier
TMP$POOL_MEMORY	INTEGER			Pool memory size
TMP$CLONE	SMALLINT			Cloned instance number
TMP$TIMESTAMP	TIMESTAMP			Start time of procedure
TMP$USE_COUNT	SMALLINT			Statements compiled with procedure
TMP$QUANTUM	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Units of execution
TMP$INVOCATIONS	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Number of calls to procedure
TMP$PAGE_READS	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Page reads all database files
TMP$PAGE_WRITES	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Page writes all database files
TMP$PAGE_FETCHES	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Page fetches all database files
TMP$PAGE_MARKS	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Page marks all database files
TMP$RECORD_SELECTS	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Records selected by procedure
TMP$RECORD_INSERTS	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Records inserted by procedure
TMP$RECORD_UPDATES	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Records updated by procedure
TMP$RECORD_DELETES	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Records deleted by procedure
TMP$RECORD_PURGES	INTEGER	DOUBLE PRECISION	NUMERIC	Garbage collect record purges

TMP$PROCEDURES				
Column name	**Data type for <=ODS 15**	**Data type for >=ODS 16, dialect 1**	**Data type for >=ODS 16, dialect 3**	**Description**
			(18,0)	
TMP$RECORD_EXPUNGES	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Garbage collect record expunges
TMP$RECORD_BACKOUTS	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Garbage collect record backouts

TMP$RELATIONS

The TMP$RELATIONS table contains one row for each relation referenced since the current connection began.

TMP$RELATIONS				
Column name	**Data type for <=ODS 15**	**Data type for >=ODS 16, dialect 1**	**Data type for >=ODS 16, dialect 3**	**Description**
TMP$RELATION_ID	SMALLINT			Relation identifier
TMP$DATABASE_ID	INTEGER			Database identifier
TMP$RELATION_NAME	CHAR[67]			Relation name
TMP$USE_COUNT	SMALLINT			Statements compiled against relation
TMP$SWEEP_COUNT	SMALLINT			Database sweep or garbage collector
TMP$SCAN_COUNT	INTEGER			Sequential scans
TMP$FORMATS	SMALLINT			Number of relation formats
TMP$POINTER_PAGES	INTEGER			Number of relation pointer pages
TMP$DATA_PAGES	INTEGER			Number of relation data pages
TMP$GARBAGE_COLLECT_PAGES	INTEGER			Number of data pages to garbage collect
TMP$PAGE_READS	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Page reads all database files
TMP$PAGE_WRITES	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Page writes all database files
TMP$PAGE_FETCHES	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Page fetches all database files
TMP$PAGE_MARKS	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Page marks all database files
TMP$RECORD_IDX_SELECTS	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Records selected by index retrieval
TMP$RECORD_SEQ_SELECTS	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Records selected by sequential scan
TMP$RECORD_INSERTS	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Records inserted into relation
TMP$RECORD_UPDATES	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Records updated in relation

TMP$RELATIONS				
Column name	**Data type for <=ODS 15**	**Data type for >=ODS 16, dialect 1**	**Data type for >=ODS 16, dialect 3**	**Description**
TMP$RECORD_DELETES	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Records deleted from relation
TMP$RECORD_PURGES	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Garbage collect record purges
TMP$RECORD_EXPUNGES	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Garbage collect record expunges
TMP$RECORD_BACKOUTS	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Garbage collect record backouts

TMP$STATEMENTS

The TMP$STATEMENTS table contains one row for each statement currently executing for any current connection.

TMP$STATEMENTS				
Column name	**Data type for <=ODS 15**	**Data type for >=ODS 16, dialect 1**	**Data type for >=ODS 16, dialect 3**	**Description**
TMP$STATEMENT_ID	INTEGER			Statement identifier
TMP$ATTACHMENT_ID	INTEGER			Connection identifier
TMP$TRANSACTION_ID	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Transaction number
TMP$SQL	VARCHAR[4094]			SQL string
TMP$POOL_ID	INTEGER			Pool identifier
TMP$POOL_MEMORY	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Pool memory size
TMP$CLONE	SMALLINT			Cloned instance number
TMP$TIMESTAMP	TIMESTAMP			Start time of statement
TMP$QUANTUM	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Units of execution
TMP$INVOCATIONS	INTEGER			Number of calls to statement
TMP$STATE	CHAR[31]			ACTIVE,INACTIVE,STALLED,CANC
TMP$PRIORITY	CHAR[31]			Reserved
TMP$PAGE_READS	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Page reads all database files
TMP$PAGE_WRITES	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Page writes all database files
TMP$PAGE_FETCHES	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Page fetches all database files
TMP$PAGE_MARKS	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Page marks all database files
TMP$RECORD_SELECTS	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Records selected by statement
TMP$RECORD_INSERTS	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Records inserted by statement
TMP$RECORD_UPDATES	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Records updated by statement

TMP$STATEMENTS				
Column name	**Data type for <=ODS 15**	**Data type for >=ODS 16, dialect 1**	**Data type for >=ODS 16, dialect 3**	**Description**
TMP$RECORD_DELETES	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Records deleted by statement
TMP$RECORD_PURGES	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Garbage collect record purges
TMP$RECORD_EXPUNGES	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Garbage collect record expunges
TMP$RECORD_BACKOUTS	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Garbage collect record backouts

TMP$TRANSACTIONS

The TMP$TRANSACTIONS table contains one row for each transaction that is active or in limbo.

TMP$TRANSACTIONS				
Column name	**Data type for <=ODS 15**	**Data type for >=ODS 16, dialect 1**	**Data type for >=ODS 16, dialect 3**	**Description**
TMP$TRANSACTION_ID	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Transaction number
TMP$ATTACHMENT_ID	INTEGER			Connection identifier
TMP$POOL_ID	INTEGER			
TMP$POOL_MEMORY	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	
TMP$TIMESTAMP	TIMESTAMP			Start time of connection
TMP$SNAPSHOT	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Snapshot transaction number
TMP$QUANTUM	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Units of execution
TMP$SAVEPOINTS	INTEGER			savepoint number of records
TMP$READONLY	CHAR[1]			Transaction is read only
TMP$WRITE	CHAR[1]			Transaction has written data
TMP$NOWAIT	CHAR[1]			Transaction is no wait
TMP$COMMIT_RETAINING	CHAR[1]			Commit retaining performed
TMP$STATE	CHAR[31]			ACTIVE, LIMBO, COMMITTING, PRECOMMITTED
TMP$PRIORITY	CHAR			Reserved
TMP$TYPE	CHAR[31]			SNAPSHOT, READ_COMMITTED
TMP$PAGE_READS	INTEGER	DOUBLE PRECISION	NUMERIC	Page reads all database files

TMP$TRANSACTIONS				
Column name	**Data type for <=ODS 15**	**Data type for >=ODS 16, dialect 1**	**Data type for >=ODS 16, dialect 3**	**Description**
			(18,0)	
TMP$PAGE_WRITES	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Page writes all database files
TMP$PAGE_FETCHES	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Page fetches all database files
TMP$PAGE_MARKS	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Page marks all database files
TMP$RECORD_SELECTS	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Records selected by transaction
TMP$RECORD_INSERTS	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Records inserted by transaction
TMP$RECORD_UPDATES	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Records updated by transaction
TMP$RECORD_DELETES	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Records deleted by transaction
TMP$RECORD_PURGES	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Garbage collect record purges
TMP$RECORD_EXPUNGES	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Garbage collect record expunges
TMP$RECORD_BACKOUTS	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Garbage collect record backouts

TMP$TRIGGERS

The TMP$TRIGGERS table contains one row for each trigger executed since the current connection began.

TMP$TRIGGERS				
Column name	**Data Type for <=ODS 15**	**Data Type for >=ODS 16, dialect 1**	**Data Type for >=ODS 16, dialect 3**	**Description**
TMP$TRIGGER_ID	INTEGER			Trigger identifier
TMP$DATABASE_ID	INTEGER			Database identifier
TMP$RELATION_NAME	CHAR[67]			Relation name for trigger
TMP$TRIGGER_NAME	CHAR[67]			Trigger name
TMP$TRIGGER_TYPE	SMALLINT			The type of trigger being defined Values are: 1 - BEFORE INSERT 2 - AFTER INSERT 3 - BEFORE UPDATE 4 - AFTER UPDATE 5 - BEFORE DELETE 6 - AFTER DELETE
TMP$TRIGGER_SEQUENCE	SMALLINT			Sequence number for the trigger being defined; determines when a trigger is executed in relation to others of the same type. Triggers with the same sequence number execute in alphabetic order by trigger name. If this number is not assigned by the user, InterBase assigns a value of 0.
TMP$TRIGGER_ORDER	CHAR[31]			Position of the trigger
TMP$TRIGGER_OPERATION	CHAR[31]			***UPDATE***, ***DELETE*** or ***INSERT***
TMP$POOL_ID	INTEGER			Pool identifier
TMP$POOL_MEMORY	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Pool memory size
TMP$CLONE	SMALLINT			Cloned instance number
TMP$TIMESTAMP	TIMESTAMP			Start time of trigger
TMP$QUANTUM	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Units of Execution
TMP$INVOCATIONS	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Number of calls to trigger
TMP$PAGE_READS	INTEGER	DOUBLE PRECISION	NUMERIC	Page reads all database file

TMP$TRIGGERS				
Column name	**Data Type for <=ODS 15**	**Data Type for >=ODS 16, dialect 1**	**Data Type for >=ODS 16, dialect 3**	**Description**
			(18,0)	
TMP$PAGE_WRITES	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Page writes all database files
TMP$PAGE_FETCHES	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Page fetches all database files
TMP$PAGE_MARKS	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Page marks all database files
TMP$RECORD_SELECTS	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Records selected by trigger
TMP$RECORD_INSERTS	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Records inserted by trigger
TMP$RECORD_UPDATES	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Records updated by trigger
TMP$RECORD_DELETES	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Records deleted by procedure
TMP$RECORD_PURGES	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Garbage collect record purges
TMP$RECORD_EXPUNGES	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Garbage collect record expunges
TMP$RECORD_BACKOUTS	INTEGER	DOUBLE PRECISION	NUMERIC (18,0)	Garbage collect record backout

System Views

You can create a SQL script using the code provided in this section to create four views that provide information about existing integrity constraints for a database. You must create the database prior to creating these views. SQL system views are a subset of system views defined in the SQL-92 standard. Since they are defined by ANSI SQL-92, the names of the system views and their columns do not start with RDB$.

- The CHECK_CONSTRAINTS view:

```
CREATE VIEW CHECK_CONSTRAINTS (
CONSTRAINT_NAME,
CHECK_CLAUSE
) AS
SELECT RDB$CONSTRAINT_NAME, RDB$TRIGGER_SOURCE
FROM RDB$CHECK_CONSTRAINTS RC, RDB$TRIGGERS RT
WHERE RT.RDB$TRIGGER_NAME = RC.RDB$TRIGGER_NAME;
```

- The CONSTRAINTS_COLUMN_USAGE view:

```
CREATE VIEW CONSTRAINTS_COLUMN_USAGE (
TABLE_NAME,
COLUMN_NAME,
CONSTRAINT_NAME
) AS
SELECT RDB$RELATION_NAME, RDB$FIELD_NAME, RDB$CONSTRAINT_NAME
FROM RDB$RELATION_CONSTRAINTS RC, RDB$INDEX_SEGMENTS RI
WHERE RI.RDB$INDEX_NAME = RC.RDB$INDEX_NAME;
```

- The REFERENTIAL_CONSTRAINTS view:

```
CREATE VIEW REFERENTIAL_CONSTRAINTS (
CONSTRAINT_NAME,
UNIQUE_CONSTRAINT_NAME,
MATCH_OPTION,
UPDATE_RULE,
DELETE_RULE
) AS
SELECT RDB$CONSTRAINT_NAME, RDB$CONST_NAME_UQ, RDB$MATCH_OPTION,
RDB$UPDATE_RULE, RDB$DELETE_RULE
FROM RDB$REF_CONSTRAINTS;
```

- The TABLE_CONSTRAINTS view:

```
CREATE VIEW TABLE_CONSTRAINTS (
CONSTRAINT_NAME,
TABLE_NAME,
CONSTRAINT_TYPE,
IS_DEFERRABLE,
INITIALLY_DEFERRED
) AS
SELECT RDB$CONSTRAINT_NAME, RDB$RELATION_NAME,
```

```
RDB$CONSTRAINT_TYPE, RDB$DEFERRABLE, RDB$INITIALLY_DEFERRED
FROM RDB$RELATION_CONSTRAINTS;
```

CHECK CONSTRAINTS

CHECK_CONSTRAINTS identifies all CHECK constraints defined in the database.

CHECK_CONSTRAINTS			
Column name	**Data type**	**Length**	**Description**
CONSTRAINT_NAME	CHAR	67	Unique name for the CHECK constraint; nullable
CHECK_CLAUSE	BLOB		Subtype Text: Nullable; original source of the trigger definition, stored in the RDB$TRIGGER_SOURCECOLUMN in RDB$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.

CONSTRAINTS_COLUMN_USAGE			
Column name	**Data type**	**Length**	**Description**
TABLE_NAME	CHAR	67	Table for which the constraint is defined; nullable
COLUMN_NAME	CHAR	67	Column used in the constraint definition; nullable
CONSTRAINT_NAME	CHAR	67	Unique name for the constraint; nullable

REFERENTIAL CONSTRAINTS

REFERENTIAL_CONSTRAINTS identifies all referential constraints defined in a database.

REFERENTIAL_CONSTRAINTS			
Column name	**Data type**	**Length**	**Description**
CONSTRAINT_NAME	CHAR	67	Unique name for the constraint; nullable
UNIQUE_CONSTRAINT_NAME	CHAR	67	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
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_CONSTRAINTS

Column name	Data type	Length	Description
CONSTRAINT_NAME	CHAR	67	Unique name for the constraint; nullable
TABLE_NAME	CHAR	67	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

Change Views

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

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

Using Change Views

See Getting Started with Change Views for a complete explanation of these topics:

- OSD Platform Updates
- Migration Issues and Dependencies
- Requirements and Constraints
- Requirements
- Constraints
- Backup/Restore Considerations
- Deferred Constraints Checking
- Trigger Inactivation

Database Restore from a Backup

Creating Subscriptions to Change Views

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

In creating subscriptions you would

Grant Subscribe: Grants the user subscribe privileges

Set Subscription: To set a subscription as active, an application issues a ***SET SUBSCRIPTION*** statement. The ***SET SUBSCRIPTION*** statement allows multiple subscriptions to be activated and includes an AT clause to denote

a destination or device name as a recipient of subscribed changes. The subscriber user name is implied by the user identity of the database connection.

See Creating Subscriptions to Change Views or a complete explanation and examples of how to create subscriptions.

Statement Execution

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

See Statement Execution for a complete explanation of how the Statement Execution feature works.

Change View API Support

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

See Change Views API Support or a complete explanation of how the Statement Execution feature works.

Change View SQL Language Support

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

See Change Views SQL Language Support for examples showing a retooling of the ISQL command-line utility that supports change views.

Metadata Support

Subscription information is stored in a new system relation RDB$SUBSCRIPTIONS with a unique key on RDB$SUBSCRIPTION_NAME, RDB$SUBSCRIBER_NAME, RDB$DESTINATION. Additional fields store control information to facilitate "check in" and "check out" of changed data. This includes transaction IDs, timestamps and transactional context of last observation of changed data on the schema object.

- The term "check out" denotes ***SELECT*** of changed columns of rows from subscribed tables when a subscription has been activated.
- The term "check in" refers to ***INSERT***, ***UPDATE*** and ***DELETE*** of changed columns of rows from subscribed tables when a subscription has been activated.
- A subscription becomes activated during a database session with the execution of ***SET SUBSCRIPTION ACTIVE***.
- It is deactivated with the execution of ***SET SUBSCRIPTION INACTIVE***.

RDB$SUBSCRIPTION and RDB$SUBSCRIBERS are new tables covering the subscription/subscriber elements. The other tables listed show columns that have been updated or added to an existing table.

For more information on the new and updated columns for the implementation of the Change View feature see Metadata Support.

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 chapter lists available character sets and their corresponding collation orders and describes how to specify:

- Default character set for an entire database.
- Alternative character set and collation order 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 value in a comparison operation.
- Collation order in an ***ORDER BY*** or ***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 character 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 default collation of the set, the one that is used if no COLLATE clause specifies an alternative order.

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
BIG_5	56	2 bytes	1 byte	BIG_5
CYRL	50	1 byte	1 byte	CYRL DB_RUS PDOX_CYRL
DOS437	10	1 byte	1 byte	DOS437 DB_DEU437 DB_ESP437 DB_FIN437 DB_FRA437 DB_ITA437

Character sets and collation orders				
Character set	**Char. set ID**	**Max. char. size**	**Min. char. size**	**Collation orders**
				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
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

Character sets and collation orders				
Character set	**Char. set ID**	**Max. char. size**	**Min. char. size**	**Collation orders**
DOS861	47	1 byte	1 byte	DOS861 PDOX_ISL
DOS863	14	1 byte	1 byte	DOS863 DB_FRC863
DOS865	12	1 byte	1 byte	DOS865 DB_DAN865 DB_NOR865 PDOX_NORDAN4
EUCJ_0208	6	2 bytes	1 byte	EUJC_0208
GB_2312	57	2 bytes	1 byte	GB_2312
ISO8859_1	21	1 byte	1 byte	ISO8859_1 CC_ESPLAT1 CC_PTBRLAT1 DA_DA DE_DE DU_NL EN_UK EN_US ES_ES FI_FI FR_CA FR_FR IS_IS IT_IT NO_NO PT_PT SV_SV
ISO8859_2	22	1 byte	1 byte	ISO8859_2 CS_CZ PL_PL
ISO8859_15	39	1 byte	1 byte	ISO8859_15

Character sets and collation orders				
Character set	**Char. set ID**	**Max. char. size**	**Min. char. size**	**Collation orders**
				DA_DA9 DE_DE9 DU_NL9 EN_UK9 EN_US9 ES_ES9 FI_FI9 FR_CA9 FR_FR9 IS_IS9 IT_IT9 NO_NO9 PT_PT9 SV_SV9
KO18R	58	1 byte	1 byte	RU_RU
KSC_5601	44	2 bytes	1 byte	KSC_5601 KSC_DICTIONARY
NEXT	19	1 byte	1 byte	NEXT NXT_DEU NXT_FRA NXT_ITA NXT_US
NONE	0	1 byte	1 byte	NONE
OCTETS	1	1 byte	1 byte	OCTETS
SJIS_0208	5	2 bytes	1 byte	SJIS_0208
UNICODE_BE UCS2BE	8	2 bytes	2 bytes	N/A at this time
UNICODE_FSS	3	3 bytes	1 byte	UNICODE_FSS
UNICODE_LE UCS2LE	64	2 byte	2 bytes	N/A
UTF_8	59	4 byte	1 bytes	N/A at this time.
WIN1250	51	1 byte	1 byte	WIN1250 PXW_CSY

Character sets and collation orders				
Character set	**Char. set ID**	**Max. char. size**	**Min. char. size**	**Collation orders**
				PXW_HUNDC PXW_PLK PXW_SLOV
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 InterBase restricts the maximum amount of storage in each field in the column to 32,767 bytes for CHAR columns and 32,765 for VARCHAR columns. In the case of a single-byte character column, one character is stored in one byte, so you can define 32,767 (or 32,765 for VARCHAR) characters per single-byte column without encountering an error.

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

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

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

Support for Paradox and dBASE

Many character sets and their corresponding collations are provided to support Paradox for DOS, Paradox for Windows, dBASE for DOS, and dBASE for Windows.

Character Sets for DOS (Support for Paradox and dBASE)

The following character sets correspond to MS-DOS code pages, and should be used to specify character sets for InterBase databases that are accessed by Paradox for DOS and dBASE for DOS:

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

The names of collation orders for these character sets that are specific to Paradox begin "PDOX". For example, the DOS865 character set for DOS code page 865 supports a Paradox collation order for Norwegian and Danish called "PDOX_NORDAN4".

The names of collation orders for these character sets that are specific to dBASE begin "DB". For example, the DOS437 character set for DOS code page 437 supports a dBASE collation order for Spanish called -"DB_ESP437".

For more information about DOS code pages, and Paradox and dBASE collation orders, see the appropriate Paradox and dBASE documentation and driver books.

Character Sets for Microsoft Windows (Support for Paradox and dBASE)

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 WIN1252 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 Character Sets

This section provides details on how to specify character sets. Specifically, it covers how to specify the following:

- The default character set for a database
- A character set for a table column
- The character set for a client attachment
- The collation order for a column
- The collation order in comparisons
- The collation order for ***ORDER BY*** and ***GROUP BY*** clauses

Default Character Set for a Database

A database's default character set designation specifies the character set the server uses to tag ***CHAR***, ***VARCHAR***, and text Blob columns in the database when no other character set information is provided. When data is stored in such columns without additional character set information, the server uses the tag to determine how to store and transliterate that data. A default character set should always be specified for a database when it is created with ***CREATE DATABASE***.

To specify a default character set, use the ***DEFAULT CHARACTER SET*** clause of ***CREATE DATABASE***. For example, the following statement creates a database that uses the ISO8859_1 character set:

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

IMPORTANT

If you do not specify a character set, the character set defaults to ***NONE***. Using character set ***NONE*** means that there is no character set assumption for columns; data is stored and retrieved just as you originally entered it. You can load any character set into a column defined with ***NONE***, but you cannot later move that data into another column that has been defined with a different character set. In this case, no transliteration is performed between the source and destination character sets, and errors may occur during assignment.

For the complete syntax of ***CREATE DATABASE***, see CREATE DATABASE.

Character Set for a Column in a Table

Character sets for individual columns in a table can be specified as part of the column's CHAR or VARCHAR data type definition. When a character set is defined at the column level, it overrides the default character set declared for the database. For example, the following isql statements create a database with a default character set of ISO8859_1, then create a table where two column definitions include a different character set specification:

```
CREATE DATABASE 'europe.ib' DEFAULT CHARACTER SET ISO8859_1;
CREATE TABLE RUS_NAME(
LNAME VARCHAR(30) NOT NULL CHARACTER SET CYRL,
FNAME VARCHAR(20) NOT NULL CHARACTER SET CYRL,
);
```

For the complete syntax of CREATE TABLE, see CREATE TABLE.

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.ib' USER 'JAMES' PASSWORD 'U4EEAH';
```

For the complete syntax of ***SET NAMES***, see SET NAMES (Reference). For the complete syntax of ***CONNECT***, see CONNECT.

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.

Collation Order in Comparison

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.

Collation Order in ORDER BY

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.

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.