

OBJECTVIEW[®]

GUPTA'S
SQLBase
SQLTalk

Language
Reference Manual

Trademarks

Quest, SQLBase, SQLGateway, SQLHost, SQLRouter, and SQLWindows are registered trademarks of Gupta Technologies, Inc.

Express Edit, Express Form, Express Master/Detail, Express Table, Express Windows, SQL/API, SQLNetwork, and SQLTalk are trademarks of Gupta Technologies, Inc.

IBM, IBM PC, PC/AT, PC-DOS, and PS/2 are registered trademarks of International Business Machines Corporation. Token-Ring, DB2, and OS/2 are trademarks of International Business Machines Corporation.

Microsoft and MS-DOS are registered trademarks of Microsoft Corporation.

3Com and 3+ are registered trademarks of 3Com Corporation. 3+Open and 3Plus are trademarks of 3Com Corporation.

Ethernet is a registered trademark of Xerox Corporation.

PC/TCP and FTP Software are registered trademarks of FTP Software, Inc.

Sun Microsystems is a registered trademark of Sun Microsystems, Inc. SPARC, SunOS, PC-NFS, and the combination of Sun with a numeric suffix are trademarks of Sun Microsystems, Inc.

UNIX is a registered trademark of AT&T.

Copyright

Copyright © 1985-1991 by Gupta Technologies, Inc. All rights reserved.
SQLTalk Reference Manual

November 1991

Authors: Bruce Ring, Rick Cumings, and Stacia Sambar

For more information on this or other products, please contact:
Gupta Technologies, Inc.

Contents

PART THREE: SQL	193
Chapter 5: Introduction to SQL	195
About this Chapter	196
SQL Command Categories	197
Data Definition Language (DDL).....	198
Data Manipulation Language (DML).....	198
Data Query Language (DQL).....	198
Transaction Control	199
Database Administration	199
SQL Commands	200
Why use SQL?	201
Relational Operators.....	201
Recursion.....	201
Joins	202
Conventions.....	203
Chapter 6: SQL Elements	205
About this Chapter	206
Names	207
Ordinary Identifier	207
Delimited Identifier	207
Long and Short Identifiers.....	208
Examples of Names	208
Types of Names	209
Authorization-id.....	209
Column Name	209
Correlation Name	210
Database Name.....	210
Index Name	210
Password.....	210
Bind Variable Name.....	211
Command Name	211
Synonym Name	211
Table Name	211
View Name	212
Summary of Naming Requirements	213
Data Types	214

Null Values.....	214
Character Data Types.....	215
CHAR (or VARCHAR)	215
LONG VARCHAR (or LONG)	215
Number Data Types.....	217
NUMBER	218
DECIMAL (or DEC).....	218
INTEGER (or INT)	220
SMALLINT	220
DOUBLE PRECISION	220
FLOAT	221
REAL	221
Date/Time Data Types.....	222
DATETIME (or TIMESTAMP)	223
DATE	223
TIME	224
Data Type Conversions	225
Data Type Conversions in Assignments.....	225
Data Type Conversions in Functions.....	225
Constants.....	226
String Constants	226
Numeric Constants	226
Date/Time Constants	226
Examples of Constants.....	227
System Keywords	228
Examples of System Keywords	228
Expressions	229
Null Values in Expressions	230
String Concatenation Operator ().....	230
Precedence.....	231
Examples of Expressions	231
Search Conditions	232
Nulls and Search Conditions	233
Examples of Search Conditions	234
Predicates.....	235
Relational Predicate.....	236
Comparison Relational Predicate	236
Quantified Relational Predicate	237
Examples of SubSELECTs and Subqueries	237
BETWEEN Predicate.....	238
Example.....	238

NULL Predicate	238
Example.....	238
EXISTS Predicate.....	239
Example.....	239
LIKE Predicate.....	239
Examples.....	240
IN Predicate.....	241
Examples.....	241
Functions.....	242
Date/Time Values.....	243
Entering Date/Time Values.....	243
Examples.....	244
Date/Time System Keywords.....	245
Resolution for Time Keywords.....	246
Example of Date/Time System Keyword.....	246
Time Zones.....	246
Date/Time Expressions.....	247
Examples of Date/Time Expressions.....	248
Joins.....	249
Equijoin.....	249
Cartesian Product.....	250
Outer Join.....	250
Self Join.....	251
Non-Equijoin.....	252
Subqueries.....	253
Examples of Subqueries.....	253
Bind Variables.....	254
Chapter 7: SQL Command Reference.....	255
About This Chapter.....	256
SQL Command Summary.....	257
ALTER DATABASE.....	260
ALTER DBAREA.....	262
ALTER PASSWORD.....	263
ALTER STOGROUP.....	264
ALTER TABLE.....	266
ALTER TABLE (Referential Integrity).....	271
ALTER TABLE (Error Messages).....	276
CHECK DATABASE.....	279
COMMENT ON.....	280
COMMIT.....	282

CREATE DATABASE	284
CREATE DBAREA	287
CREATE INDEX	289
CREATE STOGROUP	297
CREATE SYNONYM	298
CREATE TABLE	300
CREATE VIEW	308
DEINSTALL DATABASE	312
DELETE	313
DROP	316
DROP DATABASE	318
DROP DBAREA	319
DROP STOGROUP	320
GRANT (Database Authority)	321
GRANT (Table Privileges)	325
INSERT	329
INSTALL DATABASE	334
LABEL	335
REVOKE (Database Authority)	338
REVOKE (Table Privileges)	341
ROLLBACK	344
SAVEPOINT	346
SELECT	350
UNION Clause	359
SET DEFAULT STOGROUP	361
UPDATE	362
UPDATE STATISTICS	366
Chapter 8: SQL Function Reference	369
About This Chapter	370
Data Type Conversions in Functions	370
Aggregate Functions	371
String Functions	372
Date/Time Functions	373
Math Functions	374
Finance Functions	375
Logical Functions	375
Special Functions	376
SQLBase Function Summary	377
AVG	381
COUNT	382

MAX.....	383
MIN.....	384
SUM.....	385
@ABS.....	386
@ACOS.....	386
@ASIN.....	387
@ATAN.....	387
@ATAN2.....	388
@CHAR.....	388
@CHOOSE.....	389
@CODE.....	390
@COS.....	390
@CTERM.....	391
@DATE.....	391
@DATETOCHAR.....	392
@DATEVALUE.....	392
@DAY.....	393
@DECIMAL.....	393
@DECODE.....	394
@DECRYPT.....	395
@EXACT.....	396
@EXP.....	397
@FACTORIAL.....	397
@FIND.....	398
@FV.....	399
@HEX.....	400
@HOUR.....	400
@IF.....	401
@INT.....	401
@ISNA.....	402
@LEFT.....	402
@LENGTH.....	403
@LICS.....	404
@LN.....	409
@LOG.....	409
@LOWER.....	410
@MEDIAN.....	410
@MICROSECOND.....	412
@MID.....	412
@MINUTE.....	413
@MOD.....	413

@MONTH.....	414
@MONTHBEG.....	414
@NOW.....	415
@NULLVALUE.....	415
@PI.....	416
@PMT.....	416
@PROPER.....	417
@PV.....	418
@QUARTER.....	419
@QUARTERBEG.....	419
@RATE.....	420
@REPEAT.....	421
@REPLACE.....	421
@RIGHT.....	422
@ROUND.....	423
@SCAN.....	424
@SDV.....	425
@SECOND.....	426
@SIN.....	426
@SLN.....	427
@SQRT.....	428
@STRING.....	428
@SUBSTRING.....	429
@SYD.....	430
@TAN.....	431
@TERM.....	432
@TIME.....	433
@TIMEVALUE.....	433
@TRIM.....	434
@UPPER.....	434
@VALUE.....	435
@WEEKBEG.....	435
@WEEKDAY.....	436
@YEAR.....	437
@YEARBEG.....	437
@YEARNO.....	438
Chapter 9: SQL Reserved Words.....	439
About this Chapter.....	440
Chapter 10: System Catalog.....	443

About this Chapter	444
System Catalog Summary	445
System Table Operations	446
System Catalog Views	446
SYSADM.SYSCOLAUTH	447
SYSADM.SYSCOLUMNS	448
SYSADM.SYSCOMMANDS	450
SYSADM.SYSFKCONSTRAINTS	451
SYSADM.SYSINDEXES	452
SYSADM.SYSKEYS	453
SYSADM.SYSPKCONSTRAINTS	454
SYSADM.SYSSYNONYMS	455
SYSADM.SYSTABAUTH	456
SYSADM.SYSTABCONSTRAINTS	458
SYSADM.SYSTABLES	460
SYSADM.SYSUSERAUTH	461
SYSADM.SYSVIEWS	462
Server-Independent System Catalog Views	463
Chapter 11: Referential Integrity	465
About this Chapter	466
About Referential Integrity	466
What is a Primary Key?	468
What is a Foreign Key?	469
Parent and Dependent Tables and Rows	471
How to Create Tables with Referential Constraints	471
Creating a UNIQUE Index	472
Deleting from Tables	473
DELETE RESTRICT	473
DELETE CASCADE	473
DELETE SET NULL	474
Implications for SQLBase Operations	475
INSERT Implications	475
UPDATE Implications	475
DROP Implications	476
Dropping a Primary Key	476
Dropping a Foreign Key	477
Cycles of Dependent Tables	478
Restrictions on Self-Referencing Tables	481
Delete-Connected Table Restrictions	482
Customizing SQLBase Error Messages	484

Contents

Editing the Error Messages	485
Primary Key Error Messages	486
Foreign Key Error Messages.....	487

PART THREE

SQL

Chapter 5

Introduction to SQL

About this Chapter

SQL (Structured Query Language) is used to manage relational databases. SQL is pronounced "*ess-que-el*" or "*sequel*."

You can use the SQL commands documented in this part of the manual at the command line prompt in SQLTalk.

You can also use the SQL commands documented here with these Gupta products:

- SQLTalk for Windows.
- SQLWindows.
- C/API.
- SQLPrecompiler (COBOL).
- SQLNetwork gateways.

Consult the manual for the specific product you are using for more information.

SQL Command Categories

SQL has commands that let you perform various operations. The operations are grouped into these categories.

Data Definition Language (DDL)	Create database objects such as tables or views.
Data Manipulation Language (DML)	Update, add, or delete data in the database.
Data Query Language (DQL)	Ask questions about data in the database.
Transaction Control	Ensure data integrity when making changes to the database.
Database Administration	Control security and system administration.

Each of these categories is explained next.

Data Definition Language (DDL)

Commands which define the structure of a table, view, or other database objects are called data definition commands.

```
ALTER TABLE
CREATE INDEX
CREATE TABLE
CREATE VIEW
DROP
```

Data Manipulation Language (DML)

Data manipulation commands add, change, and delete data in the database.

```
DELETE
INSERT
UPDATE
```

Data Query Language (DQL)

Queries are used to find information stored in the database. The result of a query is a table with rows and columns.

```
SELECT
```


Transaction Control

Transaction control commands ensure that a logically-related sequence of actions that accomplish a particular result in an application (a logical unit of work) are performed in their entirety or cancelled in their entirety. This ensures data integrity.

**COMMIT
ROLLBACK
SAVEPOINT**

Database Administration

These commands:

- Create and maintain databases.
- Create and maintain partitions.
- Assign users to databases and tables.

**ALTER DATABASE
ALTER DBAREA
ALTER PASSWORD
ALTER STOGROUP
CHECK DATABASE
CREATE DATABASE
CREATE DBAREA
CREATE STOGROUP
DEINSTALL DATABASE
DROP DATABASE
DROP DBAREA
DROP STOGROUP
GRANT
INSTALL DATABASE
REVOKE
SET DEFAULT STOGROUP
UPDATE STATISTICS**

SQL Commands

An example of a SQL query is shown below as it would be phrased in conversational English and as it is coded in a SELECT command.

English	Give me a list of everyone who works at the Albany location who has the same job as someone who works at the Utica location.
SQL	<pre>SELECT NAME, EMP_ID FROM EMP WHERE LOC = 'ALBANY' AND JOB IN (SELECT JOB FROM EMP WHERE LOC = 'UTICA')</pre>

Some other examples of SQL commands are:

```
SELECT NAME FROM EMP;
```

```
CREATE TABLE FRIENDS (NAME CHAR(15));
```

```
SELECT * FROM EMP, DEPT WHERE EMP.DEPTNO=DEPT.DEPTNO;
```

```
ALTER TABLE FRIENDS RENAME TABLE FOLKS;
```

```
DROP TABLE FOLKS;
```

Why use SQL?

SQL lets you build complex queries. These features make complex queries possible:

- Relational operators
- Recursion
- Joins

Relational Operators

Relational operators (such as $>$, $<$, $=$, $>=$, $<=$, or $<>$) enable you to express a search condition for a query.

For example, if a table contains employee information (name, address, phone, age, salary, department, date hired), the following two questions can be answered with SQL using relational operators.

Get Macbeth's phone number.

```
SELECT NAME, PHONE FROM EMP WHERE NAME = 'MACBETH'
```

List all employees whose salary is greater than \$5000.

```
SELECT NAME, SAL FROM EMP WHERE SAL > 5000
```

Recursion

SQL is recursive. The input to one query can be the output of another query. A query can be nested within another SQL command to define the scope of a command.

The following example illustrates SQL's ability to nest queries.

```
SELECT NAME, EMP_ID
FROM EMP
WHERE LOC = 'ALBANY'
AND JOB IN
(SELECT JOB FROM EMP WHERE LOC = 'UTICA')
```

The second SELECT in the example above is called a *subselect*.

Joins

A join pulls data from different tables and compares it by matching on a common row that is in all the tables.

The primary key for a table is a value that has a match in another table. For example, a CUSTOMERS table is shown below that contains these columns: name, address, and credit rating. Also, each customer has a unique identifying number.

<u>CUSTOMERS</u>			
<u>ID</u>	<u>CUSTNAME</u>	<u>ADDRESS</u>	<u>CREDIT RATING</u>
1	ABC INC.	13 A St.	5
2	XYZ INC.	1 B St.	4
3	A1 INC.	12 C St.	3

There is another table called CONTACTS that contains the name, phone number, department of each contact. The table also includes a key that contains the customer number. This is the same number that is in the CUSTOMERS table.

<u>CONTACTS</u>			
<u>ID</u>	<u>NAME</u>	<u>PHONE</u>	<u>DEPT</u>
1	Mary	813	SALES
1	Joe	915	DP
1	Frank	
2	Jean	
3	Lou	

You can join customer information with contact information without unnecessary repetition of data.

The following SQL command uses these tables to find the name and phone number of the person at company ABC who is the contact for sales.

```
SELECT NAME, PHONE
FROM CUSTOMERS, CONTACTS
WHERE CUSTOMERS.ID = CONTACTS.ID
AND CUSTNAME = 'ABC INC.' AND DEPT = 'SALES';
```

The result of the query is:

<u>NAME</u>	<u>PHONE</u>
Mary	813

Conventions

This part of the manual uses the same conventions as the rest of the *SQLTalk Reference Manual* to show commands. The SQLTalk command prompt (SQL>) is shown and each command is terminated with a semicolon (;).

If you are using SQL from an interface other than SQLTalk, the prompt is not relevant. Also, other interfaces have a different way to terminate a command. See the manual for the interface that you are using for details.

Chapter 6

SQL Elements

About this Chapter

These elements of SQL are described in this chapter:

- Names.
- Data types.
- Constants.
- System keywords.
- Functions.
- Expressions.
- Predicates.
- Search conditions.
- Bind variables.

Names

A name is called an *identifier* in SQL. Usernames, table names, column names, and index names are examples of identifiers.

An identifier can be an *ordinary identifier* or a *delimited identifier*.

Ordinary Identifier

An ordinary identifier begins with a letter or one of the special characters (#, @ or \$) and can include letters, numeric digits and the underscore (_).

Delimited Identifier

A delimited identifier can contain *any* character including special characters such as blanks and periods. Also, a delimited identifier can start with a digit.

Delimited identifiers must be enclosed in double quotes:

```
"7.g identifier"
```

There is a precaution to using a delimited identifier: the identifier is case sensitive.

SQLTalk and SQL reserved words can be used as identifiers if they are delimited, but this is *not* recommended.

If a delimited identifier contains double quotes, then two consecutive double quotes ("") are used to represent one double quote (").

Long and Short Identifiers

Identifiers can be long or short. The maximum length of a long data identifier is 18 characters. The maximum length of a short data identifier is 8 characters.

Identifier Type	Length
Long	18
Short	8

Examples of Names

Examples of names are:

```
CHECKS  
AMOUNT_OF_$  
:CHKNUM  
$500  
"NAME & NO."  
#CUSTOMER  
:3
```

Types of Names

Names are long or short identifiers, or identifiers qualified by other identifiers. The following objects have names.

Authorization-id

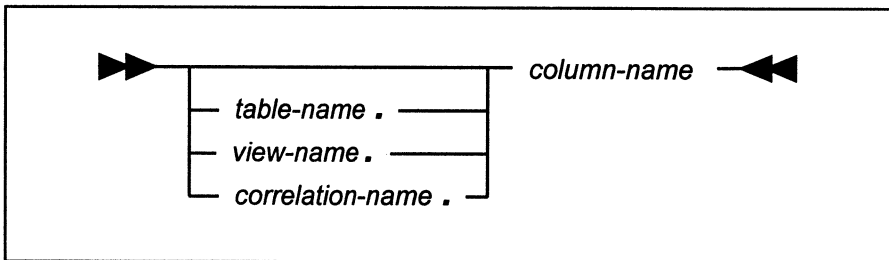
A short identifier that designates a user. Authorization-id is also called *username* in this manual. The system keyword USER contains the username.

An authorization-id is an implicit part of all database object names. To name a database object explicitly, add the authorization-id and a period to the beginning of the identifier. For example, the table name CUST created by user JOE has the explicit name JOE.CUST. The implicit name CUST is used most often.

Examples of authorization-ids are JOE and USER1.

Column Name

A qualified or unqualified long identifier that names a column of a table or view.



The qualified form is preceded by a table name, a view name, or correlation name and a period (.).

Examples of column names are EMPNO and EMPLOYEES.EMPNO.

Correlation Name

A long identifier that designates a table or view within a command.

Examples of correlation names are X and TEMP.

Database Name

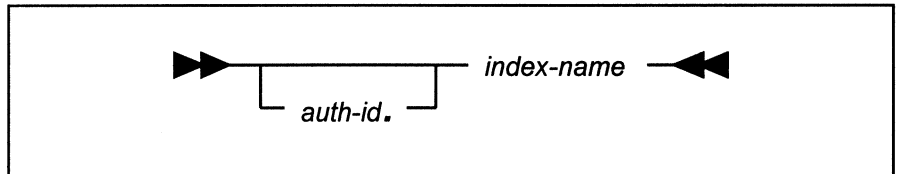
A short identifier that designates a database.

Do not specify an extension for a database name (demo.xyz is invalid). SQLBase automatically assigns a database name extension of *.dbs*. SQLBase will store a database called DEMO in a file named DEMO.DBS.

Examples of database names are DEMO and COMPANY.

Index Name

A qualified or unqualified long identifier that names an index.



The qualified form is preceded by an authorization-id and a period.

An unqualified index name is implicitly qualified by the authorization-id of the user who gave the command.

Examples are EMPX and JOE.EMPX.

Password

A short identifier that is a password for an authorization-id.

Examples are PWD1 and X2381.

Bind Variable Name

Bind variable names in a SQL command must always be ordinary identifiers or digits preceded by a colon (:).

Command Name

A long identifier that designates a user-defined command.

An example is QUERY1.

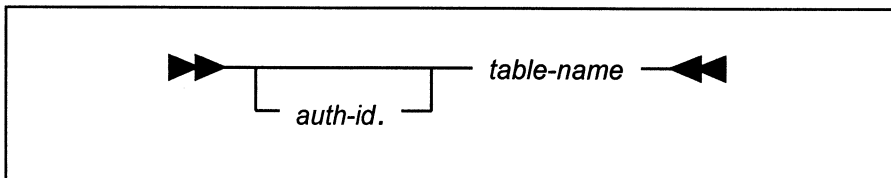
Synonym Name

A long identifier that designates a table or view. A synonym can be used wherever a table name or view name can be used to refer to a table or view.

An example of a synonym is EASY.

Table Name

A qualified or unqualified long identifier that names a table.



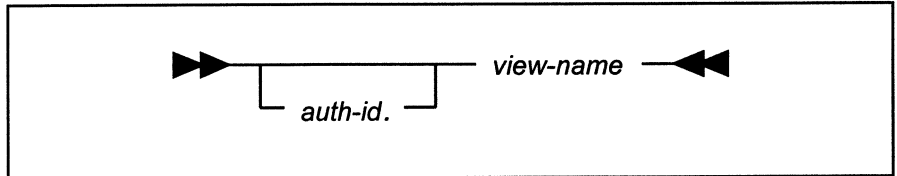
An unqualified table name is implicitly qualified by the authorization-id of the user who created the table.

The qualified form is preceded by an authorization-id and a period.

Examples are EMP and JOE.EMP.

View Name

A qualified or unqualified long identifier that designates a view.



An unqualified view name is implicitly qualified by the authorization-id of the user who gave the command.

The qualified form is preceded by an authorization-id and a period.

Examples of view names are MYEMP and DEPT10.MYEMP.

Summary of Naming Requirements

Type of Identifier	Maximum Length	Qualifiable?
Authorization-id	8	No
Column	18	Yes
Correlation	18	No
Database	8	No
Index	18	Yes
Password	8	No
Command	18	No
Synonym	18	No
Table	18	Yes
View	18	Yes

Data Types

The general data types that SQLBase uses to store data are:

- Character.
- Number.
- Date and time.

The data type determines:

- The value and length of the data as stored in the database.
- The display format when the data is displayed.

The data type for a column is specified in the CREATE TABLE command.

Null Values

A *null value* indicates the absence of data. Any data type can contain a null value.

Null is *not* equivalent to zero or to blank; it is the same as *unknown*. A value of null is *not* greater than, less than, or equivalent to any other value, including another value of null. To retrieve a field on a null match, the NULL predicate must be used.

To understand more about how nulls are treated, see the section about *Search Conditions* in this chapter.

Character Data Types

A character string is a sequence of letters, digits, or special characters. All character data is stored in SQLBase as variable-length strings.

For DB2 SQL compatibility, SQLBase allows several alternative keywords to declare the same data type.

CHAR (or VARCHAR)

A length *must* be specified for this data type. The length determines the maximum length of the string. The length cannot exceed 254 bytes.

CHAR columns can be used in comparison operations with other characters or numbers and can occur in most functions and expressions. Wild-card search operators can be used in the LIKE predicate for character-only comparisons.

This data type is defined in the system catalog as VARCHAR.

Examples:

```
CHAR (11)
VARCHAR (25)
CHAR (10)
```

LONG VARCHAR (or LONG)

A length attribute is not specified for this data type.

This data type stores strings of any length (including strings longer than 254 bytes).

Both text and binary data can be stored in LONG VARCHARs. However, only character data can be retrieved through SQLTalk.

LONG VARCHAR columns can be stored, retrieved, or modified, but cannot be used in a comparison operation in a WHERE clause. LONG VARCHAR columns cannot be used in expressions or in most functions.

The name LONG VARCHAR and LONG are synonymous.

Example:

LONG VARCHAR

Number Data Types

SQLBase allows these numeric data types:

Exact Data Types	NUMBER DECIMAL (or DEC) INTEGER (or INT) SMALLINT
Approximate Data Types	FLOAT REAL DOUBLE PRECISION

SQLBase uses its own internal representation of numbers described in the *C/API Reference Manual*. Data is cast on input and output to conform to the restrictions of the data type.

Precision and scale are maintained internally by SQLBase:

- Precision refers to the total number of allowable digits
- Scale refers to the number of digits to the right of the decimal point.

Numbers with up to 15 decimal digits of precision can be stored in the exact data types. Numbers in the range of 1.0e-99 to 1.0e+99 can be stored in the approximate data types.

SQLBase supports integer arithmetic. For example:

```
INTEGER / INTEGER = INTEGER
```

Number columns can be used in any comparison operations with other numbers and can occur in all functions and expressions.

NUMBER

This data type is not associated with a particular scale or precision. Any number up to 15 digits of precision can be stored and retrieved.

Example:

NUMBER

If you need as much precision as possible, use the NUMBER data type.

DECIMAL (or DEC)

This data type is associated with a particular scale and precision. Scale is the number of fractional digits and precision the total number of digits. If precision and scale are not specified, SQLBase uses a default precision and scale of 5,0.

A maximum of 15 digits can be stored in this data type. The valid range is:

-999999999999999 to +999999999999999

Another way to express the range is to say that the value can be $-n$ to $+n$, where the absolute value of n is the largest number that can be represented with the applicable precision and scale.

The DEC notation is compatible with DB2.

Examples:

DECIMAL (8, 2)
DECIMAL (5, 0) (same as INTEGER)
DECIMAL
DEC

Input values are rounded to the precision of the column definition. For example:

Entering 29.994 in a DECIMAL(10,2) stores 29.99.

Entering 29.995 in a DECIMAL(10,2) stores 30.00.

For division, the precision and scale of the result are determined as follows:

precision of result	=	maximum precision of SQLBase (15)						
scale of result	=	maximum precision	—	precision of first input number	+	scale of first input number	—	scale of second input number

For example, if you have two columns as defined below:

```
D1    DECIMAL (10,2)
D2    DECIMAL (10,2)
```

and you divide D1 by D2, then:

```
precision = 15
scale     15-10+2-2=5
=
```

Some functions change the maximum precision. For example, SUM changes the maximum precision to 15. Therefore:

```
SUM(D1)/SUM(D2)
```

means:

```
precision = 15
scale     15-15+2-2=0
=
```

INTEGER (or INT)

This data type has no fractional digits. Digits to the right of the decimal point are truncated.

An INTEGER can have up to 10 digits of precision:

-2147483648 to +2147483647

The INT notation is compatible with DB2.

Examples:

INTEGER
INT

SMALLINT

This data type has no fractional digits. Digits to the right of the decimal point are truncated.

A SMALLINT can have up to 5 digits of precision:

-32768 to +32767

SQLBase does not store a SMALLINT value relative to the size of a 16-bit integer, but approximates it with the same number of digits. C programmers should check for overflow.

Example:

SMALLINT

DOUBLE PRECISION

This data type specifies a column containing double-precision floating point numbers.

Example:

DOUBLE PRECISION

FLOAT

This data type stores numbers of any precision and scale.

A FLOAT column can also specify a precision:

FLOAT (*precision*)

If between 1 to 21 inclusive, the format is single-precision floating point; if between 22 and 53, the format is double-precision floating point.

If the precision is omitted, double-precision is assumed.

Examples:

```
FLOAT  
FLOAT (20)  
FLOAT (53)
```

REAL

This data type specifies a column containing single-precision floating point numbers.

Example:

```
REAL
```

Date/Time Data Types

SQLBase has these data types for date and time data:

- DATETIME (or TIMESTAMP)
- DATE
- TIME

You can use date columns in comparison operations with other dates. You can also use dates in some functions and expressions.

Internally, SQLBase stores all date and time data in its own floating point format. The internal floating point value is available through an application program API call.

This format interprets a date or time as a number with the form:

DAY.TIME

DAY is a whole number that represents the number of days since December 30, 1899. December 30, 1899 is 0, December 31, 1899 is 1, and so forth.

TIME is the fractional part of the number. Zero represents 12:00 AM.

March 1, 1900 12:00:00 PM is represented by the floating point value 61.5 and March 1, 1900 12:00:00 AM is 61.0.

Anywhere a date/time string can be used in a SQL command, a corresponding floating point number can also be used.

SQLTalk and SQLBase provide extensive support for date/time values. See the section in this chapter called *Date/Time Values* for more.

DATETIME (or TIMESTAMP)

This data type is used for columns which contain data that represents both the date and time portions of the internal number.

DATETIME data can be input using any of the allowable date and time formats listed for the DATE and TIME data types.

When a part of an input date/time string is omitted, SQLBase supplies the default of 0, which converts to December 30, 1899 (date part) 12:00:00 AM (time part).

TIMESTAMP can be used instead of DATETIME for DB2 compatibility.

Examples:

```
DATETIME  
TIMESTAMP
```

The time portion of DATETIME has resolution to the second; the time portion of TIMESTAMP has resolution to the microsecond.

DATE

This data type stores a date value. The time portion of the internal number is 0. On output, only the date portion of the internal number is retrieved.

Example:

```
DATE
```

TIME

This data type stores a time value. The date portion of the internal number is 0. On output, only the time portion of the internal number is retrieved.

Example:

TIME

TIME has resolution to the second.

Data Type Conversions

Data Type Conversions in Assignments

SQLBase is flexible in the data types it accepts for assignment operations:

Source Data Type	Target Data Type	Comment
Character	Numeric	Source value must form a valid numeric value (only digits and standard numeric editing characters).
Numeric	Character	Single quotes are not needed.
Date/Time	Numeric	
Numeric	Date/Time	
Date/Time	Character	Single quotes are not needed.
Character	Date/Time	Source value must form a valid date/time value.

Data Type Conversions in Functions

Usually, functions accept any data type as an argument if the value conforms to the operation that function performs. SQLBase will automatically convert the value to the required data type.

For example, in functions that perform arithmetic operations, arguments can be character data types if the value forms a valid numeric value (only digits and standard numeric editing characters).

For date/time functions, an argument can be a character or numeric data type if the value forms a valid date/time value.

Constants

A constant (also called a literal) specifies a single value. Constants are classified as:

- String constants.
- Numeric constants.
- Date and time constants.

String Constants

A string is a sequence of characters. A string constant must be enclosed in *single* quotes (apostrophes) when used in a SQL command.

To include a single quote in a string constant, use two adjacent single quotes.

Numeric Constants

A numeric constant refers to a single numeric value.

A numeric constant is specified with digits. The value can include a leading plus or minus sign and a decimal point.

A numeric constant can be entered in E notation.

Date/Time Constants

Date and time values can be used as constants. See the section called *Date/Time Values* in this chapter for more.

Examples of Constants

Constant Type	Example	Explanation
Character String	'CHICAGO'	Character string must be enclosed in single quotes.
	'DON''T'	To include a quote character as part of a character string, use two consecutive single quotes.
	''	Two consecutive single quotes with no intervening character represents a null value.
	'1492'	If digits are enclosed in quotes, it is assumed to be a character string and not a number.
Numeric	2580	Digits not enclosed in quotes are assumed to be numeric values.
	1249.57	Numeric constant with decimal point.
	-1249	Leading plus or minus signs may be used on numerics.
	4.00E+7	E-notation can be used to express numeric values.
Date/time	10-27-85	Date/time constants do not need to be quoted.
	27-Oct-1985	

System Keywords

Certain keywords have values that can be used in some commands in place of column names or constants. These special keywords are:

NULL	The absence of a value. NULL can be used as a constant in a select list or in a search condition.
ROWID	The internal address of a row. ROWID can be used instead of a column name in a select list or in a search condition.
USER	The authorization-id of the current user. USER can be specified instead of a constant in a select list or in a search condition.

SQLBase also provides these date/time system keywords:

SYSDATETIME
SYSDATE
SYSTEMTIME
SYSTEMTIMEZONE

See the section called *Date/Time Values* in this chapter for more.

Examples of System Keywords

```
SQL> SELECT JOB FROM EMP WHERE DEPTNO IS NULL;
```

```
SQL> SELECT ROWID FROM EMP WHERE SAL > 5000;
```

```
SQL> CREATE VIEW MYTABLES AS  
SELECT * FROM SYSADM.SYSTABLES WHERE CREATOR = USER;
```

Expressions

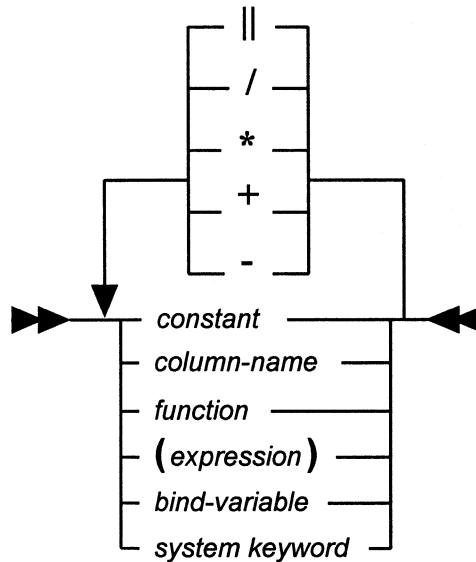
An expression is:

- An item that yields a single value.
- A combination of items and operators that yield a single value.

An *item* can be any of the following:

- A column name.
- A constant.
- A bind variable.
- The result of a function.
- A system keyword.
- Another expression.

The form of an expression is:



If you do not use arithmetic operators, the result of an expression is the specified value of the term. For example, the result of 1+1 is 2; the result of the expression AMT (where AMT is a column name) is the value of the column.

Null Values in Expressions

If any item in an expression contains a null value, then the result of evaluating the expression is null (*unknown* or *false*).

This operator (||) concatenates two or more strings:

```
string || string
```

The result is a single string.

For example, if the column PLACE contains the value "PALO ALTO", then:

```
' was born in ' || PLACE
```

Returns the string "was born in PALO ALTO".

This example prefixes the mens' names with "Mr. ":

```
SQL> SELECT 'Mr. ' || LASTNAME FROM VIPS;
```


Precedence

The following precedence rules are used in evaluating arithmetic expressions:

- Expressions in parentheses are evaluated first.
- The unary operators (+ and -) are applied before multiplication and division.
- Multiplication and division are applied before addition and subtraction.
- Operators at the same precedence level are applied from left to right.

Examples of Expressions

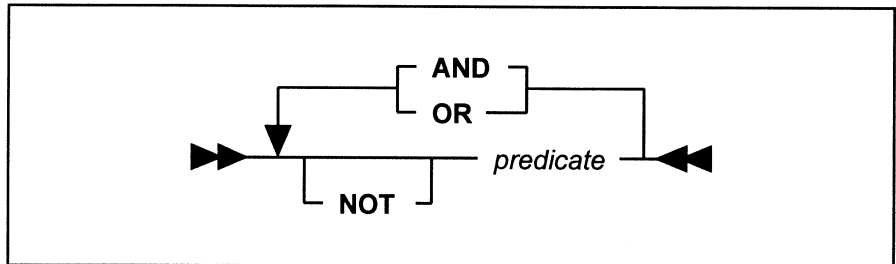
AMOUNT*TAX	Column arithmetic.
(CHECKS.AMOUNT * 10) - PAST_DUE	Nested arithmetic with columns.
HIREDATE + 90	Column and constant arithmetic.
SAL + MAX(BONUS)	Function with column arithmetic.
SAL + :1	Bind variable with column arithmetic.
SYSDATETIME+4	Date/time system keyword arithmetic.

Search Conditions

A search condition in a WHERE clause qualifies the scope of a query by specifying the particular conditions that must be met. The WHERE clause can be used in these SQL commands:

- SELECT.
- DELETE.
- UPDATE.

A search condition contains one or more *predicates* connected by the logical (Boolean) operators OR, AND, and NOT.



The types of predicates that can be used in a search condition are discussed on the next page.

The specified logical operators are applied to each predicate and the results combined according to the following rules:

- Boolean expressions within parentheses are evaluated first.
- When the order of evaluation is not specified by parentheses, then NOT is applied before AND.
- AND is applied before OR.
- Operators at the same precedence level are applied from left to right.

A search condition specifies a condition that is *true*, *false*, or *unknown* about a given row or group. NOT (true) means false, NOT (false) means true and NOT (unknown) is unknown (false). AND and OR are shown in the truth table below.

Assume P and Q are predicates. The first two columns show the conditions of the individual predicates P and Q. The next two columns show the condition when P and Q are evaluated together with the AND operator and the OR operator. If an item in an expression in a search condition is null, then the search condition is evaluated as unknown (false).

P	Q	P AND Q	P OR Q
True	True	True	True
True	False	False	True
True	Unknown	Unknown	True
False	True	False	True
False	False	False	False
False	Unknown	False	Unknown
Unknown	True	Unknown	True
Unknown	False	False	Unknown
Unknown	Unknown	Unknown	Unknown

Nulls and Search Conditions

If a search condition specifies a column that might contain a null value for one or more rows, be aware that such a row will *not* be retrieved, because a null value is neither less than, equal to, nor greater than the value specified in the condition. The value of a null is *unknown* (false).

To select values from rows that contain null values, use the NULL predicate (explained later in this chapter):

WHERE *column-name* **IS NULL**

Examples of Search Conditions

This returns rows for employees who are in department 20.

```
SQL> SELECT * FROM EMP  
WHERE DEPTNO = 20;
```

This returns rows for employees who are in department 20 and are assigned to programming projects, or returns rows for employees who are programmers.

```
SQL> SELECT * FROM EMP  
WHERE (DEPTNO = 20 AND PROJ = 'Programming')  
OR JOB = 'Programmer';
```

The following WHERE clauses are equivalent.

```
SQL> SELECT * FROM EMP  
WHERE NOT (JOB = 'Programmer' OR PROJ = 'Programming');
```

```
SQL> SELECT * FROM EMP  
WHERE JOB != 'Programmer' AND PROJ != 'Programming';
```

Predicates

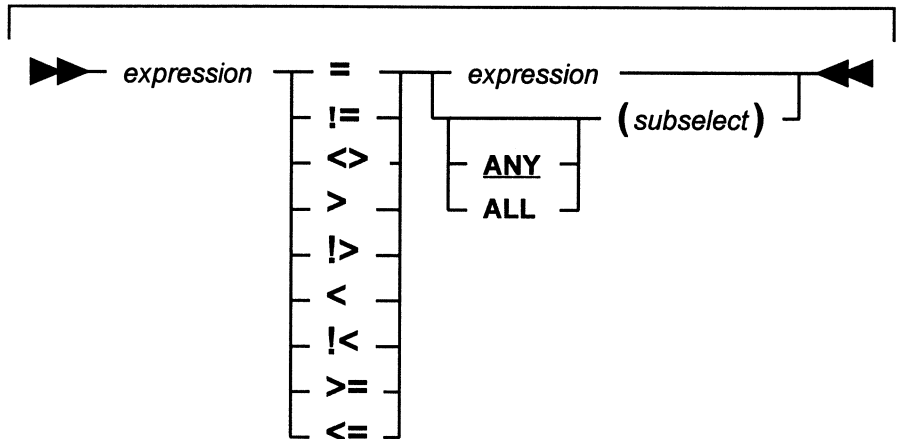
A predicate in a WHERE or HAVING clause specifies a search condition that is true, false, or unknown with respect to a given row or group of rows in a table.

Predicates use operators, expressions, and constants to specify the condition to be evaluated.

These types of predicates are described in this section:

- Relational.
- BETWEEN.
- NULL.
- EXISTS.
- LIKE.
- IN.

Relational Predicate



There are two types of relational predicates:

- *Comparison* relational predicate.
- *Quantified* relational predicate.

Comparison Relational Predicate

A comparison relational predicate compares a value to another value based on standard relational operators. The basic form of a comparison predicate is two expressions connected by a relational operator:

A > B

col1 != col2

The following are examples of comparison predicates:

```
SQL> SELECT * FROM EMP
WHERE SOC_SEC_NO = '506-42-7849';
```

```
SQL> SELECT * FROM EMP
WHERE HIREDATE <= '1-Jan-1984';
```

Quantified Relational Predicate

A quantified relational predicate compares the first expression value to a *collection* of values which result from a subselect command.

A SELECT command that is used in a predicate is called a subselect or subquery. A subselect is a SELECT command that appears in a WHERE clause of a SQL command.

The NOT operator can be used in place of the symbol (!).

You cannot use an ORDER BY clause in a subselect. Also, you cannot use a LONG VARCHAR column in a subselect.

Examples of SubSELECTs and Subqueries

SALARY is not equal to the average salary:

```
SQL> SELECT * FROM EMP  
WHERE SAL != (SELECT AVG(SAL) FROM EMP);
```

```
SQL> SELECT * FROM EMP  
WHERE SAL <> (SELECT AVG(SAL) FROM EMP);
```

SALARY is greater than the average salary:

```
SQL> SELECT * FROM EMP  
WHERE SAL > (SELECT AVG(SAL) FROM EMP);
```

SALARY is less than the average salary:

```
SQL> SELECT * FROM EMP  
WHERE SAL < (SELECT AVG(SAL) FROM EMP);
```

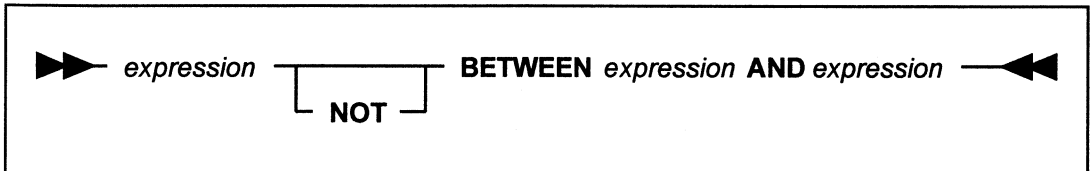
```
SQL> SELECT * FROM EMP  
WHERE SAL !> (SELECT AVG(SAL) FROM EMP);
```

SALARY is greater than or equal to any salary:

```
SQL> SELECT * FROM EMP  
WHERE SAL >= (SELECT SAL FROM EMP);
```

BETWEEN Predicate

The BETWEEN predicate compares a value with a range of values.

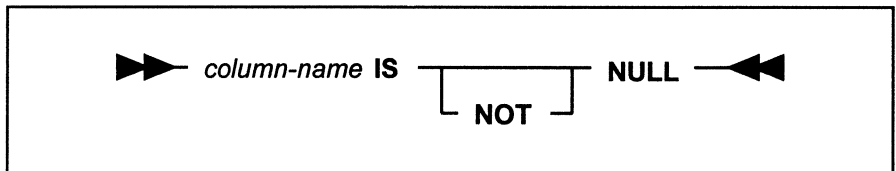


Example

```
SQL> SELECT * FROM EMP  
WHERE SAL BETWEEN 3000 AND 6000;
```

NULL Predicate

The NULL predicate tests for null values.

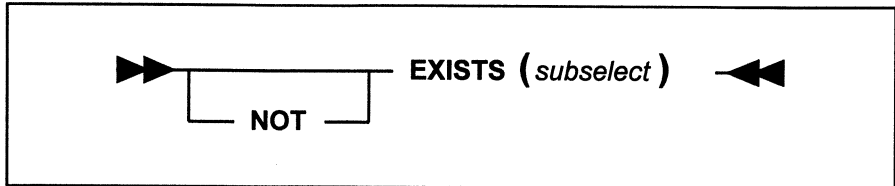


Example

```
SQL> SELECT * FROM EMP  
WHERE PROJNO IS NULL;
```


EXISTS Predicate

The EXISTS predicate tests for the existence of certain rows in a table.



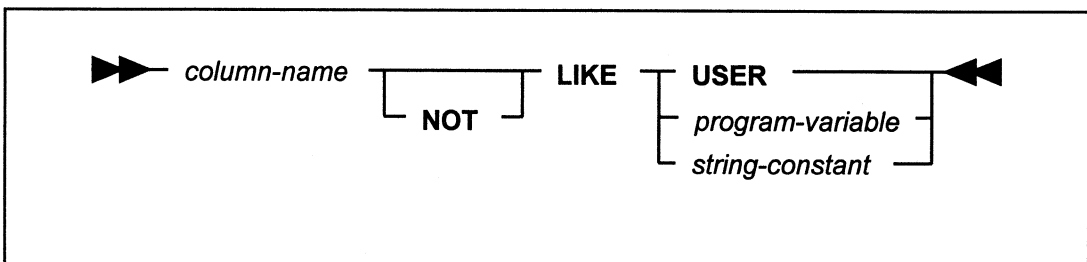
Example

This example retrieves the rows from the EMP table for the department number stored in bind variable :1.

```
SQL> SELECT * FROM EMP
WHERE EXISTS (SELECT * FROM DEPT WHERE DEPTNO = :1);
```

LIKE Predicate

The LIKE predicate searches for strings that match a specified pattern. The LIKE predicate can only be used with CHAR or VARCHAR data types.



The underscore (`_`) and the percent (`%`) are the pattern-matching characters:

<code>_</code>	Matches <i>any single character</i> .
<code>%</code>	Matches <i>one or more characters</i> .

The backslash (`\`) is the escape character for percent (`%`), underscore (`_`), and itself.

Examples

True for any name with the string 'son' anywhere in it.

```
SQL> SELECT * FROM EMP
WHERE NAME LIKE '%son%';
```

True for any 2-character code beginning with 'M'.

```
SQL> SELECT * FROM EMP
WHERE CODE LIKE 'M_';
```

Returns all rows where the value in the SPECIAL column is 'A24%'.

```
SQL> SELECT * FROM EMP
WHERE SPECIAL LIKE 'A24\%';
```

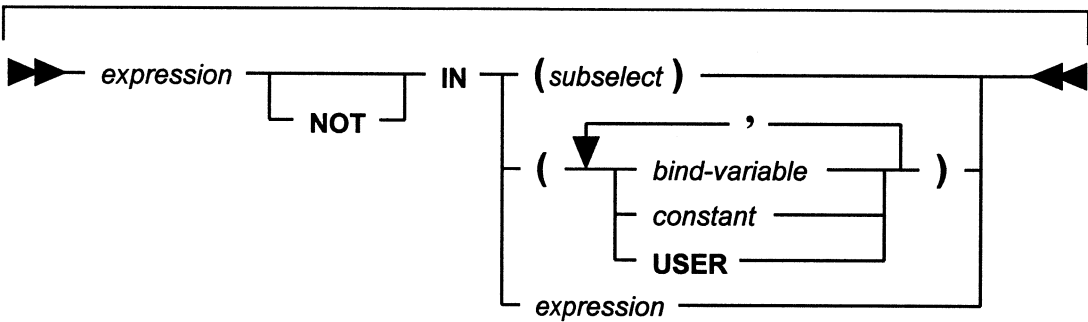
Returns all rows where the value in the SPECIAL column begins with 'A24%'

```
SQL> SELECT * FROM EMP
WHERE SPECIAL LIKE 'A24\%%';
```

IN Predicate

The IN predicate compares a value to a collection of values. The collection of values can be listed in the command or can be the result of a subselect.

If there is only one item in the list of values, parentheses are not required.



Examples

```
SQL> SELECT * FROM EMP
WHERE DEPTNO IN (10,20,30);
```

```
SQL> SELECT * FROM EMP
WHERE ORDERDATE NOT IN
(SELECT ORDERDATE FROM ORDERS WHERE ORDERDATE < 12-MAY-86);
```

```
SQL> SELECT * FROM EMP
WHERE NAME NOT IN (:1, :2, 'JONES');
```

```
SQL> SELECT * FROM EMP
WHERE @LEFT (NAME, 1) IN ('J', 'M', 'D');
```

Functions

A function returns a value that is derived by applying the function to its arguments.

SQLBase has many functions that manipulate strings, dates and numbers. Functions are classified as:

- Aggregate functions.
- String functions.
- Date and time functions.
- Logical functions.
- Special functions.
- Math functions.
- Finance functions.

The functions are described in chapter 8 of this manual.

Date/Time Values

Entering Date/Time Values

Although SQLBase stores dates and times in its own internal format, it accepts all conventional date and time input formats, including ISO, European, and Japanese Industrial Time.

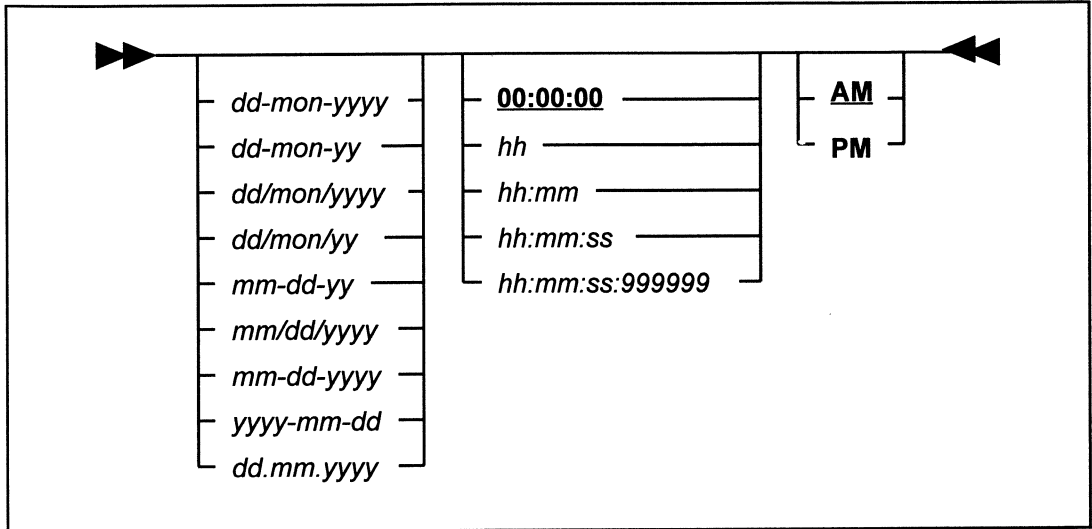
Input for a date or time column is a string that contains date or time information. The input string has a date portion and/or a time portion, depending on whether the date/time is a DATE, a TIME or a DATETIME.

A forward slash (/), hyphen (-) or period (.) are used as the delimiter for the parts of a date, as shown in the diagram on the next page. You must be consistent within a single command. A colon (:) or a period (.) are both accepted as the delimiter for times. Case is ignored by SQLBase when entering months. Either a space or a hyphen can separate the date portion from the time portion.

Letter combinations used in the formats below have the following meanings:

<i>yy, yyyy</i>	Year
<i>mm, mon</i>	Month
<i>dd</i>	Day
<i>mi</i>	Minutes
<i>hh</i>	Hours
<i>ss</i>	Seconds
<i>999999</i>	Microseconds

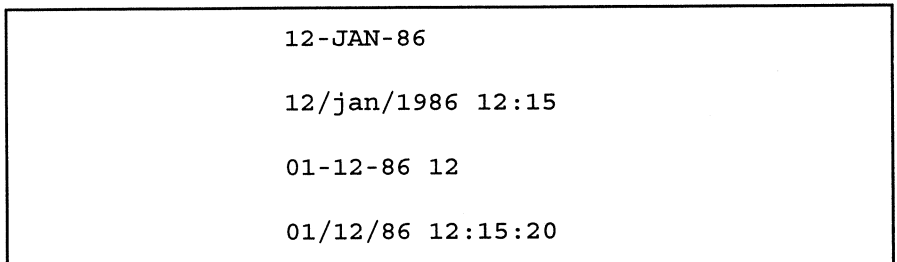
Valid input formats for date/time values are:



A time string can contain an AM or PM designation. The default is AM. SQLBase recognizes military time (24 hour clock) on input if the AM/PM parameter is omitted.

Examples

Some examples of date/time input strings are:



Date/Time System Keywords

Certain system keywords return a date/time values. These system keywords can be used in expressions to specify an interval of a specified type.

The keyword values for SYSDATETIME, SYSDATE, SYSTIME, and SYSTIMEZONE are set at the beginning of execution of a command.

System Keyword	Meaning
SYSDATETIME CURRENT TIMESTAMP * CURRENT DATETIME *	Current date and time.
SYSDATE CURRENT DATE *	Current date.
SYSTIME CURRENT TIME *	Current time.
SYSTIMEZONE CURRENT TIMEZONE *	Timezone interval.
MICROSECOND[S]	Time in microseconds.
SECOND[S]	Time in seconds.
MINUTE[S]	Time in minutes.
HOUR[S]	Time in hours.
DAY[S]	Time in days.
MONTH[S]	Time in months.
YEAR[S]	Time in years.
* DB2 Compatible	

Resolution for Time Keywords

The table below show the resolution in seconds for the time keywords.

Time Keyword	Resolution
SYSDATETIME CURRENT TIME CURRENT DATETIME	Seconds (hh:mm:ss)
SYSTIME CURRENT TIMESTAMP	Microseconds (hh:mm:ss:999999)
SECOND[S]	Seconds (ss)
MICROSECONDS	Microseconds (ss:999999)

Example of Date/Time System Keyword

```
INSERT INTO CALLS (DATE) VALUES (SYSDATETIME)
```

Time Zones

The keyword `SYSTIMEZONE` returns the time zone for the system as a time interval in hours. The time interval is the difference between local time and Greenwich Mean Time:

$$\text{TIMEZONE interval} = \text{LOCAL TIME} - \text{GMT}$$

This interval is set with the *timezone* keyword in *sql.ini*. The default value is 0 (Greenwich Mean Time).

For instance, GMT is 5 hours later than EST (Eastern Standard Time). If the time was 5:00 A.M. EST, then

```
TIMEZONE interval = 5 - 10 = -5
TIMEZONE= -5
```

To get the current time in GMT, use the expression:

```
(SYSTIME - SYSTIMEZONE)
```

Date/Time Expressions

Addition or subtraction operators can be applied to dates. The results are as follows:

- Date + Number (of days) is DATETIME.
- Date - Number (of days) is DATETIME.
- Date - Date is a number (of days).

Note that if you add or subtract a non-date/time value to or from DATE, the result is a DATETIME. To make the result a DATE, use an expression like this:

- Date + Number DAYS

where *Number* is a numeric value.

The system keywords that represent time intervals (such as MONTH or MICROSECOND) can be added to or subtracted from other date and time quantities to get new date and time quantities.

For example, the following expression yields a new DATETIME value.

```
SYSDATETIME + 3 MINUTES
```

If you do not specify the type of interval, the number is assumed to be DAYS. The following example adds one day to the current date.

```
SYSDATE + 1
```

You could also use the expression:

`SYSDATE + 1 DAY`

Only a constant can precede a date/time keyword.

Microseconds, seconds, minutes, hours, and days behave like numbers. MONTH and YEAR intervals however, are special cases since they do not have a fixed value in terms of the number of days in the month or year. February has 28 or 29 days, March has 30; a year can be 365 or 366 days.

Use the following rules for MONTH and YEAR intervals:

- MONTH and YEAR intervals can only be added to or subtracted from a DATE or a DATETIME quantity.

Valid: `(SYSDATE + 3 DAYS) + 1 YEAR`

Invalid: `SYSDATE + (3 DAYS + 1 YEAR)`

- When MONTHs are added, the month number (and if necessary the year number) is incremented. If the day represents a day beyond the last valid day for the month and year, it is adjusted to be a valid date.

Examples of Date/Time Expressions

Date/Time Expression	Result
<code>31-Jan-1988 + 1 MONTH</code>	<code>29-Feb-1988</code>
<code>20-Jan-1988 + 1 MONTH</code>	<code>20-Feb-1988</code>
<code>29-Feb-1988 + 1 YEAR</code>	<code>28-Feb-1989</code>
<code>31-Jan-1988 + 1 MONTH - 1 MONTH</code>	<code>29-Jan-1988</code>

Joins

This section discusses these types of joins:

- Equijoins.
- Non-equijoins.
- Outer joins.
- Self joins.

A join takes place when the result set comes from more than one table.

The command:

```
SQL> SELECT * FROM CUSTOMER WHERE NAME = 'A1';
```

yields a single row from the CUSTOMER table (assuming there is only a single customer, A1).

You cannot perform an operation with the CURRENT OF clause on a result set that you formed with a join.

Equijoin

The following query matches customer names and order numbers. Two tables are used: CUSTOMER and ORDERS.

```
SQL> SELECT CUSTOMER.NAME, ORDERNO  
FROM CUSTOMER, ORDERS  
WHERE CUSTOMER.CUSTNO = ORDERS.CUSTNO;
```

Each result row contains the customer name and an order number. If customer number 1 made three orders, three rows would result. The single customer row containing the customer's name and number would be "joined" to each of the three order rows.

The ORDERS rows are related to the CUSTOMER using the key column, CUSTNO, which appears in both the CUSTOMER table and the ORDERS table.

This type of search condition, which specifies a relationship between two tables based on an *equality*, is called an equijoin.

Cartesian Product

The specification of the join condition as a relational predicate in the search condition is necessary to avoid a Cartesian product: the set of all possible rows resulting from a join of more than one table. For example, suppose we specified the above query as follows:

```
SQL> SELECT CUSTOMER.NAME, ORDERNO FROM CUSTOMERS, ORDERS;
```

The result would be the product of the number of rows in the customer table and the number of rows in the orders table. If CUSTOMER had 100 rows, and ORDERS had 500 rows, the Cartesian product would be every possible combination, or 50,000 rows — probably not the desired result.

The correct way to get each customer and order listed (a set of 500 rows) is with an equijoin, as follows:

```
SQL> SELECT CUSTOMER.NAME, ORDERNO  
FROM CUSTOMER, ORDERS  
WHERE CUSTOMER.CUSTNO = ORDERS.CUSTNO;
```

Outer Join

In the example of the equijoin above, the search condition specified a join on customers and orders. What happens if customer NEWACCOUNT has not yet made an order? The above query does not retrieve that customer.

An outer-join forces a row from one of the participating tables to appear in the result if there is no matching row. A plus sign (+) is added to the join column of the table which might not have rows to satisfy the join condition.

Only *one-way outer joins* are supported; you cannot use the (+) on both names. Only one outer join is allowed per SELECT.

List customer names and their order numbers. Include customers who have made no orders.

```
SQL> SELECT CUSTOMER.CUSTNO, NAME
FROM CUSTOMER, ORDERS
WHERE CUSTOMER.CUSTNO = ORDERS.CUSTNO(+);
```

The plus sign (+) after ORDERS.CUSTNO causes an extra row containing all null values to be temporarily added to the ORDERS table. This NULL row of the ORDERS table is joined to rows in the CUSTOMER table which do not have matching orders. Therefore, all customer numbers are retrieved.

To retrieve all customers who have *not* placed orders, the SQL command joins the tables and then searches for the NULL row matches only.

```
SQL> SELECT CUSTNAME, CUSTOMER.NAME
FROM CUSTOMER, ORDERS
WHERE CUSTOMER.CUSTNO = ORDERS.CUSTNO (+)
AND ORDERS.CUSTNO IS NULL;
```

Self Join

A self join lets you join a table to itself, as though it was two separate tables. To do this, the self-join table is given a correlation name. The example below finds all the suspects who were arrested on the same day as 'JAMES'.

```
SQL> SELECT SUS2.NAME, SUS2.ARREST_DATE
FROM SUSPECTS SUS1, SUSPECTS SUS2
WHERE SUS1.ARREST_DATE=SUS2.ARREST_DATE
AND SUS1.NAME = 'JAMES';
```

The SUSPECTS table is treated as two tables, using the correlation names SUS1 and SUS2. The arrest date of JAMES is retrieved from correlation table SUS1. Then this arrest date is used as a search condition for table SUS2.

This same information can be retrieved using a subquery. See the section called *Subqueries* in this chapter.

Non-Equijoin

A non-equijoin joins tables to one another based on comparisons other than equality. Any of the relational operators can be used, (such as >, <, !=, BETWEEN, or LIKE).

This example specifies a join using the BETWEEN operator. The table SENTENCES contains the minimum and maximum sentences for each crime. Each crime has an identifying number.

The query makes sure that each prisoner's sentence falls between the minimum and maximum allowed for that crime by finding any rows outside of that range.

```
SQL> SELECT NAME, SUSPECTS.CRIME, SENTENCE
FROM SUSPECTS, SENTENCES
WHERE SUSPECTS.CRIME = SENTENCES.CRIME
AND SENTENCE NOT BETWEEN MAXSENT AND MINSENT;
```

Subqueries

A subquery is a search condition that is a nested SELECT command (also called a subselect). The subquery specifies a result table from one or more tables or views, in the same manner as any other SELECT. Each result row of the subselect is used as a basis for qualifying a candidate result row in the outer select.

You cannot use an ORDER BY clause in a subquery. Also, you cannot use a LONG VARCHAR in a subquery.

Examples of Subqueries

For example, find all the suspects who were arrested on the same day as James.

```
SQL> SELECT NAME, ARREST_DATE FROM SUSPECTS
WHERE ARREST_DATE =
(SELECT ARREST_DATE FROM SUSPECTS
WHERE NAME = 'JAMES');
```

First, the arrest date of James is retrieved and this value is used to complete the search condition of the main or outer query.

In the previous example, the subquery was executed once to retrieve a single value used by the main query. In the following SELECT command, called a correlated subquery, the subquery is executed once for each candidate row in the main query.

For example, find the prisoners whose sentence is longer than the average sentence of other prisoners who committed the same crime.

```
SQL> SELECT PRISONER, CRIME, SENTENCE
FROM PRISONERS P
WHERE SENTENCE >
(SELECT AVG(SENTENCE) FROM PRISONERS
WHERE P.CRIME = CRIME);
```

Bind Variables

A bind variable refers to a data value associated with a SQL command. Bind variables associate (bind) a syntactic location in a SQL command with a data value that is to be used in that command.

Bind variables can be used wherever a data value is allowed:

- WHERE clause.
- VALUES clause in an INSERT command.
- SET clause of in UPDATE command.

Bind variable names start with a colon (:). The name can be:

- The name of a variable that is declared in a program (such as :ARG1).
- A number that refers to the relative position among the data items associated with the SQL command (such as :1, :2, :3).

See the manual for the client application that you are using (such as the C/API, SQLTalk, or SQLWindows) for the details on how to use bind variables.

Chapter 7

SQL Command Reference

About This Chapter

This chapter contains the syntax, description, and examples of each SQL command. This chapter is organized alphabetically by command name.

SQL Command Summary

Command	Function
ALTER DATABASE	Changes storage group or log for database.
ALTER DBAREA	Changes the size of a database area.
ALTER PASSWORD	Changes a password.
ALTER STOGROUP	Adds or drops a database area from a storage group.
ALTER TABLE	Changes the description of a table.
CHECK DATABASE	Checks database for integrity.
COMMENT ON	Replaces or adds a comment to the description of a table, view, or column in the system catalog.
COMMIT	Ends a logical unit of work and commits database changes made by it.
CREATE DATABASE	Physically creates a database.
CREATE DBAREA	Creates a database area.
CREATE INDEX	Creates an index on a table.
CREATE STOGROUP	Creates a storage group.
CREATE SYNONYM	Defines an alternate name for a table or view.
CREATE TABLE	Defines a table.

Command	Function
CREATE VIEW	Defines a view of one or more tables or views.
DEINSTALL DATABASE	Takes a database off the network, making it unavailable to users.
DELETE	Deletes one or more rows from a table.
DROP	Removes an object from the system catalog.
DROP DATABASE	Physically deletes a database.
DROP DBAREA	Physically deletes a database area.
DROP STOGROUP	Deletes a storage group.
GRANT	Grants database authority or privileges.
INSERT	Inserts one or more rows into an existing table.
INSTALL DATABASE	Puts a database on the network, making it accessible to users.
LABEL	Adds or changes labels in catalog descriptions
REVOKE	Revokes database authority or privileges.
ROLLBACK	Terminates a logical unit of work and backs out database changes made during the last transaction.
SAVEPOINT	Assigns a checkpoint within a transaction.
SELECT	Queries tables or views.

Command	Function
SET DEFAULT STOGROUP	Specifies the default storage group.
UNION	Merges results from 2 or more SELECTs.
UPDATE	Updates the values of columns in a table or view.
UPDATE STATISTICS	Updates the statistics for an index in a table.

ALTER DATABASE

```

▶▶ ALTER DATABASE database-name ▶

▶ [ STOGROUP
  LOG ] stogroup-name ◀◀
  
```

Description

This command changes the storage group for a database or its log files. ALTER DATABASE only affects future allocations of space. Existing databases or log files are not moved or affected.

database-name

The name of the database to be changed.

STOGROUP

Changes the storage group for a database.

LOG

Changes the storage group for the database's log files.

stogroup-name

The name of the storage group to be changed. A storage group is a list of database areas, which are similar to files or a partition.

Example

```
SQL> ALTER DATABASE ACCT STOGROUP ACCTLIST;
```

See Also

```
CREATE DATABASE  
CREATE STOGROUP  
DELETE STOGROUP
```

ALTER DBAREA

```
ALTER DBAREA dbarea-name
```

```
SIZE megabytes
```

Description

This command changes the size of a database area. When increasing the size of a database area, available space is checked at the time of the operation. If the size is decreased, the available space must not be in current use.

dbarea-name

The name of the database area to be changed.

SIZE *megabytes*

The size of the database area in megabytes.

Example

```
SQL> ALTER DBAREA ACCTAREA SIZE 10;
```

See Also

```
ALTER DATABASE  
CREATE DBAREA  
DROP DBAREA
```

ALTER PASSWORD



```
ALTER PASSWORD — old-password →  
TO — new-password ←←
```

Description

This command changes your password.

The password is stored in the system catalog and can be read by a user with SYSADM or DBA privileges. However, the password is encrypted and a user must use the @DECRYPT function to read it. Note that passwords are encrypted when transmitted across a network.

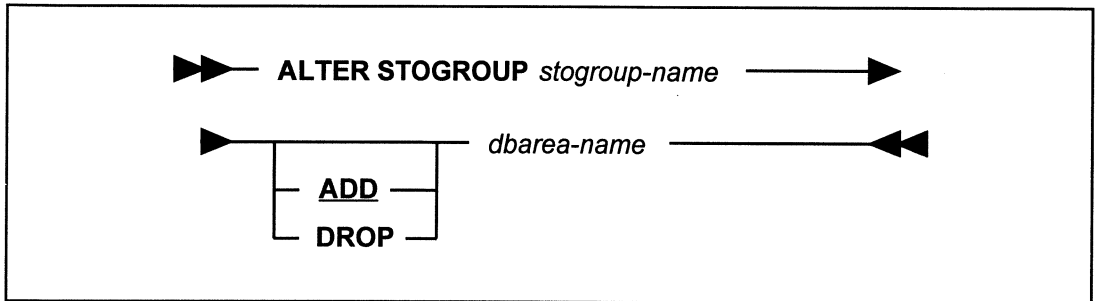
Examples

```
SQL> ALTER PASSWORD OLDSTUFF TO NEWSTUFF;
```

See Also

GRANT
REVOKE

ALTER STOGROUP



Description

This command changes the storage group for a database area. The command only affects future allocations of space. Existing databases or log files are not moved or affected.

stogroup-name

A storage group is a list of database areas.

ADD *dbarea-name*

Adds a database area to the storage group.

DROP *dbarea-name*

Drops a database area from the storage group.

Example

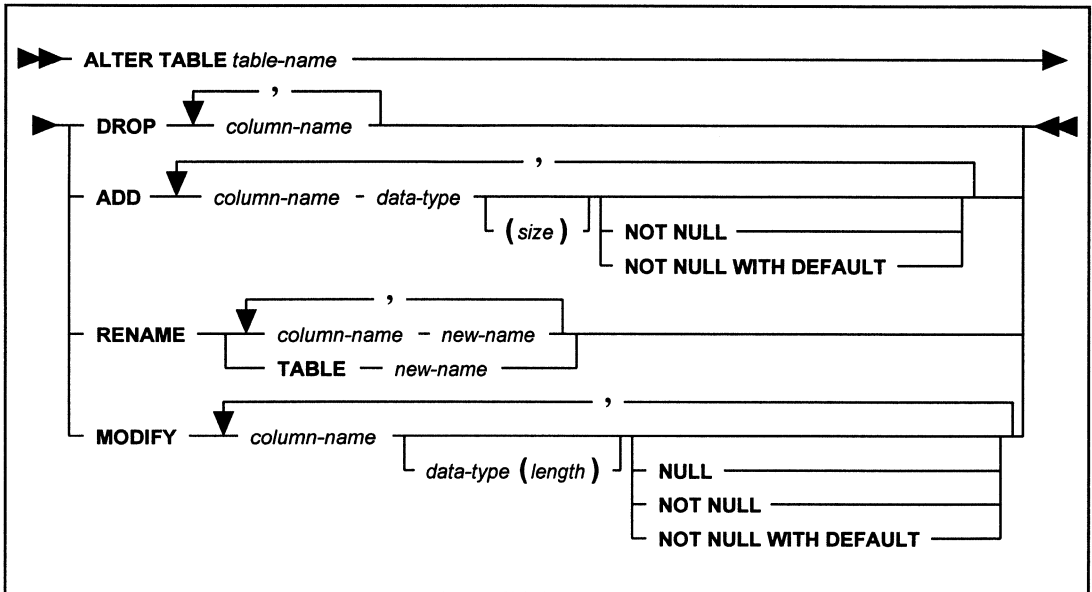
```
SQL> ALTER STOGROUP ACCTFILES ADD ACCTAREA;
```

```
SQL> ALTER STOGROUP ACCTFILES DROP ACCTAREA;
```

See Also

```
CREATE STOGROUP  
DELETE STOGROUP
```

ALTER TABLE



Description

This command:

- Adds, drops, or modifies a column.
- Renames a column or table.

Views that reference dropped or renamed columns or tables are automatically dropped. Views that reference modified columns are also dropped.

Precompiled commands that reference dropped or renamed columns or tables are not dropped. Such precompiled commands could become invalid and return an error if executed.

You must have the ALTER privilege on the table to execute this command.

ADD

This adds a column to a table. Columns are defined the same way as in the CREATE TABLE command. See *Data Types* in chapter 6 of this manual.

Adding a column does not effect existing views or precompiled commands.

You can add columns to user tables or to system catalog tables. However, if you add columns to system catalog tables, they are not maintained by SQLBase. For example, UNLOAD will ignore any user-defined columns in a system catalog table.

DROP

This removes a column from a table.

If the column has data, it is lost.

You cannot drop:

- An indexed column.
- A column belonging to a primary or foreign key.
- System defined columns in the system catalog.

MODIFY

This changes the attributes for a column.

You can increase the length of a character column, but you cannot decrease the length. You specify the data type when you increase the length of a character column.

You *cannot* change the data type of a column.

You *cannot* change the the length of a numeric column.

NULL

This removes a NOT NULL attribute for a column.

NOT NULL

This adds a NOT NULL attribute to a column that currently accepts nulls.

If the column contains NULL values, you cannot redefine the column as NOT NULL.

You cannot modify system-defined columns in the system catalog.

NOT NULL WITH DEFAULT

This clause prevents a column from containing null values and allows a default value other than the null value. The default value used depends on the data type of the column, as follows:

Data Type	Default Value
Numeric	0 (zero)
Date/Time	Current date/time
Character	Blank

The NOT NULL WITH DEFAULT clause causes an INSERT to use the above defaults. SQLBase puts a 'D' in the NULLS columns of the SYSCOLUMNS table and treats it like a NOT NULL field.

The ALTER TABLE command *does not* allow the addition of a column defined as NOT NULL if rows of data already exist. The ALTER TABLE command *does* allow a new column defined as NOT NULL WITH DEFAULT to be added if no rows of data exist for the specified table.

To add columns defined as NOT NULL or NOT NULL WITH DEFAULT when rows of data already exist:

1. Add the column with ALTER TABLE, but do *not* specify a NOT NULL or NOT NULL WITH DEFAULT clause.
2. Update the values in the new column to some value other than NULL.
3. Change the column to NOT NULL or NOT NULL WITH DEFAULT with the ALTER TABLE command.

The NOT NULL WITH DEFAULT clause is compatible with DB2.

RENAME

This renames a table or column.

System catalog tables and system-defined columns in the system catalog cannot be renamed.

Examples

Add a new column called JOB that contains a maximum of 20 characters to the EMP table:

```
SQL> ALTER TABLE EMP ADD JOB VARCHAR(20);
```

Increase the size of the column JOB to 40 characters and make it a NOT NULL column:

```
SQL> ALTER TABLE EMP MODIFY JOB VARCHAR(40) NOT NULL;
```

Drop the columns JOB and SALARY.

```
SQL> ALTER TABLE EMP DROP JOB, SALARY;
```

Change the name of EMP to EMPLOYEE.

```
SQL> ALTER TABLE EMP RENAME TABLE EMPLOYEE;
```

Add the NOT NULL attribute to the START_DATE column:

```
SQL> ALTER TABLE EMPLOYEE MODIFY START_DATE NOT NULL;
```

Now drop the NOT NULL attribute:

```
SQL> ALTER TABLE EMPLOYEE MODIFY START_DATE NULL;
```

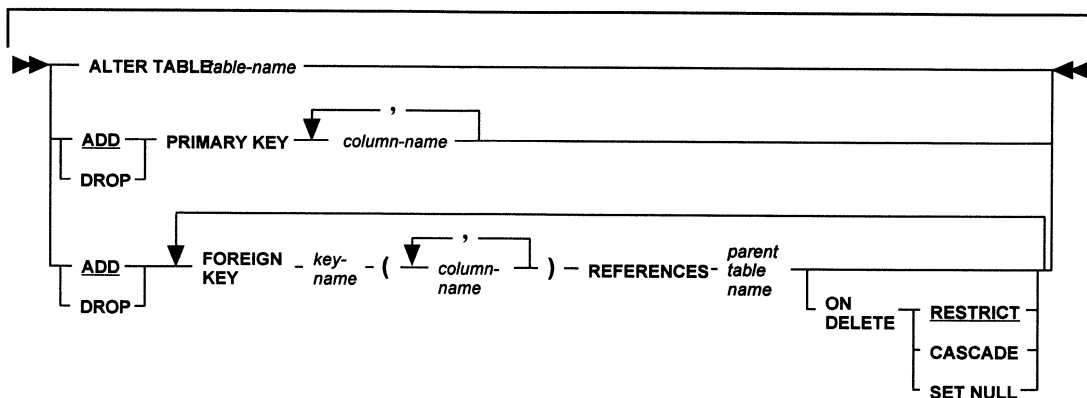
Drop the primary key:

```
SQL> ALTER TABLE EMPLOYEE  
DROP PRIMARY KEY;
```

See Also

CREATE TABLE
DROP

ALTER TABLE (Referential Integrity)



Description

When you use ALTER TABLE with referential constraints, you can add or drop primary and foreign keys. You do not need to specify ADD since it is the default, but you must specify DROP if this is your intention.

Before dropping the primary key, consider the effect this will have on the application programs. The programs must then enforce referential constraints without the primary key.

When you use ALTER TABLE to apply the FOREIGN KEY referential constraint, the values in the foreign and primary keys must conform with referential integrity. Otherwise, the command will be rejected.

PRIMARY KEY

In a database with referential integrity, this adds or drops the primary key of the table. If you drop the primary key, the table continues to exist with a unique index on the same list of columns (if the table has a unique index). The relationship between the tables is dropped if the table has a dependent.

To drop the primary key, you must have the ALTER privilege on both the parent and dependent tables.

The following rules apply to primary keys:

- The values of the primary key must be unique — no two rows of a table can have the same key values.
- A table can have only one primary key.
- The primary key can be made up of one or more columns in a table.
- Each column in the primary key should be classified with the NOT NULL constraint.

FOREIGN KEY

This adds or drops a foreign key to an existing table.

To drop a foreign key, you must have the ALTER privilege on both the parent and dependent tables.

Before you drop a foreign key, consider carefully the effect this will have on application programs. Dropping a foreign key drops the corresponding referential relationship and delete rule. Without the foreign key, programs must enforce these constraints.

The following rules apply to foreign keys:

- A table can have many foreign keys.
- A foreign key can be made up of one or more columns of a table.
- The foreign key and primary key must contain the same number of columns, and the foreign key's data types must match those of the primary key on a one-to-one basis.
- A foreign key column may or may not be NULL.
- A foreign key value is NULL if any part is NULL.

REFERENCES

This identifies the parent table in a relationship and defines the necessary constraints. The REFERENCES clause must accompany the FOREIGN KEY clause.

ON DELETE

This specifies the DELETE rules for the table.

The DELETE rules are optional.

The default is RESTRICT.

DELETE rules are only used to define a foreign key.

CASCADE

This deletes the selected rows first, and then deletes the dependent rows, honoring the delete rules of their dependents.

RESTRICT

This specifies that a row can be deleted if no other row depends on it. If a dependent row exists in the relationship, the delete will fail.

SET NULL

This specifies that for any delete performed on the primary key, matching values in the foreign key are set to null.

Examples

Add a foreign key to the EMP_SAL table:

```
SQL> ALTER TABLE EMP_SAL  
FOREIGN KEY (EMPNO) REFERENCES EMPLOYEE  
ON DELETE RESTRICT;
```

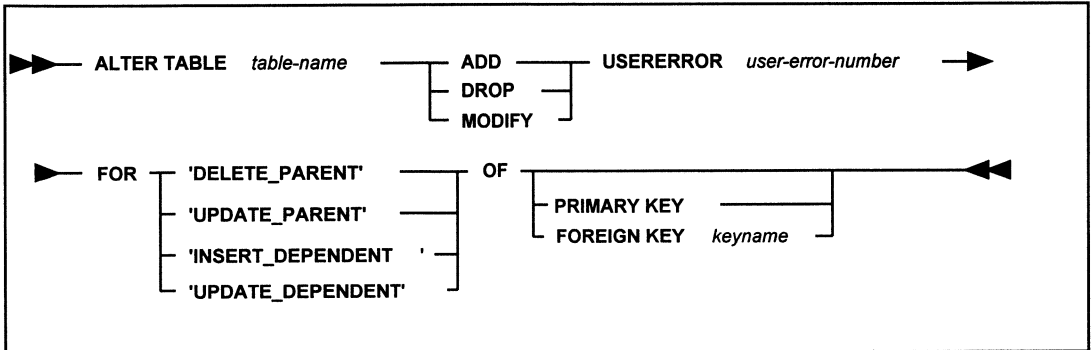
Add a primary key to the EMPLOYEE table:

```
SQL> ALTER TABLE EMPLOYEE  
PRIMARY KEY (EMPNO);
```

See Also

CREATE TABLE

ALTER TABLE (Error Messages)



Description

To make error messages specific to a particular violation of referential integrity, you can edit the `error.sql` file and use `ALTER TABLE` statements.

ADD

This adds a specific error message for referential integrity. You must enter the `user_error_number` from the `error.sql` file.

DROP

This deletes a specific error message. You do not need to enter the `user_error_number` for a `DROP` command.

MODIFY

This modifies the error message number for referential integrity.

USERERROR *user-error-number*

This specifies the error number in the `error.sql` file. You can modify the `error.sql` file to add an appropriate error message.

'INSERT_DEPENDENT'

This specifies that an insertion failed because there was no parent row in the parent table.

'UPDATE_DEPENDENT'

This specifies that an update failed because there was no parent row in the parent table for the new set of values.

'DELETE_PARENT'

This specifies that a deletion failed because there were dependent rows in the dependent table.

'UPDATE_PARENT'

This specifies that an update failed because there were dependent rows in the dependent table (dependent on the values to be updated).

Example

A user may attempt to delete the employee number of an employee who still works for the company. You can avoid this problem by editing the *error.sql* file:

```
20000 xxx xxx Employee number cannot be deleted while  
employee still works for this company.
```

Then you can use the ALTER TABLE statement to add the new error message:

```
SQL> ALTER TABLE empno ADD USERERROR 20000 FOR  
'DELETE_PARENT" OF PRIMARY KEY;
```

Then, if a user attempts to delete the employee number, the new error message will appear:

```
SQL> DELETE FROM empno where empno = 77;  
Error: Employee number cannot be deleted while employee  
still works for this company.
```

See Also

CREATE TABLE

CHECK DATABASE

▶▶ CHECK DATABASE ◀◀

Description

This command performs integrity checks on the entire database:

- Checks the integrity of the database and the table free space data structures.
- Checks each table row count against the actual number of rows.
- Cross-checks each index against the base table.
- Checks each row and index page for integrity.
- Ensures that each page in the database is a part of an allocated structure or is on a free page list.

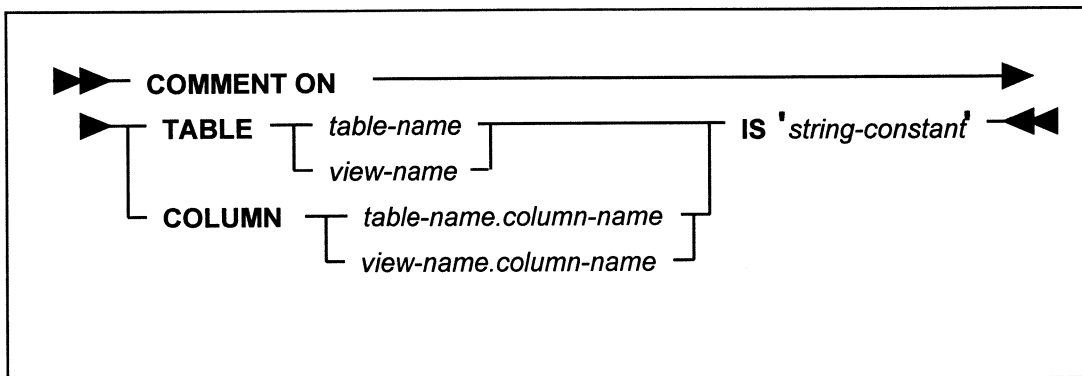
Example

```
SQL> CHECK DATABASE;
```

See Also

REORGANIZE

COMMENT ON



Description

This command places a comment in the REMARKS column of the SYSTABLES or SYSCOLUMNS tables. A comment can be added for a table, view, or column.

You must have ALTER privileges on the table to use this command.

In SQLTalk/Character, the REMARKS column is not displayed on the screen unless you enter the command COLUMN 4 WIDTH 20;.

The COMMENT ON command is like the LABEL ON command. The difference is that the REMARKS columns (maintained by COMMENT ON) is 254 characters long while the LABEL column (maintained by LABEL ON) is 30 characters long.

TABLE

This specifies the name of a table to which to add a comment.

COLUMN

This specifies the name of a column to which to add a comment.

IS 'string-constant '

The comment cannot be longer than 254 characters (maximum length of a VARCHAR column). The comment must be enclosed in *single* quotes.

Examples

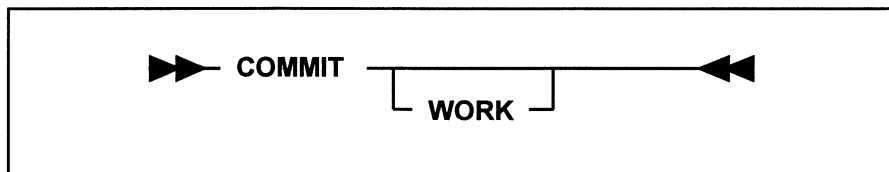
```
SQL> COMMENT ON TABLE EMP IS  
'CONTAINS EMPLOYEE PERSONAL INFORMATION';
```

```
SQL> COMMENT ON COLUMN EMP.TOTCOMP IS  
'CONTAINS TOTAL COMPENSATION FOR EMPLOYEE';
```

See Also

LABEL ON

COMMIT



Description

This command ends the current transaction (logical unit of work). A transaction has one or more SQL commands that must either all be executed or none at all.

This command commits all changes made to the database since the last COMMIT or ROLLBACK, or if none has been previously given, since the user connected to the database. This command commits the work for all cursors that the SQLTalk session or application has connected to the database.

The COMMIT operation applies to all SQL commands including data definition commands (CREATE, DROP, ALTER) and data control commands (GRANT, UPDATE, DELETE).

Locks are always released after a COMMIT except when cursor-context preservation is on.

Any user with CONNECT authority can execute this command.

WORK

This is a noise word that can be coded, but it has no effect. It is provided for DB2 compatibility.

Example

SQL> **COMMIT** (signals end of transaction and start of new one)

SQL Command ...

SQL Command ...

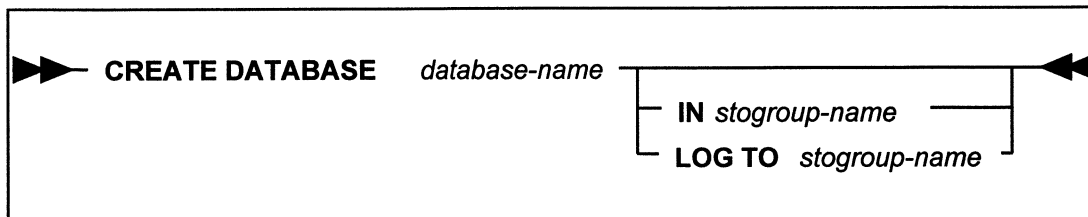
SQL Command ...

SQL> **ROLLBACK** (undoes previous three SQL commands)

See Also

ROLLBACK
SAVEPOINT

CREATE DATABASE



Description

This command physically creates a database on the server specified by the last SET SERVER command and installs the database on the network.

database-name

The name of the new database to be created. The maximum length of the database name is 8 characters.

Do not specify an extension for a database name (*demo.xyz* is invalid). SQLBase automatically assigns a database name extension of *.dbs*. SQLBase will store a database called *demo* in a file named *demo.dbs*. These rules do not affect partitioned databases.

Do not specify the name *main* for your database, since this is used to store control information for partitioned databases. Also, do not specify the name of an existing server for your database.

If a database file with the same name already exists (and the PAUSE option is turned ON), SQLTalk prompts you with the message:

```
Database file already exists. Overwrite it (Y/N)?
```

This enables you to decide if you really want to remove the existing database.

IN *stogroup-name*

You may specify a storage group for the database and a separate storage group for the log. If you do not specify a storage group, the default storage group in the main database is used if one exists. If either the *main* database does not exist or you do not specify a default storage group, the database is created and allocated as is done in SQLBase version 4.

LOG TO *stogroup-name*

You can place the log file on a disk separate from the database for better performance and integrity. If you specify a database storage group, but not one for the log, the log file space will be allocated using the database storage group.

□ *About New Databases*

In SQLBase, a database contains a database file placed in a sub-directory. The database file must have the extension *.dbs*, for example, *demo.dbs*. The name of the sub-directory must be the same as the database file name without the extension, for example, *demo*.

Usually the database sub-directory is placed in the `\SQLBASE` directory. This is the default, but you can change to any location using the *dbdir* keyword in *sql.ini*.

SQLBase expects the name of the *.dbs* file to be exactly the same as the name of the sub-directory.

Note: the above rules *only* apply to non-partitioned databases.

Examples

```
SQL> CREATE DATABASE ACCTPAY;
```

```
SQL> CREATE DATABASE ACCTPAY IN ACCTFILES LOG TO ACCTFILES;
```

See Also

CREATE DBAREA
CREATE STOGROUP
DROP DATABASE
INSTALL DATABASE
SET SERVER

CREATE DBAREA

```

CREATE DBAREA dbarea-name AS [filename] / [raw device] [SIZE megabytes]
  
```

Description

This command physically creates a database area of a specified size either on the server or in a raw partition. Commands or characters enclosed in brackets ([]) are optional.

The default size for a database area is 1 megabyte. The maximum size is limited by available disk space. If you are creating a database area on a raw device, you do not need to specify the size.

An error message appears if the file already exists or the disk space is already being used for another database area. An error also appears if a file of the specified size cannot be created or if the actual size of the raw device is smaller than the specified size of the area.

dbarea-name

The name of the new database area that you create. The maximum length of the database area name is 18 characters.

AS *filename/raw device*

Allows you to create a database area in a specified filename or raw device. If the filing system in use allows embedded blanks, you must use single quotes around the filename or raw device.

SIZE *megabytes*

Allows you to specify the size of the database area in megabytes. Do not attempt to create a database area which is larger than the actual amount of free disk space available on the specified device.

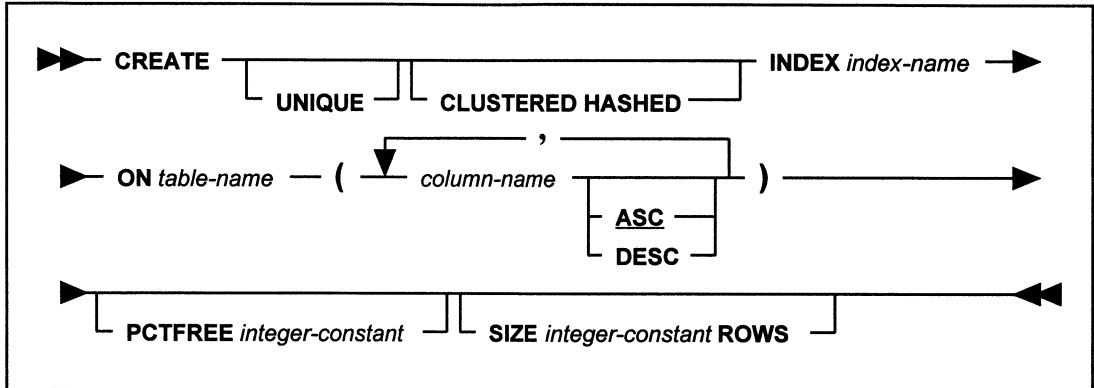
This is optional for UNIX.

Example

```
SQL> CREATE DBAREA ACCT1 AS PAYROLL SIZE 5;
```

See Also

```
ALTER DBAREA  
DROP DBAREA  
DROP DATABASE  
INSTALL DATABASE
```

CREATE INDEX**Description**

This command creates an index on one or more columns of a table.

Indexes optimize data retrieval since the data can be found without scanning an entire table. Indexes can also force unique data values in a column.

There is no limit on the number of indexes per table.

The maximum size of an index key is:

$$6 + \text{number of columns in index} + \text{sum of lengths of all columns in index}^2 \leq 255$$

The length of each column depends on its data type. For example, a CHAR(10) column is 10 bytes; any numeric column is 12 bytes; and any date/time column is 12 bytes.

To illustrate, if you are creating a concatenated index on these three columns:

LASTNAME	CHAR(20)
FIRSTNAME	CHAR(20)
MI	<u>CHAR(1)</u>
	41

would result in:

$$6 + 3 + 41 = 50$$

which is acceptable ($50 < 255$).

As another example, if you are creating an index on one column defined as:

LARGE CHAR(249)

would result in:

$$6 + 1 + 249 = 256$$

which is not allowed ($256 > 255$).

You must possess the INDEX privilege on the table to execute this command.

index-name

Each index name is a long identifier prefixed by an implicit qualifier which is the authorization-id of the creator of the index. The index name (including the qualifier) must be unique within a database.

table-name

View names *cannot* be used in the creation of an index.

UNIQUE

This keyword enforces unique key values within the table. It specifies that no combination of indexed columns in the table can be identical. If during index creation, or during an insert or update, this uniqueness property is violated, an error is returned.

CLUSTERED HASHED

This clause stores the data rows in locations based on the key hash value (clustering). A clustered hashed index speeds random access to rows in a table. If a table has a unique key that identifies each row, declaring a clustered hashed index on that key usually allows rows to be accessed with 1 disk read.

If this clause is not specified, a *B-tree (non-clustered)* index is created.

The table can grow or shrink, but clustered hashed indexes are intended for tables which are static or where an upper bound for the size of the table can be specified. A clustered hashed index can be specified for a non-unique key, but access only improves if there are relatively few rows for each key value.

Only one clustered hashed key can be created for a table.

A CREATE INDEX command that specifies a clustered hashed index must be given after the CREATE TABLE command and *before* any data is added to the table.

ASC DESC

This specifies whether the index is in ascending or descending order. ASC is the default order. This clause is only relevant for B-tree indexes.

PCTFREE *integer-constant*

The PCTFREE (percent free) clause specifies how much free space to leave in each index page when the index is initially built. After the index is built, key insertions and deletions can make the actual free space vary between 0% and 50%. If not specified, the default free space is 10%.

The PCTFREE keyword is followed by a number (between 0 and 99 inclusive) that specifies the percentage of free space to be left in each index page when the index is first built.

Normal values are 0-50%. Specifying 90-99% makes a binary index tree (2 entries per page) which results in the maximum height B-tree. This degrades retrieval performance.

This clause is ignored for a CLUSTERED HASHED index.

SIZE *integer-constant* **ROWS**

This controls the “expected” size of the index and is specified as a number of rows. If the size is too small, overflow pages are used and performance degrades. If the size is too large, overflow pages are not used, but disk space is wasted. This clause is only relevant for clustered hashed indexes.

This clause must be specified if the CLUSTERED HASHED clause is specified.

Functions in Indexes

An index can be created for one or more column values resulting from applying a function to the column. Functions for an index cannot be nested. Not all functions can be used to create an index.

Indexes created in this manner are used when the respective function is used in the WHERE clause. For functions which have arguments in addition to the table column (such as @SUBSTRING), all arguments must agree exactly between the CREATE INDEX and WHERE clause invocations in order for the index to be used.

A case-insensitive index results from applying the @UPPER or @LOWER function to the column in the CREATE INDEX command. When querying a column containing names that were entered using mixed case, and using the respective function in the WHERE clause to constrain the query, the rows returned include those in upper and lower case.

The following functions are allowed in CREATE INDEX.

**@CHAR
@CODE
@DATEVALUE
@DAY
@HOUR
@LEFT
@LENGTH
@LICS
@LOWER
@MICROSECOND
@MID
@MINUTE
@MONTH
@MONTHBEG
@PROPER
@QUARTER
@QUARTERBEG
@RIGHT
@SECOND
@STRING
@SUBSTRING
@TIMEVALUE
@TRIM
@UPPER
@VALUE
@WEEKBEG
@WEEKDAY
@YEAR
@YEARBEG
@YEARNUM**

Examples

Create an index named CHKINDX using the CHKNUM column.

```
SQL> CREATE INDEX CHKINDX ON CHECKS (CHKNUM);
```

Create a concatenated index composed of CLASS and SECTION.

```
SQL> CREATE INDEX CSX ON ROSTER (CLASS, SECTION);
```

Create a descending index on the PARTNO column of the PARTS table. Disallow duplicate part numbers.

```
SQL> CREATE UNIQUE INDEX PARTIDX  
ON PARTS (PARTNO DESC);
```

This example illustrates the creation and use of a case insensitive index.

```
SQL> CREATE INDEX EMPIX ON EMP (@UPPER(NAME));
```

In the above example, an upper case index is created for NAME. This index is used when the @UPPER function is specified in the WHERE clause of a SELECT, thereby using case insensitive sort order and using the index. The example below illustrates this.

```
SQL> SELECT NAME FROM EMP  
WHERE @UPPER(NAME) = 'JONES' ORDER BY 1;  
NAME  
====  
JONES  
Jones  
jones  
3 Rows Selected
```

Create an index on the first 3 characters of a string.

```
SQL> CREATE INDEX CODEIDX ON PEOPLE (@LEFT(PHONE, 3));
```

The select command that uses this index must agree with the definition of CODEIDX.

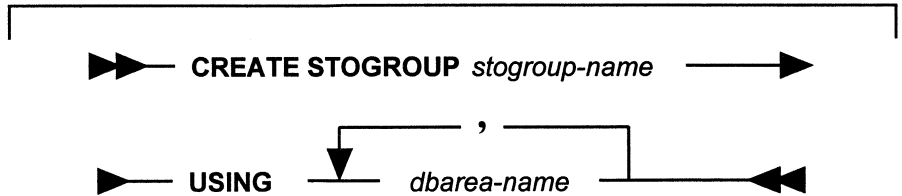
Get all the rows for people in the 415 area code.

```
SQL> SELECT * FROM PEOPLE WHERE @LEFT(PHONE,3) = '415';
```

See Also

CREATE TABLE

CREATE STOGROUP



Description

This command creates a storage group. If the volumes containing the database areas are not mounted, an error occurs when you try to create a database.

stogroup-name

This names the storage group that you create. The maximum length of the storage group name is 18 characters.

dbarea-name

This is a list of database areas. Database areas must already exist.

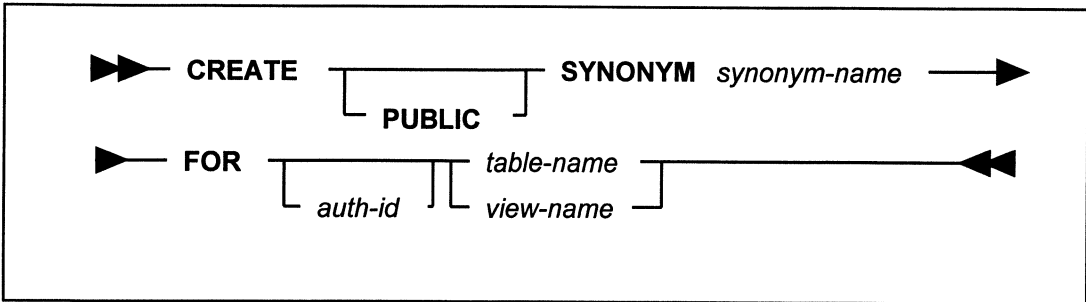
Example

```
SQL> CREATE STOGROUP ACCTDEPT USING ACCT1, ACCT2;
```

See Also

```
ALTER STOGROUP
DROP STOGROUP
```

CREATE SYNONYM



Description

This command defines an alternate name for a table or view. This lets you reference another user's tables or views without having to use the qualified name (*auth-id.table-name*).

You can only use the synonym in a command executed by the creator of the synonym.

PUBLIC

This allows you to access the table through the synonym without fully qualifying the table name with the authorization-id of the owner.

You must own the table or be a DBA or SYSADM to create a PUBLIC synonym.

You must have the appropriate privileges on the underlying table to access it through a PUBLIC synonym.

synonym-name

The synonym is named in the same manner as a table or a view. It must not be the same as any other synonym, table, or view that you own.

table-name

The table-name must name an existing view or table in the database.

view-name

The view-name must name an existing view or table in the database.

Examples

```
SQL> CREATE SYNONYM DEPT FOR USER1.DEPT;
```

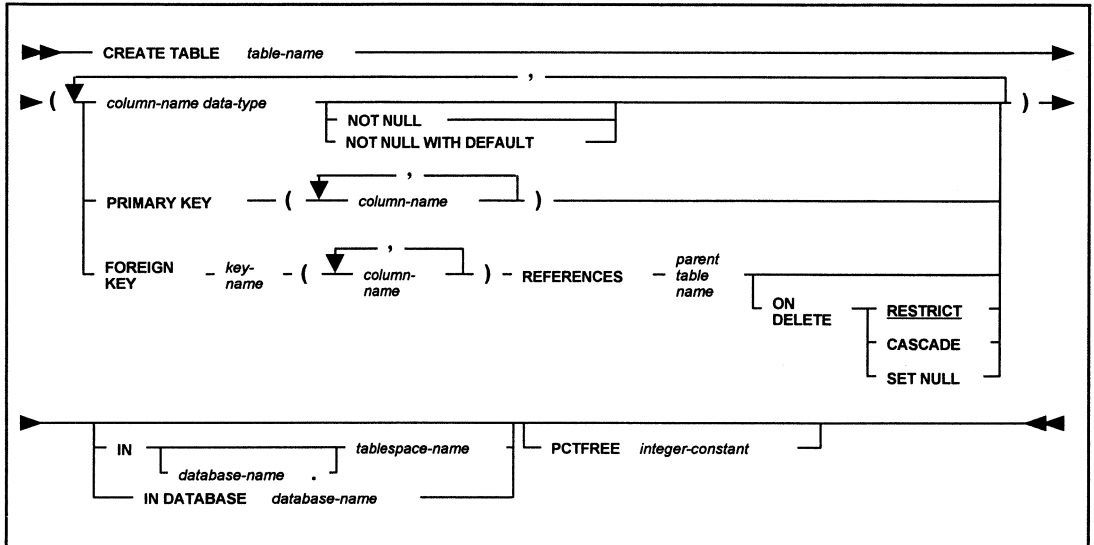
```
SQL> CREATE SYNONYM GL FOR GENERAL_LEDGER;
```

```
SQL> CREATE PUBLIC SYNONYM CUSTOMER  
FOR SYSADM.CUSTOMER;
```

See Also

CREATE TABLE

CREATE TABLE



Description

This command creates a table with the specified columns.

You can define a maximum of 250 columns for each table.

You must have RESOURCE, SYSADM, or DBA authority to execute this command.

When you use `CREATE TABLE` with referential constraints, you should define a foreign key with the same specifications as the primary key of the parent table. A referential constraint defines the rules for a relationship between the primary key of a parent table and a foreign key of a dependent table. A referential constraint requires that for each row in a dependent table, the value of the foreign key must appear as the primary key of a row in the parent table.

You must designate the parent table name when you define the foreign key. In addition, you must already have a parent table definition with a primary key and a unique index. You can also specify the delete rule of the referential constraint. The default rule is RESTRICT.

table-name

A fully-qualified SQL table name has the form:

authorization-id.table-name

The authorization-id is a qualifier denoting the creator of the table. The combined authorization-id.table name must form a unique name which does not identify any existing table, view, or synonym in the database.

When you create a table, if you do not specify the authorization-id, your default authorization-id is automatically prefixed to the table name.

column-name

A column name must begin with a letter (A through Z and the special characters #, @ and \$) and must not exceed 18 characters.

A fully-qualified column name has the form:

table-name.column-name

You can use the unqualified column name when you define the table, and it must be a long identifier (18 characters maximum). Each column name must be unique within a table.

data-type

A column can be one of the following data types. These data types are described in the section called *Data Types* in chapter 6.

<p>CHAR (length) VARCHAR (length) SMALLINT INTEGER LONG VARCHAR DECIMAL [(precision, scale)] DATETIME DATE TIME TIMESTAMP FLOAT NUMBER</p>
--

Columns defined as CHAR or VARCHAR require a length attribute.

Columns defined as DECIMAL have a default size attribute of 5,0; any other precision and scale must be declared in parentheses.

SQLBase does not allocate the full space for a row when it is inserted with null columns. An application that inserts a row with uninitialized columns and later writes values to those columns will expand the row with extent pages. To avoid the extent pages, the application should write blank-filled columns on the first INSERT of each row.

PRIMARY KEY

This creates the primary key for a table.

The following rules apply to primary keys:

- The values of the primary key must be unique — no two rows of a table can have the same key values.
- A table can have only one primary key.
- The primary key can be made up of one or more columns in a table.
- Each column in the primary key should be classified with the NOT NULL constraint.

FOREIGN KEY

This specifies the foreign key for a table.

Every value in a foreign key must match some value in the primary key from which the foreign key column originates.

The parent table must have a unique index on the primary key.

The following rules apply to foreign keys:

- A table can have many foreign keys.
- A foreign key can be made up of one or more columns of a table.
- The foreign key and primary key must contain the same number of columns, and the foreign key's data types must match those of the primary key on a one-to-one basis.
- A foreign key column may or may not be NULL.

- A foreign key value is NULL if any part is NULL.

REFERENCES

This identifies the parent table in a relationship and defines the necessary constraints. The REFERENCES clause must accompany the FOREIGN KEY clause.

NOT NULL

If you declare a column NOT NULL, it requires data to be present in the column every time a row is added to the table. If omitted, the column can contain null values, and its default value is the null value.

NOT NULL WITH DEFAULT

The NOT NULL WITH DEFAULT clause prevents a column from containing null values and allows a default value other than the null value.

The default value used depends on the data type of the column, as follows:

Data Type	Default Value
Numeric	0 (zero)
Date/Time	Current date/time
Character	One blank

The NOT NULL WITH DEFAULT clause causes the INSERT to insert the above defaults. If the column is not specified in the INSERT command, SQLBase puts a 'D' in the NULLS columns of the SYSCOLUMNS table and treats it like a NOT NULL field.

The NOT NULL WITH DEFAULT clause is compatible with DB2.

IN DATABASE *database-name*

IN [*database-name*] *tablespace-name*

SQLBase accepts these clauses but ignores them. The IN clauses are compatible with DB2.

ON DELETE

This specifies the DELETE rules for the table.

The DELETE rules are optional.

The default is RESTRICT.

DELETE rules are only used to define a foreign key.

CASCADE

This deletes the selected rows first, and then deletes the dependent rows, honoring the delete rules of their dependents.

RESTRICT

This specifies that a row can be deleted if no other row depends on it. If a dependent row exists in the relationship, the delete will fail.

SET NULL

This specifies that for any delete performed on the primary key, matching values in the foreign key are set to null.

PCTFREE *integer-constant*

This sets the free space left in each table row page when it is first filled. The default free space is 10 percent.

If you plan to expand rows later by adding more columns or increasing the width of existing columns, this feature leaves space for expansion so that extension pages are not needed.

Also, for small, heavily-accessed tables where page locking can cause contention, the PCTFREE option can force fewer rows to be assigned to each page which reduces contention.

The PCTFREE value must be between 0 and 99.

Examples

Create a table EMPLOYEE for storing employee data and EMPSAL for keeping a salary history.

```
SQL> CREATE TABLE EMPLOYEE
  (EMPNO INTEGER NOT NULL,
  SOC_SEC_NO CHAR (11),
  LNAME VARCHAR(25),
  FNAME CHAR(10),
  MNAME CHAR(10),
  DEPTNO SMALLINT,
  HIREDATE DATE,
  INTERVIEW_COMMENTS LONG VARCHAR);
```

```
SQL> CREATE TABLE EMPSAL
  (EMPNO INTEGER NOT NULL,
  NEWSAL DECIMAL(8,3),
  CHANGEDATE DATE);
```

Create a table that allows the foreign key EMPNO in the EMPSAL table to reference EMPNO in the EMPLOYEE table, with a DELETE CASCADE rule.

```
SQL> CREATE TABLE EMPLOYEE
(EMPNO INT NOT NULL,
SOC_SEC_NO CHAR (11),
LNAME VARCHAR(25),
FNAME CHAR(10),
MNAME CHAR(10),
DEPTNO SMALLINT,
HIREDATE DATE,
INTERVIEW_COMMENTS LONG VARCHAR,
PRIMARY KEY (EMPNO));

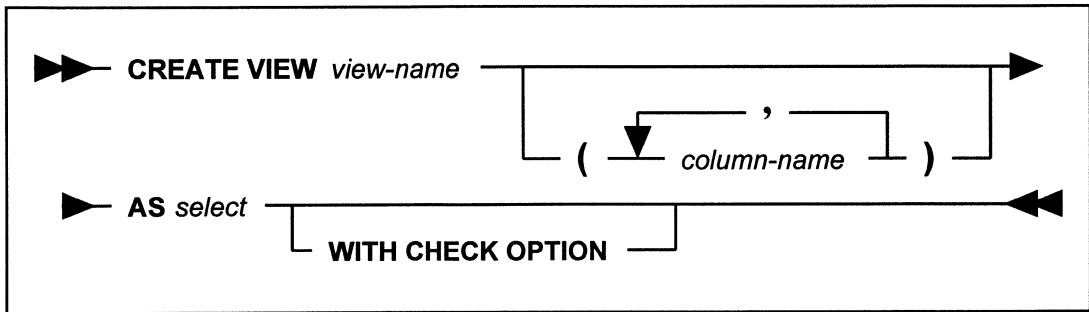
CREATE UNIQUE INDEX EMP_IDX ON EMPLOYEE (EMPNO);

CREATE TABLE EMPSAL
(EMPNO INT, SALARY INT,
FOREIGN KEY (EMPNO) REFERENCES EMPLOYEE
ON DELETE CASCADE);
```

See Also

```
ALTER TABLE
CREATE INDEX
DELETE
UPDATE
```

CREATE VIEW



Description

This command creates a view on one or more tables or views.

By granting certain privileges on a view instead of on base tables, you can selectively restrict access to the data in the base tables. See `GRANT` (Table Privileges) for more information.

You can modify tables through a view only if the view references a single table name in the `FROM` clause of the `SELECT` command, and the view columns are not derived from a function or arithmetic expression.

If you create the view from a table join, or it has derived columns, it is read-only and you cannot update the underlying tables through it.

To create a view, you must possess the corresponding `SELECT`, `INSERT`, `UPDATE`, and `DELETE` privileges on the columns of the base tables that comprise the view.

view-name

The view name has the form:

authorization-id.view-name

The view name, including the authorization-id, must not be the name of an existing view in the database.

column-name

Specify column names if you want to give different names to the columns in the view. If you do not specify column names, the columns of the view have the same names as those of the result table of the SELECT command.

If the results of the SELECT command have duplicate column names (as can occur with a join), or if a column is derived from a function or arithmetic expression, you must give names to all the columns in the view.

select

A SELECT command defines the view. The view has the rows that would result if the SELECT command were executed. See the description of SELECT for an explanation of this clause.

You cannot use the ORDER BY clause in a view definition.

A view is considered read-only and cannot be updated if its definition involves any of these:

- A FROM clause that names more than one table or view.
- A DISTINCT keyword.
- A GROUP BY clause.
- A HAVING clause.
- An aggregate function.

WITH CHECK OPTION

This causes all inserts and updates through the view to be checked against the view definition and rejected if the inserted or updated row does not conform to the view definition. If the clause is omitted, then no checking occurs.

If a view is read-only, or if the SELECT command includes a subselect, the WITH CHECK OPTION must *not* be specified. If the view definition allows updates to some columns, the WITH CHECK OPTION applies *only* to the updates.

Examples

This view is the result of a two-table join, and is therefore read only. Since the column names of the view are not specified, they are the same as the column names in the underlying table.

```
SQL> CREATE VIEW PROJEMP AS
SELECT EMPNO, PROJNAME, START_DATE
FROM EMP, PROJ
WHERE EMP.PROJNO = PROJ.PROJNO;
```

The next view example has different column names than the underlying table.

The creator of the view does not have to be the creator of the underlying table, since the fully-qualified table name is used. Using the fully-qualified name is a good idea when creating views if the views will be used by a variety of users.

Since this view references only one table, you could update the EMP table through this view, given the proper privileges.

```
SQL> CREATE VIEW BIRTHDAYS (NAME, BIRTHDAY) AS
SELECT FIRST_NAME, DATE_BORN
FROM PERSONNEL.EMP;
```


This next view contains a column (TOTSAL) which is derived from the application of an aggregate function. This makes it read only.

```
SQL> CREATE VIEW PROJ_SAL (PROJNO, TOTSAL) AS
SELECT PROJECT, SUM(SAL) FROM PERSONNL.EMP
GROUP BY PROJECT;
```

This view uses the WITH CHECK OPTION clause. Any update of the column TARGET_DATE is checked to make sure the value is a date later than the current date.

```
SQL> CREATE VIEW CURPROJ AS
SELECT * FROM PROJECTS
WHERE TARGET_DATE > SYSDATE
WITH CHECK OPTION;
```

See Also

CREATE TABLE
SELECT

DEINSTALL DATABASE

▶▶ DEINSTALL DATABASE *database-name* ◀◀

Description

This command removes the database name from the network (removes the database name from the list of names for which the server is listening) and updates the *dbname* keyword in *sql.ini*.

This command does not physically delete the database, but it makes the database unavailable to users.

You cannot DROP a database that is open (a database that has a user connected).

This command deinstalls the database on the server that you specified by the last SET SERVER command.

database-name

The name of the database that you dropped.

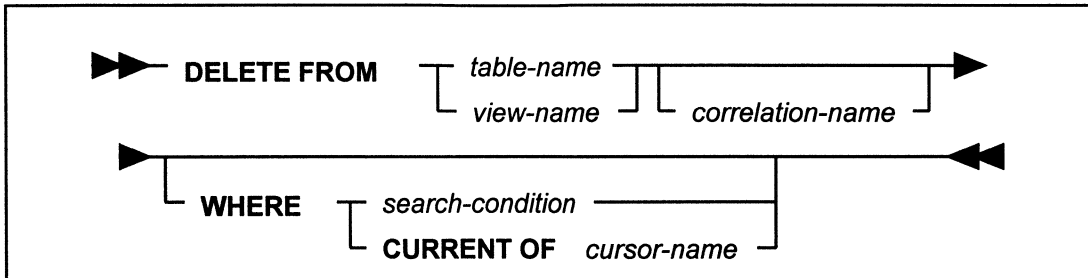
Example

```
SQL> DEINSTALL DATABASE CUSTOMER;
```

See Also

CREATE DATABASE
DROP DATABASE
INSTALL DATABASE
SET SERVER

DELETE



Description

This command deletes one or more rows from a single table or view. All rows that satisfy the search condition are deleted from the table, or if a view is specified, from the base table of that view.

You must possess the DELETE privilege on a table to execute this command.

table-name

Any table name can be specified for which the user has delete privileges. The name cannot identify a system table.

view-name

Any view name can be specified for which the user has delete privileges. The name cannot identify a read-only view.

correlation-name

A correlation name can be used within a search condition to designate the table or view.

WHERE *search-condition*

The search condition qualifies a set of rows for deletion.

A DELETE command with this clause is called a “searched DELETE.”

If you do not specify a search condition, all the rows in the specified table or view are deleted.

See the section called *Search Conditions* in chapter 6 for more.

WHERE CURRENT OF *cursor-name*

A DELETE command with this clause is called a “positioned DELETE” or a “cursor-controlled DELETE.”

This type of update requires two open cursors:

- Cursor 1 is associated with a SELECT command. The current row references the row of the most recent fetch.
- Cursor 2 is associated with the DELETE command.

A cursor-name must be associated with cursor 1 before this command can be executed.

You can only use a CURRENT OF clause if all of the following are true for the corresponding SELECT command:

- The cursor must be named or be in result set mode.
- The SELECT command cannot contain joins, GROUP BY, DISTINCT, SET functions, UNION, or ORDER BY.
- Any subselect in the SELECT command must satisfy the previous condition.

Examples

This command deletes employee 1234 from the EMP table.

```
SQL> DELETE FROM EMP WHERE EMPNO = 1234;
```

Delete employees whose performance rating is less than 5 from the BONUS table.

```
SQL> DELETE FROM BONUS
WHERE EMPNO IN
(SELECT EMPNO FROM EMP WHERE PERFGRADE <= 4);
```

Delete all rows from the table.

```
SQL> DELETE FROM TESTTABLE;
```

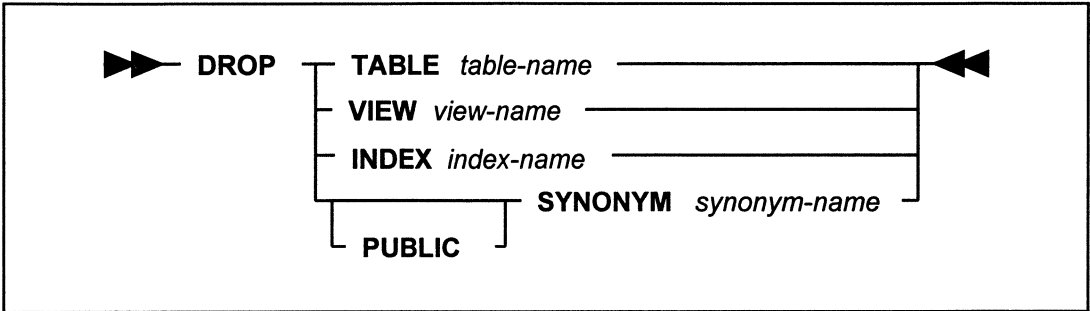
Delete the row referenced by the current fetch, using the cursor named EMPCURSOR.

```
SQL> SET SCROLL ON;
SQL> SET CURSORNAME EMPCURSOR;
SQL> PREPARE SELECT FROM EMP;
SQL> PERFORM;
SQL> SET SCROLLROW 0;
SQL> FETCH 1;
SQL> CONNECT 2;
SQL> DELETE FROM EMP WHERE CURRENT OF EMPCURSOR;
```

See Also

```
CREATE TABLE
SET CURSORNAME
```

DROP



Description

This command removes the specified object from the database.

Precompiled commands that reference the dropped tables, views, indexes, and synonyms are *not* automatically dropped.

An object can only be dropped by its creator or by a user with SYSADM or DBA authority.

In a database with referential constraints, dropping a table drops its primary key. This also drops any foreign keys in other tables that reference the parent table. When the parent table of the relationship is dropped, or when the primary key of the parent table is dropped, the referential constraint is also dropped.

DROP TABLE drops all constraints in which the table is a parent or dependent. Dropping a table is not the same as deleting all its rows. Instead, when you drop a table, you also drop all the relationships in which the table is involved, either as a parent or dependent. This can affect application programs that depend on the existence of a parent table, so use caution with the DROP TABLE command.

TABLE *table-name*

This removes the table, all synonyms and indexes defined for the table, and all privileges granted on the table. Also, any views whose definition depends either partially or wholly on the dropped table are also dropped. All privileges on the tables are also removed.

System tables cannot be dropped.

VIEW *view-name*

This removes the view from the system catalog. Also, any views whose definition depends either partially or wholly on the dropped view are also dropped. All privileges on the views are also removed.

INDEX *index-name*

This removes the index. Indexes on system tables cannot be dropped. The existence of views and tables are not affected.

SYNONYM *synonym*

This removes the synonym. Views based on the synonym are also dropped.

PUBLIC

The removes the PUBLIC synonym. Views based on the synonym are also dropped.

Examples

```
SQL> DROP INDEX CHKINDX;  
SQL> DROP VIEW MYEMP;  
SQL> DROP SYNONYM EASY_TO_REMEMBER;  
SQL> DROP PUBLIC SYNONYM EMP;  
SQL> DROP TABLE EMP;
```

See Also

```
CREATE INDEX  
CREATE SYNONYM  
CREATE TABLE  
CREATE VIEW
```

DROP DATABASE

 **DROP DATABASE** *database-name*

Description

This command physically deletes the entire database directory for a database *including* all associated transaction log files on the server specified by the last SET SERVER command. If the log is redirected, the log directory for the database is also completely removed.

database-name

The name of the database to be deleted.

Example

```
SQL> DROP DATABASE ACCTPAY;
```

See Also

```
CREATE DATABASE  
DEINSTALL DATABASE  
INSTALL DATABASE  
SET SERVER
```

DROP DBAREA

▶▶ DROP DBAREA *dbarea-name* **◀◀**

Description

This command physically deletes the entire database area if none of its file space is currently allocated.

dbarea-name

The name of the database area to delete.

Example

```
SQL> DROP DBAREA ACCT1;
```

See Also

ALTER DATABASE
CREATE DATABASE
CREATE DBAREA
SET DEFAULT STOGROUP

DROP STOGROUP

▶▶ DROP STOGROUP *stogroup-name* **◀◀**

Description

This command deletes the storage group if it is not being used by any database and it is not the default storage group. This command does not affect any existing space allocations for databases or logs.

stogroup-name

The name of the storage group to be deleted.

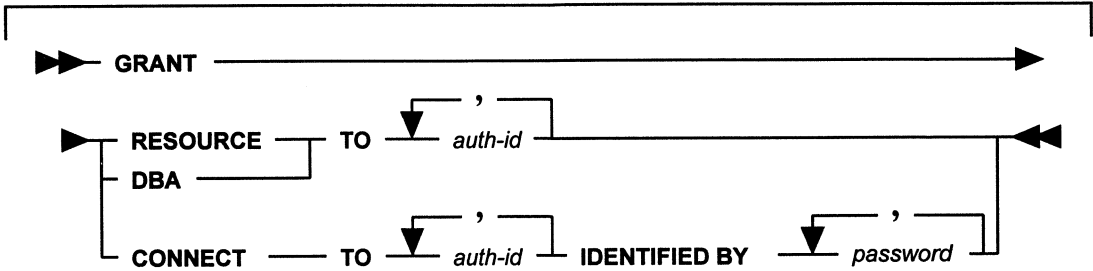
Example

```
SQL> DROP STOGROUP ACCTDEPT;
```

See Also

```
ALTER DATABASE  
CREATE DATABASE  
CREATE STOGROUP  
SET DEFAULT STOGROUP
```

GRANT (Database Authority)



Description

This form of the GRANT command assigns users of the database and assigns their authority level. Authority level means the types of operations a user can perform (such as logging on, creating tables, or creating users).

A different form of the GRANT command assigns privileges for individual tables.

This form of the GRANT command can only be given by SYSADM. SYSADM can create new users and change the authority levels and table privileges of existing users. This is the highest authority level and it is preassigned by SQLBase to SYSADM.

A user cannot be granted SYSADM authority. The username SYSADM cannot be changed and there can only be one SYSADM for a database. The only thing that can be changed for SYSADM is the password.

If you GRANT a user the RESOURCE or DBA authority, it does not take effect until the next time the user connects.

Authority Levels

The following authority levels can be granted by SYSADM:

CONNECT	This authority level must be granted before any other. It allows the user to log onto the database and exercise any of the privileges assigned for <i>specific</i> tables. The IDENTIFIED BY clause is required for granting CONNECT.
RESOURCE	This gives a user the right to create tables, to drop those tables, and to grant, modify or revoke privileges to those tables for valid users of the database. A user with RESOURCE authority automatically has all privileges on tables that he or she has created.
DBA	<p>This level of authority automatically assigns all privileges on any table in the database to a user, including the right to grant, modify, or revoke the table privileges of any other user in the database.</p> <p>However, a DBA <i>cannot</i> create new users or change a password or authority level of an existing user. These privileges are restricted to SYSADM.</p>

authorization-id

The authorization-id is the username that gives a user authorization to connect to a database. The authorization-id SYSADM is preassigned by the system and reserved for the SQLBase "superuser."

IDENTIFIED BY *password*

This is required *only* when granting CONNECT authority to a user and is the phrase used to introduce the new user's password.

password

A GRANT CONNECT command must include a password. The password can be any valid SQL short identifier. To change the password of a user, grant that user CONNECT authority with the new password.

When a database is first created, the original creator of the database (SYSADM) is always identified by the password SYSADM. The owner of the database can change the password to a private password before granting authority to any other user.

The password is stored in the system catalog and can be read by a user with SYSADM or DBA authority. However, the password is encrypted and a user must use the @DECRYPT function to read it. Note that passwords are encrypted when transmitted across a network.

Examples

Create two new users, JOE and JEAN. JOE is given the password SWAN and JEAN is given the password EAGLE.

```
SQL> GRANT CONNECT TO JOE, JEAN  
IDENTIFIED BY SWAN, EAGLE;
```

Give Jean the privilege to CREATE tables.

```
SQL> GRANT RESOURCE TO JEAN;
```

Give Joe DBA privilege, which includes RESOURCE privileges.

```
SQL> GRANT DBA TO JOE;
```

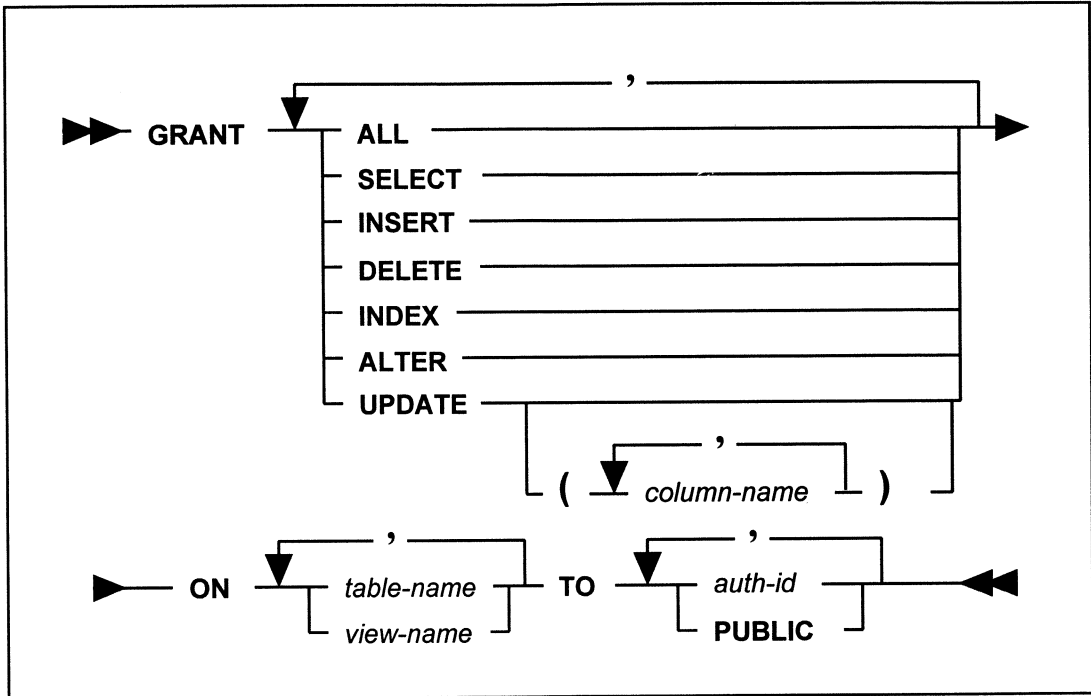
Change SYSADM's password.

```
SQL> GRANT CONNECT TO SYSADM IDENTIFIED BY CONDOR;
```

See Also

GRANT (Table Privileges)
REVOKE

GRANT (Table Privileges)



Description

This form of the GRANT command gives a user one or more specified privileges for a table or view.

You cannot GRANT the INDEX and ALTER privileges on views.

Table privileges can be granted by any user who has the authority to do so. A user with DBA authority can grant privileges on any tables or views in the database. A user with RESOURCE authority (but without DBA authority) can grant privileges only on tables created by him or on views that are based completely on tables created by him. A user with only CONNECT authority cannot grant privileges. Nor does he have privileges to any tables or views unless he is explicitly granted such privileges with a GRANT command.

A different form of the GRANT command assigns privileges for a database.

The system catalog tables are owned by the creator of the database (SYSADM) so their name must be prefixed with the authorization-id SYSADM. For a description of the system catalog tables, see chapter 10 of this manual.

Privileges

The following privileges can be assigned.

Privilege	Description
SELECT	Select data from a table or view.
INSERT	Insert rows into a table or view.
DELETE	Delete rows from a table or view.
UPDATE	Update a table and (optionally) update only the specified columns.
INDEX	Create or drop indexes for a table.
ALTER	Alter a table.
ALL	Exercise all the above for a table.

Note that you cannot GRANT the INDEX or ALTER privileges for a view. You should GRANT these privileges directly on the base tables.

table-name

Table names (including an implicit qualifier) must identify a table that exists in the database.

view-name

View names (including any implicit qualifier) must identify a view that exists in the database.

column-name

This is a column in the tables or views specified in the ON clause. Each column name must be unqualified and each column name must be in every table or view identified in the ON clause.

authorization-id

The authorization-id must refer to a user who has been granted at least CONNECT authority to the database.

PUBLIC

This means all users. By granting a privilege to PUBLIC, it means that all current and future users have the specified privilege on the table or view.

Examples

Give Linda privilege to read (SELECT from) the CHECKS table and change (UPDATE) two columns, AMOUNT and PAYDATE.

```
SQL> GRANT SELECT, UPDATE (AMOUNT, PAYDATE)
ON CHECKS TO LINDA;
```

Give JOE global privileges on the tables CHECKS and INVOICES.

```
SQL> GRANT ALL ON CHECKS, INVOICES
TO JOE;
```

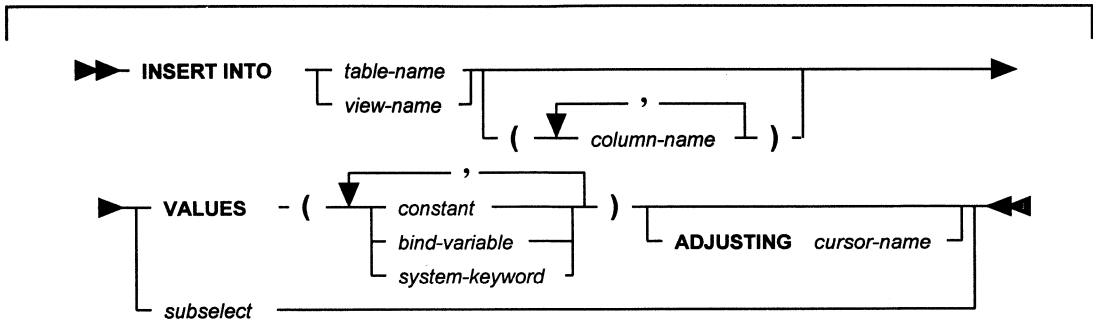
Allow all users (PUBLIC) to read SYSADM.SYSTABLES.

```
SQL> GRANT SELECT
ON SYSADM.SYSTABLES TO PUBLIC;
```

See Also

GRANT (Database Authority)
REVOKE (Database Privileges)
REVOKE (Table Privileges)

INSERT



Description

This command inserts rows of data into a table or view.

For a view, the rows are inserted into the base table.

If the insertion of a row causes a unique index to become non-unique, or if the row does not satisfy the definition of a view that has the WITH CHECK OPTION, then the insert is not allowed.

You must possess INSERT privileges on the table to execute this command.

If the database has referential constraints, and you insert data into a parent table with a primary key:

- Do not enter identical values for the primary key.
- Insert only non-null values for any column of the primary key.

If you insert data into a dependent table with foreign keys:

- Each non-null value inserted into a foreign key column must be equal to a value in the primary key.
- The entire foreign key is regarded as null if any column in the foreign key is null. The INSERT will work if all foreign keys are null (as long as there are no unique index violations).
- An INSERT into either the parent table or dependent table will not work if the index enforcing the primary key of the parent table has been dropped.

table-name

Table names (including any qualifier) must reference a table that exists in the database but they cannot be any system catalog tables.

column-name

This is one or more column names in the specified table or view for which you provide insert values. You can name the columns in any order.

If you omit the column list, you are implicitly using a list of all the columns, in the order they were created in the table or view, and must therefore provide a value for each column.

view-name

View names (including any qualifier) must reference a view that exists in the database but they *cannot* be any system catalog views.

VALUES

This clause contains one row of column values to be inserted. The values can be constants, bind variables, or system keywords.

Separate the column values with commas. Do *not* put a space before or after the comma.

SQLBase will convert the values to the target data type wherever possible.

The SQLBase system keywords NULL, USER, SYSDATE, SYSTIME and SYSDATETIME can be used in the VALUES clause.

subselect

This clause inserts the rows of a result table produced by a SELECT command. The number of columns retrieved must match the number of columns being inserted. Similarly, the rows of the select must match the create definition with respect to data types and length of data. SQLBase attempts data type conversions where possible.

You cannot specify a LONG VARCHAR column in a subselect.

You cannot use an ORDER BY clause in a subselect.

ADJUSTING *cursor-name*

This clause is used for result set programming. This clause allows a user to INSERT a row without invalidating the current result set.

INSERTed rows are added to the end of the result set and the database.

You cannot perform a multirow insert with an ADJUSTING clause and a subselect.

You cannot perform an insert with an ADJUSTING clause and a subselect with a join.

Examples

This SQL command inserts one complete row into the EMP table.

```
SQL> INSERT INTO EMP VALUES  
(1234, 'JOHN', 01-JAN-1986, 2500);
```

If all columns in the row are not being filled, you must specify the column names.

```
SQL> INSERT INTO EMP (EMPNO, NAME, HIREDATE) VALUES  
(1234, 'JOHN', 01-JAN-1986);
```

Use a subquery to derive rows for insertion.

```
SQL> INSERT INTO PARTTIMERS SELECT EMPNO, NAME FROM EMP  
WHERE STATUS = 'PT';
```

This command uses bind variables to insert multiple rows of data.

```
SQL> INSERT INTO EMP VALUES (:1, :2, :3, :4)  
\  
1234, John, 01-JAN-1986, 2500  
1235, Jean, 01-JAN-1986, 2650  
/
```

The example below uses an ADJUSTING clause and a result set.

```
SQL> SET CURSORNAME MYCUR;  
  
SQL> SET SCROLL ON;  
  
SQL> SELECT * FROM PRESIDENT;  
  
SQL> SET SCROLLROW 3;  
  
SQL> FETCH 10;
```

A different cursor is used to INSERT, preserving the result set.

```
SQL> CONNECT DEMO 2;
```

INSERT into result set.

```
SQL> INSERT INTO PRESIDENT (PRES_NAME) VALUES ('George  
Bush') ADJUSTING MYCUR;
```

Return to the result set cursor.

```
SQL> USE 1;
```

Since the result set is unaffected, we can fetch without reissuing the SELECT.

```
SQL> FETCH 5;
```

See Also

```
SELECT  
SET CURSORNAME
```

INSTALL DATABASE

▶▶ **INSTALL DATABASE** *database-name* ◀◀

This command assumes that the specified database exists and installs the database name on the network, adding a *dbname* keyword in *sql.ini*, and making the database accessible to users.

The database is installed on the server specified by the last SET SERVER command.

database-name

The name of the database to be installed.

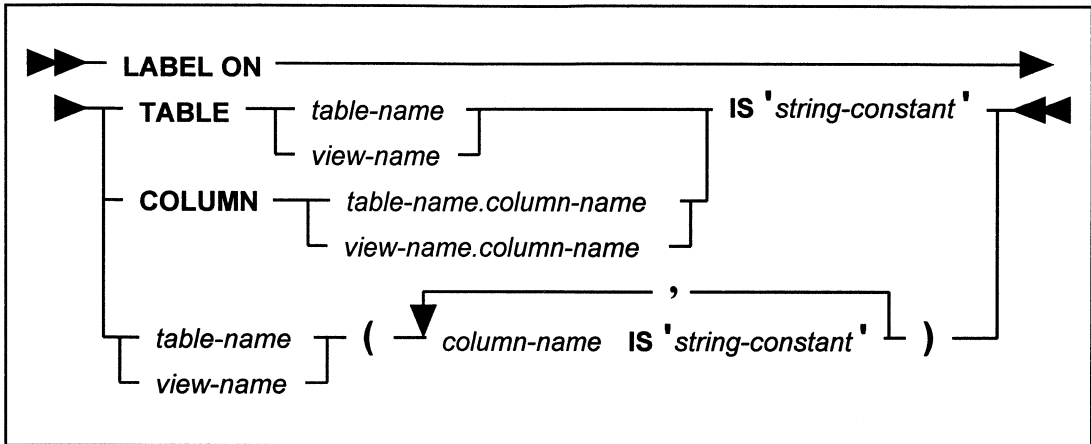
Example

```
SQL> INSTALL DATABASE CUSTOMER;
```

See Also

DEINSTALL DATABASE
DROP DATABASE
INSTALL DATABASE
SET SERVER

LABEL



Description

This command adds or replaces labels in the system catalog descriptions of tables, views, columns, or sets of columns.

The system catalog can maintain a comment on every table, view, or column in the SYSTABLES or SYSCOLUMNS tables. The LABEL command places a comment in the LABEL column of the SYSTABLES or SYSCOLUMNS tables.

The COMMENT ON command is like the LABEL ON command. The difference is that the REMARKS columns (maintained by COMMENT ON) is 254 characters long while the LABEL column (maintained by LABEL ON) is 30 characters long.

The LABEL column can be retrieved through an API call.

table-name

You can use this to specify the name of a table that you want to add a LABEL column for.

view-name

You can use this to specify the name of a view that you want to add a LABEL column for.

table-name.column-name

You can use this to specify the name of a column in a table that you want to add a LABEL column for.

view-name.column-name

You can use this to specify the name of a column in a view that you want to add a LABEL column for.

IS 'string-constant'

You can use this to specify the comment. It can be up to 30 characters.

□ Adding labels for more than one column

Do not specify the keywords TABLE or COLUMN. Give the table or view name and then, in parentheses, specify the label for each column. Separate each label definition with a comma. See the examples.

Examples

```
SQL> LABEL ON TABLE EMP IS  
'CONTAINS EMP. INFO.');
```

```
SQL> LABEL ON COLUMN EMP.TOTCOMP IS  
'CONTAINS TOT. COMP.';
```

```
SQL> LABEL ON EMP  
(TOTCOMP IS 'CONTAINS TOT. COMP.',  
STARTDATE IS 'STARTING DATE');
```

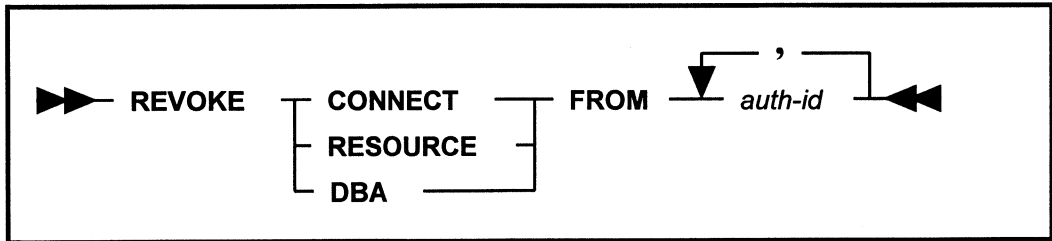
The example below selects all labels from the SYSCOLUMNS system catalog table. Note that you must enclose the column name (LABEL) in double-quotes and it must be in upper-case:

```
SQL> SELECT NAME, TBNAME, "LABEL" FROM SYSCOLUMNS;
```

See Also

COMMENT ON

REVOKE (Database Authority)



Description

This form of the REVOKE command removes the authority level of a user who has previously been granted authority for a database.

Only a user with SYSADM authority can revoke the DBA authority of another user.

If you REVOKE a user's RESOURCE or DBA authority, it does not take effect until the next time the user connects.

Authority Levels

The authority levels DBA, RESOURCE and CONNECT can be revoked by SYSADM. See the table on the next page.

Privilege	Description
SYSADM	This authority level cannot be removed. It is assigned by the system when the database is created.
DBA	Revoking this authority means the user can no longer create or drop tables, or grant or revoke privileges from users. However, the user retains CONNECT authority. All tables and views previously created by this user remain.
RESOURCE	Revoking this authority means the user no longer has the right to create or drop tables. However, the user retains CONNECT authority. Previously-created tables and views remain.
CONNECT	Revoking this authority means that the user is no longer authorized to access the database. All privileges on tables and views must be revoked from a user before revoking CONNECT authority. CONNECT authority cannot be revoked while a user owns tables.

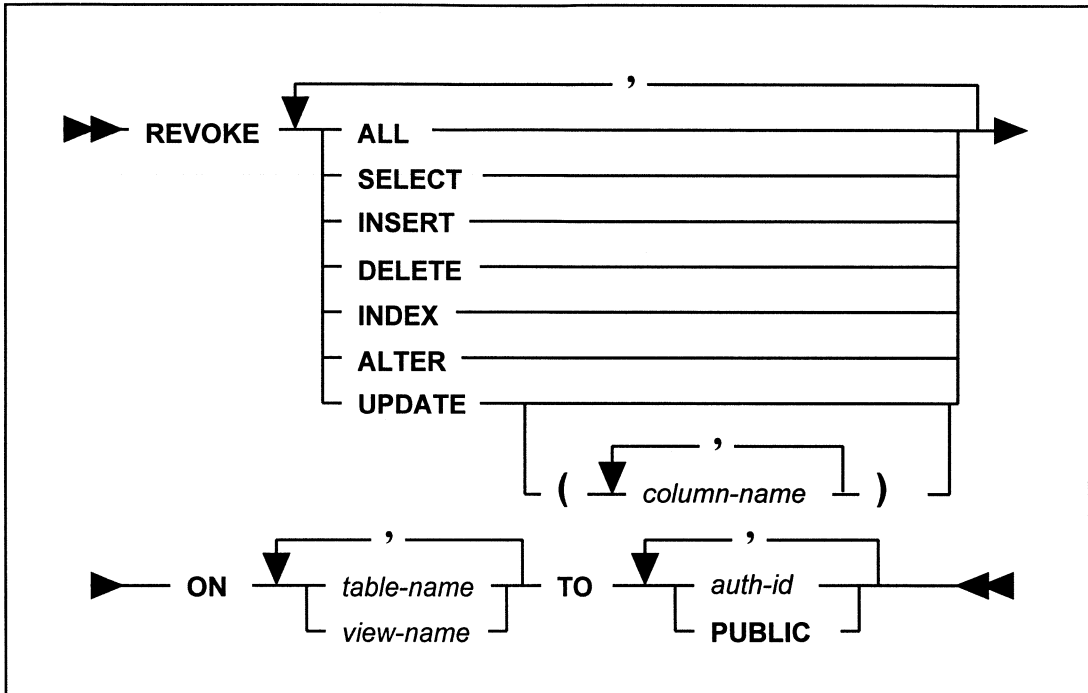
Examples

```
SQL> REVOKE CONNECT
FROM JOE, JOHN;
```

```
SQL> REVOKE RESOURCE
FROM LINDA;
```

See Also

GRANT (Database Authority)
GRANT (Table Privileges)
REVOKE (Table Privileges)

REVOKE (Table Privileges)**Description**

This form of the REVOKE command revokes privileges previously granted to users for a table or view.

Any user with the appropriate GRANT (Table Privileges) authority for a table can revoke the privileges for the corresponding tables or views. The creator of a table can revoke privileges on it.

The following privileges can be revoked.

Privilege	Description
SELECT	Select data from a table or view.
INSERT	Insert rows into a table or view.
DELETE	Delete rows from a table or view.
UPDATE	Update a table and (optionally) update only the specified columns.
INDEX	Create or drop indexes for a table.
ALTER	Alter a table.
ALL	All of the above for a table.

table-name

Table names (including any implicit qualifier) must identify a table that exists in the database.

view-name

View names (including any implicit qualifier) must identify a view that exists in the database.

column-name

If you specify more than one table or view, and UPDATE privileges are revoked for selected columns, then each column named must be in the specified tables or views.

authorization-id

The authorization-id must refer to a valid user who currently has the privileges that are being revoked.

PUBLIC

This keyword signifies all users. By revoking a privilege from PUBLIC, it means that all current users have the specified privilege revoked.

Examples

Prevent Linda from reading the CHECKS table or updating the columns AMOUNT and PAYDATE.

```
SQL> REVOKE SELECT, UPDATE (AMOUNT, PAYDATE)
ON CHECKS
FROM LINDA;
```

Revoke all privileges on CHECKS and INVOICES from JOE.

```
SQL> REVOKE ALL
ON CHECKS, INVOICES
FROM JOE;
```

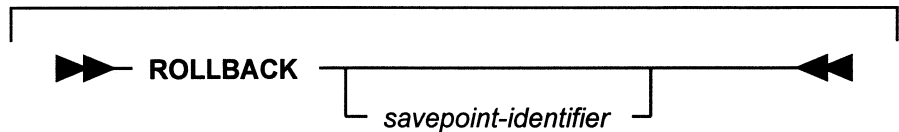
Prevent users from reading the system catalog table SYSTABLES.

```
SQL> REVOKE SELECT
ON SYSADM.SYSTABLES
FROM PUBLIC;
```

See Also

GRANT (Database Authority)
GRANT (Table Privileges)
REVOKE (Database Authority)

ROLLBACK



Description

This command ends the current transaction (logical unit of work). A transaction contains one or more SQL commands that must either all be committed or none at all.

The ROLLBACK command causes the current transaction to be aborted. This restores the database to the state it was in at the last COMMIT or ROLLBACK, or if none has been previously given, since the user connected to the database. The rollback applies to the work done for all cursors that the SQLTalk session or the application has connected to the database.

A ROLLBACK applies to all SQL commands including data definition (CREATE, DROP, ALTER) and data manipulation commands (GRANT, REVOKE, UPDATE, INSERT).

If you have CONNECT authority, you can execute the ROLLBACK command.

savepoint-identifier

If you specify the savepoint identifier, the transaction is rolled back to that savepoint. A savepoint is marked within a transaction by the SAVEPOINT command.

If the specified savepoint does not exist, the entire transaction is rolled back and an error is returned.

If you use the same savepoint-identifier again, a ROLLBACK to that savepoint-identifier will cause a rollback to the later savepoint.

Rolling back to a savepoint does *not* release locks. Rolling back without specifying a savepoint *does* release locks.

Example

```
SQL> COMMIT (signals end of transaction and start of new one)
```

```
SQL Command ...
```

```
SQL Command ...
```


```
SQL Command ...
```

```
SQL> ROLLBACK (undoes previous three SQL commands)
```

See Also

COMMIT
SAVEPOINT

SAVEPOINT



SAVEPOINT *savepoint-identifier*

Description

The SAVEPOINT command assigns a savepoint within the current transaction.

The ROLLBACK command can optionally specify a savepoint identifier. If an identifier is specified, the transaction is rolled back to that savepoint. If the specified savepoint does not exist, the entire transaction is rolled back and an error is returned.

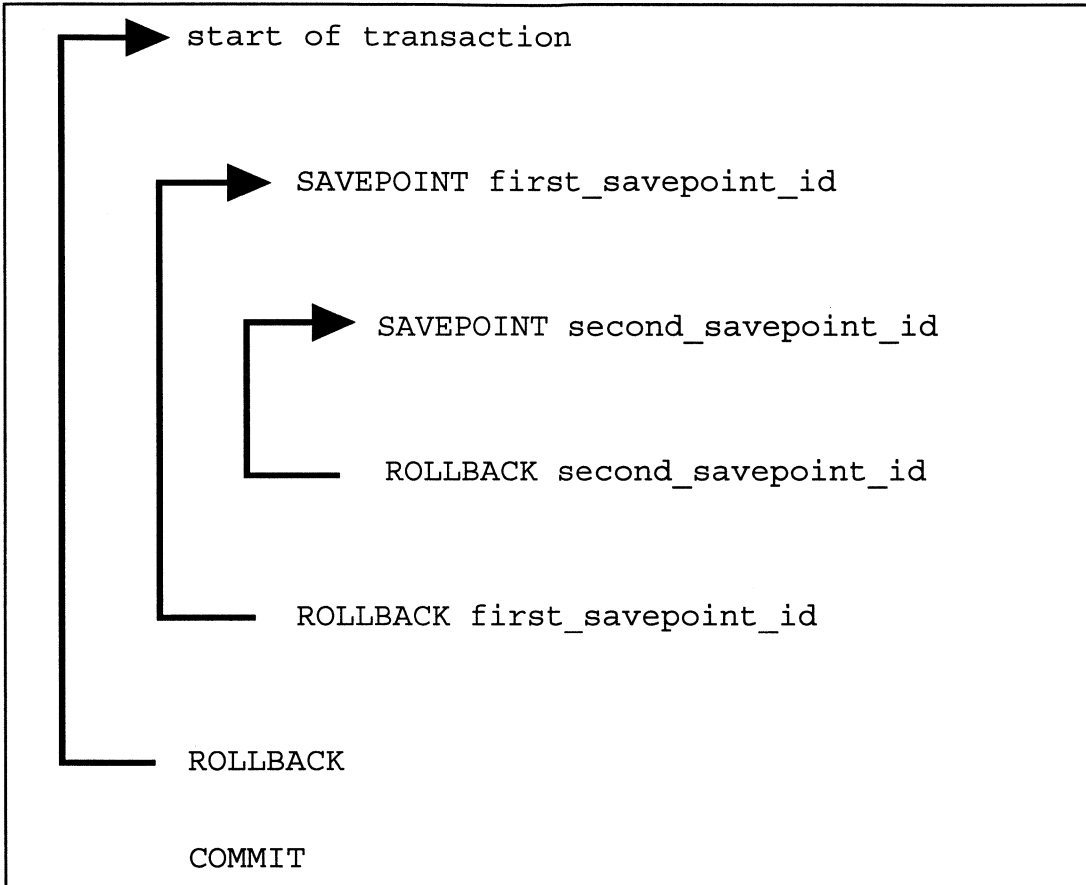
The diagram on the next page illustrates the use of the SAVEPOINT command.

Rolling back to a savepoint does *not* release locks. Rolling back without specifying a savepoint *does* release locks.

savepoint-identifier

The savepoint is identified by a long identifier that can be up to 18 characters in length.

If the same savepoint-identifier is specified twice in SAVEPOINT commands within the same transaction, the transaction will rollback to the location of the most-recent savepoint when the ROLLBACK command is given (the first savepoint is forgotten).



Example

This example shows the COMMIT, ROLLBACK, and SAVEPOINT commands used in a C program.

```
/* Example of savepoint use */

/* Start of application is an implicit begin transaction */

for (;;)
{
    /*-----
    Process 1st screen
    -----*/

    /*-----
    If non-fatal error encountered processing 1st screen,
    rollback work done so far and reprocess 1st screen.
    -----*/
    if non_fatal_error
    {
        sqlcex(cur, "ROLLBACK", 0);
        continue;
    }
}

/*-----
1st screen successfully processed. Set SAVEPOINT so we
don't have to reprocess 1st screen if subsequent errors
are encountered.
-----*/
sqlcex(cur, "SAVEPOINT screen1", 0);

for (;;)
{
    /*-----
    Process 2nd screen
    -----*/

    /*-----
    If non-fatal error encountered processing 2nd screen,
```

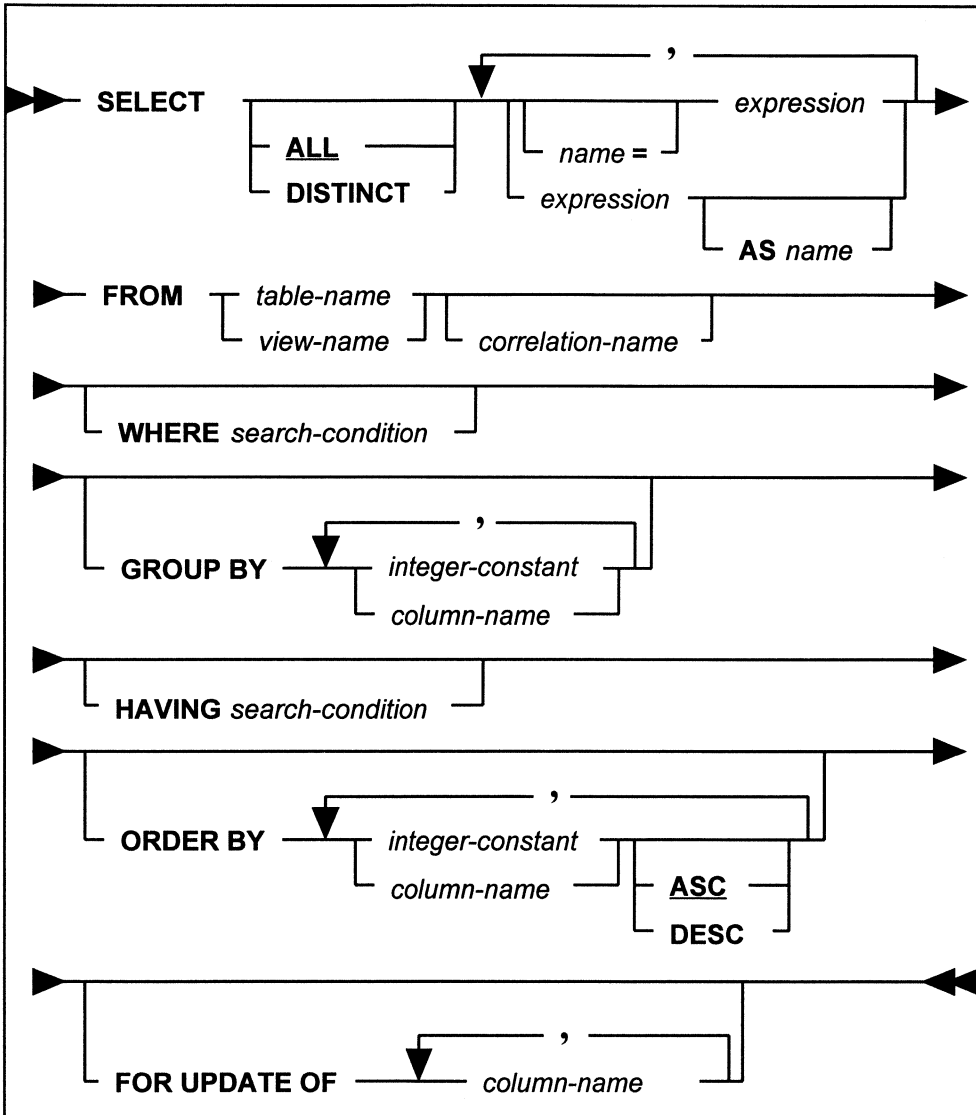
```
rollback work done so far for 2nd screen and
reprocess 2nd screen.
-----*/
if non_fatal_error
{
  sqlcex(cur, "ROLLBACK screen1", 0);
  continue;
}
}

sqlcex(cur, "COMMIT");
```

See Also

COMMIT
ROLLBACK

SELECT



Description

This command finds, retrieves, and displays data.

This command specifies:

- The tables or views in the database which are searched to find the data.
- The conditions for the search.
- The sequence in which the data is output.

SELECT commands are recursive; they can be nested within the main SELECT clause. A nested SELECT command is called a subquery.

The result of a SELECT is a set of rows called a result table which meets the conditions specified in the SELECT command.

You must have SELECT privileges on the tables and views to execute this command.

ALL

The default for a SELECT is to retrieve ALL rows.

DISTINCT

This suppresses duplicate rows.

You cannot use the DISTINCT keyword to SELECT LONG VARCHAR data types.

You cannot use a DISTINCT keyword while in restriction mode.

You cannot perform an operation with the CURRENT OF clause on a result set that you formed with the DISTINCT keyword.

expression

This is a select list that contains expressions that are separated by commas. An expression can be:

- A column name.
- A constant.
- A bind variable.
- The result of a function.
- A system keyword.

See the section called *Expressions* in chapter 6.

A select list is usually a list of columns from one or more tables.

An asterisk (*) is a wildcard search operator that represents the entire set of columns in the tables or views specified in the FROM clause. You can also specify all the columns in a single table if "*" is qualified with the desired table name. For example, the command

```
SQL> SELECT TAB1.*, COL1 FROM TAB1, TAB2;
```

Returns all of the columns in table TAB1 and the single column COL1 from table TAB2.

Each column name in the select list must unambiguously identify a column in one of the tables or views named in the FROM clause. If a result set is derived from a select list of columns from more than one table or view, any column name in the list which is the same in two tables must be qualified by the table name to make it a unique name.

```
SQL> SELECT CUSTOMER.CUSTNO, ORDERNO  
FROM CUSTOMER, ORDERS  
WHERE CUSTOMER.CUSTNO = ORDERS.CUSTNO;
```

In the above example, the name CUSTNO appears in the CUSTOMER table and the ORDERS table. It therefore must be qualified to make it unambiguous within the SQL command.

The select list can only contain aggregate functions when the GROUP BY clause is used, or when the select list consists entirely of aggregate functions.

name = *expression*
expression AS name

Both of these formats do the same thing: assign a *name* that is used as a column heading in the output. For example:

```
SQL> SELECT Customer Number=CUSTNO  
FROM CUSTOMER;
```

FROM

The FROM clause contains the names of the tables or views from which the set of resulting rows are formed. Each name must identify a table or view that exists in the database.

A *correlation name* can be assigned for the table or view immediately preceding the name. Each correlation name in a FROM clause must be unique.

Correlation names are required when a search condition is executed more than once for the same table or view in a single SQL command (as in joining a table to itself or in correlated subqueries, described below). They provide a shorthand way to qualify column names.

The above SQL command can be written using the correlation name C to designate CUSTOMER and O to designate ORDERS:

```
SQL> SELECT C.CUSTNO, ORDERNO  
FROM CUSTOMER C, ORDERS O  
WHERE C.CUSTNO = O.CUSTNO;
```

WHERE *search-condition*

The WHERE clause specifies a *search condition* for the base tables or views.

The search condition of the WHERE clause cannot contain any aggregate functions (unless part of a subselect). See the section called *Search Conditions* in chapter 6.

You cannot use a LONG VARCHAR column in a subselect search condition.

GROUP BY

The GROUP BY clause groups the result rows of the query in sets according to the columns named in the clause. If the column by which a grouping occurs is an expression (but not an aggregate function), you must specify a number that indicates its relative position in the select list. Aggregate functions, since they yield one value, cannot be grouping columns.

The result of a grouping is the set of rows for which all values of the grouping column are equal. NULL values in a grouping column are treated as a separate group.

If a GROUP BY clause is specified, each column in the select list must be listed in the GROUP BY clause or each column in the select list must be used in an aggregate set function that yields a single value.

The following example finds the total salary budget for each department, the average salary, the number of people in each department, the average number of years of experience. It illustrates a GROUP BY and an equijoin (for getting the department name).

```
SQL> SELECT EMP.DEPTNO, DEPTNAME, SUM(SALARY),
AVG(SALARY), COUNT(EMPNO), AVG(YRS_EXP)
FROM EMP, DEPT
WHERE EMP.DEPTNO = DEPT.DEPTNO
GROUP BY EMP.DEPTNO, DEPTNAME;
```

You cannot use GROUP BY while in restriction mode.

You cannot perform an operation with the CURRENT OF clause on a result set that you formed with a GROUP BY clause.

HAVING search-condition

The HAVING clause allows a search condition for a group of rows resulting from a GROUP BY or grouping columns. If a grouping column is an expression that is *not* an aggregate function (such as SAL*10), it cannot be used in the HAVING clause.

Using the example for the GROUP BY clause, we are only interested in the departments where the average salary is greater than 4000.

```
SQL> SELECT EMP.DEPTNO, DEPTNAME, SUM(SALARY),
AVG(SALARY), COUNT(EMPNO), AVG(YRS_EXP)
FROM EMP, DEPT
WHERE EMP.DEPTNO = DEPT.DEPTNO
GROUP BY EMP.DEPTNO, DEPTNAME
HAVING AVG(SALARY) > 4000;
```

You cannot use a HAVING clause while in restriction mode.

ORDER BY

This specifies the ordering, or sorting, of rows in a result table. Rows can be sorted on more than one column. The major sort is on the first column specified in the ORDER BY clause and the minor sorts are on the columns specified after that.

If the sort is on a column derived from a function or arithmetic expression, the column must be specified by an integer that signifies its relative number in the select list of the command.

Each column name (or number) can be optionally followed by ASC or DESC for ascending or descending sort sequence. ASC is the default order.

You cannot use the ORDER BY clause in a SELECT command that is a component of a UNION of SELECT commands. You cannot use the ORDER BY clause in a view definition.

You cannot perform an operation with the CURRENT OF clause on a result set that you formed with an ORDER BY clause.

You cannot use an ORDER BY clause while in restriction mode.

You cannot use an ORDER BY clause in a subselect.

You cannot use string functions in an ORDER BY clause. Instead, specify the string function in the select list and then use the select list column number in the ORDER BY clause.

FOR UPDATE OF

This locks parts of a table so that a subsequent UPDATE or DELETE will not cause a deadlock between concurrent users. This clause is compatible with DB2.

You can UPDATE columns in the column-name list. Those columns must be a part of the table or view named in the FROM clause of the SELECT command.

When you use the FOR UPDATE OF clause, SQLBase uses update locks. An update lock reduces the possibility of deadlocks. Update locks are compatible with other update locks and shared locks, but not with exclusive locks. An update lock is released if the transaction does not continue with an UPDATE or DELETE. This is in contrast to exclusive locks which are held until a COMMIT or ROLLBACK.

For the read repeatability (RR) and cursor stability (CS) isolation levels, the FOR UPDATE OF clause uses update locks. The FOR UPDATE OF clause has no effect on read only (RO) or release lock (RL) isolation levels.

You can use the CURRENT OF clause in an UPDATE or DELETE command on a result set formed with the FOR UPDATE OF clause.

You cannot use the FOR UPDATE OF clause with the UNION or ORDER BY clauses.

Examples

Select all rows from the CUSTOMER table.

```
SQL> SELECT * FROM CUSTOMERS;
```

Make a list of the job titles.

```
SQL> SELECT DISTINCT JOB FROM EMP;
```

Display the name and annual salary of people whose monthly salary is greater than \$5000.

```
SQL> SELECT NAME, SALARY*12  
FROM EMP  
WHERE SALARY > 5000;
```

Find the minimum and average salary for each department.

```
SQL> SELECT DEPTNO, MIN(SALARY), AVG(SALARY)  
FROM EMP  
GROUP BY DEPTNO;
```

Find the total sales for each quarter. This command illustrates the use of an integer when using a function in a GROUP BY clause.

```
SQL> SELECT @QUARTER(SALES_DATE), SUM(SALES)
FROM SALES
GROUP BY 1;
```

Get the employee information for people with the same job as WONG.

```
SQL> SELECT * FROM EMP
WHERE JOB IN
(SELECT JOB FROM EMP WHERE NAME = 'WONG');
```

Find the orders where the price paid was equal to the list price.

```
SQL> SELECT * FROM ORDERS X
WHERE PRICE =
(SELECT LISTPRICE FROM PARTS
WHERE PARTS.PNUM = X.PNUM);
```

Find a customer invoice so that you can update it.

```
SQL> SELECT * FROM INVOICE
WHERE CUST='GBS COMPUTERS'
ORDER BY INVDATE;
```

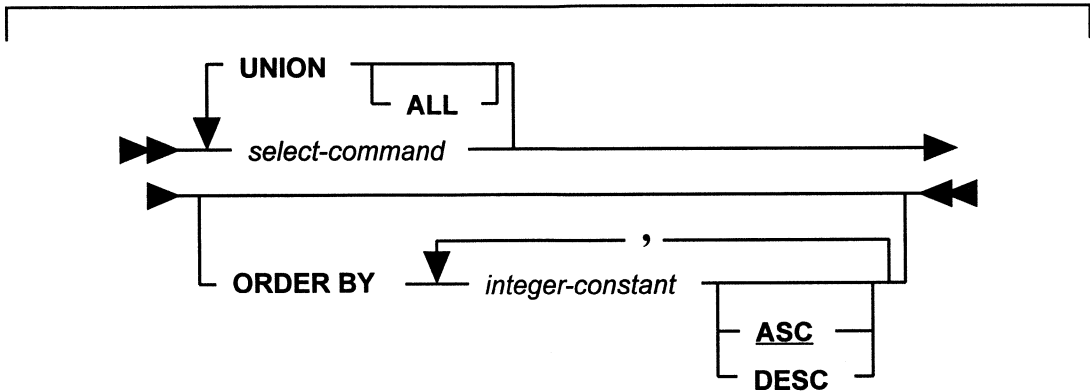
Update the PRESIDENTS database to show a candidate in the 1960 election.

```
SQL> SELECT candidate
FROM election
WHERE election_year = 1960
FOR UPDATE OF winner_loser_indic;
```

```
UPDATE election
SET winner_loser_indic = '?'
WHERE CURRENT OF selcur;
```

```
FETCH 1;
```

```
CANDIDATE
=====
Kennedy J
```


UNION Clause

This clause merges the result of two or more SELECT commands. Any duplicate rows are eliminated.

Each result table must have the same number of columns. None of the columns can be LONG VARCHAR columns. Except for column names, the description of the corresponding column in each table must be identical.

You cannot use UNIONS in a view or in restriction mode.

You cannot perform an operation with the CURRENT OF clause on a result set that you formed with a UNION clause.

ALL

If this is specified, duplicate rows will *not* be eliminated. The result contains all the rows selected. If ALL is used, it must be repeated for every SELECT command:

```
SQL> select-cmd-1 UNION ALL select-cmd2, ....
```

UNION ALL *select-cmd-n.*

ORDER BY

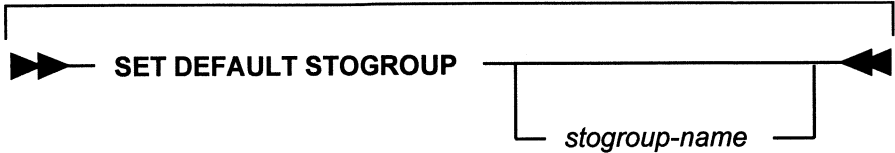
An ORDER BY clause sorts the final result set of rows from the UNION of two (or more) tables. When an ORDER BY clause is used with a UNION, you must use an integer specifying the sequence number of the column in the select list.

Example

This command finds the employees from department 10 and those whose project number is 20.

```
SQL> SELECT EMPNO FROM EMP WHERE DEPTNO = 10  
UNION  
SELECT EMPNO FROM PROJ WHERE PROJNO = 20;
```

SET DEFAULT STOGROUP



Description

This command sets the default storage group. After a default name is given to a storage group, all subsequent CREATE DATABASE commands will cause databases to be partitioned.

stogroup-name

The name of the specified storage group. The storage group name is optional. If you omit the storage group name, the storage group is null. This allows databases to be created in the normal file system (non-partitioned).

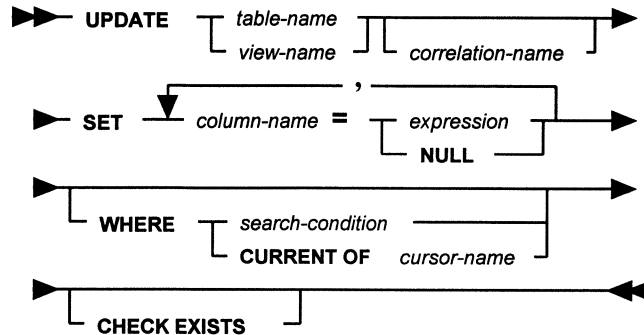
Example

```
SQL> SET DEFAULT STOGROUP ACCTLIST;
```

See Also

```
ALTER STOGROUP
CREATE STOGROUP
DROP STOGROUP
```

UPDATE



Description

This command updates the value of one or more columns of a table or view based on the specified search conditions. You must possess the UPDATE privilege on the columns of the table or view.

The UPDATE command (for referential integrity) updates tables with primary or foreign keys. Any non-null foreign key values that you enter must match the primary key for each relationship in which the table is a dependent.

If you are updating a parent table, you cannot modify a primary key for which dependent rows exist. (This would violate referential constraints for dependent tables and would leave a row without a parent.) In addition, you cannot give a primary key a null value.

In a database with referential integrity, the only UPDATE rule that can be applied to a parent table is RESTRICT. This means that any attempt to update the primary key of the parent table is restricted to cases where there are no matching values in the dependent table.

If an UPDATE against a table with a referential constraint fails, an error message is returned.

table-name

This identifies an existing table.

System catalog tables can be named, but only-user defined columns can be updated.

view-name

This identifies an existing view.

You cannot UPDATE a view based on more than one table.

correlation-name

The correlation name must be specified if the search condition involves a correlated subquery.

column-name

This identifies the columns to be updated in the table or view.

Columns derived from an arithmetic expression or a function cannot be updated.

If a view was specified with WITH CHECK OPTION, the updated row must conform to the view definition.

SET

An expression whose value is to be used in updating the column must not contain any functions.

If the update value is specified as NULL, the column must have been defined to accept null values.

If a unique index is specified on a column, the update column value must be unique or an error results. Note that for a multi-column index, it is the *aggregate* value of the index that must be unique.

WHERE *search-condition*

The WHERE clause specifies the rows to be updated based on a search condition.

When this clause is used, it is called a “searched UPDATE.”

WHERE CURRENT OF *cursor-name*

The causes the row at which a cursor is currently positioned to be updated according to the specification of the SET clause.

When this clause is used, it is called a “positioned UPDATE” or a “cursor-controlled UPDATE.”

This type of update requires two open cursors:

- Cursor 1 is associated with a SELECT command. The current row references the row of the most recent fetch.
- Cursor 2 is associated with the UPDATE command.

A cursor-name must be associated with cursor 1 before this command can be executed.

You can only use CURRENT OF if all of the following are true for the corresponding SELECT command:

- The cursor must be named or be in result set mode.
- The SELECT command cannot contain joins, GROUP BY, DISTINCT, SET functions, UNION, or ORDER BY.
- Any subselect in the SELECT command must satisfy the previous condition.

CHECK EXISTS

This clause specifies to return an error if at least one row is *not* updated. This clause can be used in any context, including in chained commands.

Examples

Change Jones' salary.

```
SQL> UPDATE EMP SET SAL = 5000 WHERE NAME = 'JONES';
```

Give all employees on project SPECIAL a 10% raise.

```
SQL> UPDATE EMP SET SAL = SAL*1.10
WHERE PROJECT = 'SPECIAL';
```

Prefix all bug numbers with the letter P. Update every row in the table.

```
SQL> UPDATE BUG SET BUGNUM = 'P' || BUGNUM;
```

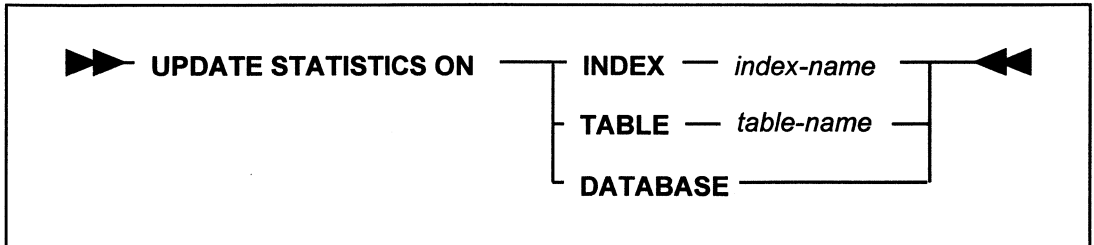
Update the row reference by the current fetch of cursor named FINDBUG.

```
SQL> UPDATE BUG SET PRIO = 4 WHERE CURRENT OF FINDBUG;
```

See Also

```
CREATE TABLE
SELECT
SET CURSORNAME
```

UPDATE STATISTICS



Description

This command updates the statistics for an index.

At `CREATE INDEX` time, statistics are gathered about the *selectivity factors* of a particular index. For example, if there are four possible values for column A, then the selectivity factor for column A is .25. Also, selectivity factors for correlations between multiple columns within a concatenated index are gathered.

The optimizer uses the selectivity factor to weigh each access path to a table and tries to choose the best access path based on cost computation.

You should execute this command when there have been significant changes to an index that have changed the selectivity factors.

Stored commands should be re-stored in order to take advantage of the updated statistics.

Turn on the `TIME` option to check the performance before and after the statistics have been updated.

INDEX *index-name*

Updates the statistics for the specified index.

TABLE *table-name*

Updates the statistics for all indexes in the specified table.

DATABASE

Updates the statistics for all indexes in the database.

Example

```
SQL> UPDATE STATISTICS ON INDEX CUSTOMER_ID;
```

See Also

```
CREATE INDEX  
SET TIME
```


Chapter 8

SQL Function Reference

About This Chapter

SQLBase has a set of functions for manipulating strings, dates and numbers. Each function is described in this chapter.

A function returns a value that is derived by applying the function to its arguments.

Functions are classified as:

- Aggregate functions.
- String functions.
- Date and time functions.
- Logical functions.
- Special functions.
- Math functions.
- Finance functions.

SQLBase provides DB2-compatible functions, and other functions. Functions which are extensions of DB2 and are *not* compatible with DB2 are prefixed with an "at sign" (@).

Data Type Conversions in Functions

In most cases, functions accept any data type as an argument if the value conforms to the operation that function performs. SQLBase will automatically convert the value to the required data type.

For example, in functions that perform arithmetic operations, arguments can be character data types if the value forms a valid numeric value (only digits and standard numeric editing characters).

For date/time functions, an argument can be a character or numeric data type if the value forms a valid date/time value.

Aggregate Functions

An aggregate function computes one summary value from a group of values.

Aggregate functions can be applied to the data values of an entire table or to a subset of the rows in a table.

They may be nested up to two levels deep.

The data type of the argument may be numeric, date/time, or character. If an argument is a character data type, the value must form a valid numeric or date/time value (only digits and standard editing characters). SQLBase will automatically convert the value to the required data type.

You cannot use aggregate functions while in restriction mode.

The aggregate functions are as follows:

<p>AVG COUNT MAX @MEDIAN MIN SUM @SDV</p>
--

String Functions

String functions return information about character data types.

The output of a string function is always a string or a number. Some functions yield TRUE or FALSE. TRUE is expressed as the number 1 and FALSE is expressed as 0.

String functions may be nested within one another, so that the output of the inner function is used as an argument to the outer function.

**@CHAR
@CODE
@DECODE
@EXACT
@FIND
@LEFT
@LENGTH
@LOWER
@MID
@NULLVALUE
@PROPER
@REPEAT
@REPLACE
@RIGHT
@SCAN
@STRING
@SUBSTRING
@TRIM
@UPPER
@VALUE**

String functions cannot be used in an ORDER BY clause of the SELECT command. Instead, specify the string function in the select list and then use the select list column number in the ORDER BY clause.

Date/Time Functions

These functions return information about date/time data values or return a date/time result. The date/time functions are as follows:

```
@DATE  
@DATETOCHAR  
@DATEVALUE  
@DAY  
@HOUR  
@MICROSECOND  
@MINUTE  
@MONTH  
@MONTHBEG  
@NOW  
@QUARTER  
@QUARTERBEG  
@SECOND  
@TIME  
@TIMEVALUE  
@WEEKBEG  
@WEEKDAY  
@YEAR  
@YEARBEG  
@YEARNO
```

For date/time functions, an argument can be a character or numeric data type if the value forms a valid date/time value.

When a portion of the input date/time string is missing, SQLBase supplies the default of 0, which converts to December 30, 1899 12:00:00 AM. Functions behaving this way are @DATE, @DATEVALUE, @NOW, @TIME and @TIMEVALUE.

Math Functions

These functions take single numeric values as arguments and return numeric results.

The mathematical functions are similar to Microsoft C Library math functions.

Trigonometric functions are based on radians instead of degrees.

Arguments can be character data types if the value forms a valid numeric value (only digits and standard numeric editing characters). SQLBase will automatically convert the value to the required data type.

```
@ABS  
@ACOS  
@ASIN  
@ATAN  
@ATAN2  
@COS  
@EXP  
@FACTORIAL  
@INT  
@LN  
@LOG  
@MOD  
@PI  
@ROUND  
@SIN  
@SQRT  
@TAN
```

Finance Functions

The finance functions are similar to Microsoft C Library math functions.

Arguments can be character data types if the value forms a valid numeric value (only digits and standard numeric editing characters). SQLBase will automatically convert the value to the required data type.

```
@CTERM
@FV
@PMT
@PV
@RATE
@SLN
@SYD
@TERM
```

Logical Functions

Logical functions return a value based on a condition. The result of these functions is always 1 or 0 (TRUE = 1, FALSE = 0).

```
@IF
@ISNA
```

Special Functions

These functions provide special capabilities.

**@CHOOSE
@DECIMAL
@DECRYPT
@DECODE
@HEX
@LICS**

SQLBase Function Summary

Function Name	Description
AVG	Average of items.
COUNT	Count of items.
MAX	Maximum of items.
MIN	Minimum of items.
SUM	Sum of items.
@ABS	Absolute value.
@ACOS	Arc-cosine.
@ASIN	Arc-sine.
@ATAN	Two-quadrant arc-tangent.
@ATAN2	Four-quadrant arc-tangent.
@CHAR	ASCII character for a decimal code.
@CHOOSE	Select a value from a list based on a correlation.
@CODE	ASCII decimal code of the first character in a string.
@COS	Cosine.
@CTERM	Compounding periods to earn a future value.
@DATE	Convert to a date.
@DATETOCHAR	Edit a date value.
@DATEVALUE	Edit a date value.
@DAY	Day of the month.
@DECIMAL	Decimal value of a hexadecimal string.
@DECODE	Returns a string, given an expression.

Function Name	Description
@EXACT	Compare two strings.
@EXP	Natural logarithmic base (e) raised to the x power.
@FACTORIAL	Factorial.
@FIND	Position within string1 that occurs in string2.
@FV	Future value of a series of equal payments.
@HEX	Hexadecimal string of a decimal number.
@HOUR	Hour of the day.
@IF	Test number and return 1 if TRUE or 2 if FALSE.
@INT	Integer portion.
@ISNA	Return TRUE if NULL.
@LEFT	Left-most substring.
@LENGTH	Length of a string.
@LICS	Sort using international character set.
@LN	Natural logarithm (base e) of (positive) x.
@LOG	Positive base-10 logarithm of x.
@LOWER	Upper-case to lower-case.
@MEDIAN	Middle value in a set of items.
@MICROSECOND	Microsecond value.
@MID	Return string, starting with character at start-pos.
@MINUTE	Minute of the hour.
@MOD	Modulo (remainder) of x/y.
@MONTH	Month of the year.

Function Name	Description
@MONTHBEG	First day of the month.
@NOW	Current date and time.
@NULLVALUE	Return a string or number specified by y if x is NULL.
@PI	Value Pi ($\pi = 3.14159265$).
@PMT	Periodic payments needed to pay off loan principal.
@PROPER	Convert first character of each word in a string to uppercase and make other characters lowercase.
@PV	Present value of a series of equal payments.
@QUARTER	Number that represents the quarter.
@QUARTERBEG	First day of the quarter.
@RATE	Interest rate for an investment to grow to a future value.
@REPEAT	Concatenates a string with itself for the specified number of times.
@REPLACE	Replace characters in a string.
@RIGHT	Rightmost substring.
@ROUND	Round a number.
@SCAN	Search a string for a pattern.
@SDV	Standard deviation.
@SECOND	Second of the minute.
@SIN	Sine.
@SLN	Straight-line depreciation.

Function Name	Description
@SQRT	Square root.
@STRING	Convert a number to a string.
@SUBSTRING	Return a portion of a string.
@SYD	Sum-of-the-Years'-Digits depreciation.
@TAN	Tangent.
@TERM	Number of payment periods for an investment.
@TIME	Return a date/time value given the hour, minute, and second.
@TIMEVALUE	Return a date/time value, given HH:MM:SS [AM or PM].
@TRIM	Strip leading and trailing blanks; compress multiple spaces.
@UPPER	Lower-case to upper-case.
@VALUE	Convert a character string with digits to a number.
@WEEKBEG	Monday of the week.
@WEEKDAY	Day of the week.
@YEAR	The year relative to 1900.
@YEARBEG	First day of the year.
@YEARNO	Calendar year.

AVG

The diagram shows the syntax for the AVG function. It consists of the keyword **AVG** followed by an opening parenthesis **(**. Inside the parentheses, there is a bracketed section containing the keywords **ALL** and **DISTINCT**, with a vertical line between them. To the right of this bracketed section is the text *expression*, followed by a hyphen **-** and a closing parenthesis **)**. The entire function syntax is enclosed in a rectangular box with double arrowheads pointing outwards from the left and right sides.

This function returns the average of the values in the argument.

The data type of the argument may be numeric, date/time, or character. If an argument is a character data type, the value must form a valid numeric or date/time value (only digits and standard editing characters). SQLBase will automatically convert the value to the required data type.

The data type of the result is numeric.

The keyword **DISTINCT** eliminates duplicates. If **DISTINCT** is not specified, then duplicates are not eliminated.

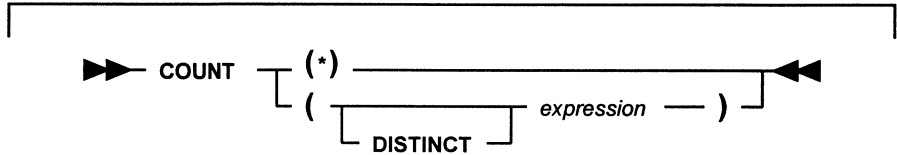
Null values are ignored.

Example

This example finds the total salary budget for each department, the average salary, the number of people in each department, the average number of years of experience.

```
SQL> SELECT EMP.DEPTNO, DEPTNAME, SUM(SALARY),
AVG(SALARY), COUNT(EMPNO), AVG(YRS_EXP)
FROM EMP, DEPT
WHERE EMP.DEPTNO = DEPT.DEPTNO
GROUP BY EMP.DEPTNO, DEPTNAME;
```

COUNT



This function returns a count of items.

COUNT(*) always returns the number of rows in the table. Rows that contain null values are included in the count.

COUNT(column-name) returns the number of column values.

COUNT(DISTINCT column-name) filters out duplicate column values.

LONG VARCHARs can be counted.

The keyword DISTINCT eliminates duplicates. If DISTINCT is not specified, then duplicates are not eliminated.

Example

How many rows are in each department of the EMP table?

```
SQL> SELECT DEPTNO, COUNT(*) FROM EMP
GROUP BY DEPTNO;
```


MAX


Diagram illustrating the syntax of the MAX function: `MAX ([ALL | DISTINCT] expression)`. The words `ALL` and `DISTINCT` are enclosed in a bracket, indicating they are optional modifiers for the `expression`.

This function returns the maximum value in the argument, which is a set of column values.

The data type of the argument may be numeric, date/time, or character. If an argument is a character data type, the value must form a valid numeric or date/time value (only digits and standard editing characters). SQLBase will automatically convert the value to the required data type.

The data type of the result is the same as the input argument.

The keyword `DISTINCT` eliminates duplicates. If `DISTINCT` is not specified, then duplicates are not eliminated.

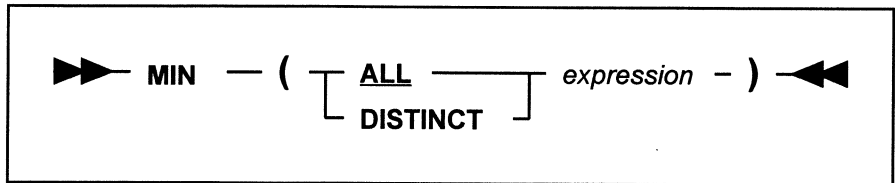
Null values are ignored.

Example

This example finds the highest and the lowest salary in department 10.

```
SQL> SELECT MAX(SALARY), MIN(SALARY)
FROM EMP WHERE DEPTNO = 10;
```

MIN



This function returns the minimum value in the argument, which is a set of column values.

The data type of the argument may be numeric, date/time, or character. If an argument is a character data type, the value must form a valid numeric or date/time value (only digits and standard editing characters). SQLBase will automatically convert the value to the required data type.

The data type of the result is the same as the input argument.

The keyword DISTINCT eliminates duplicates. If DISTINCT is not specified, then duplicates are not eliminated.

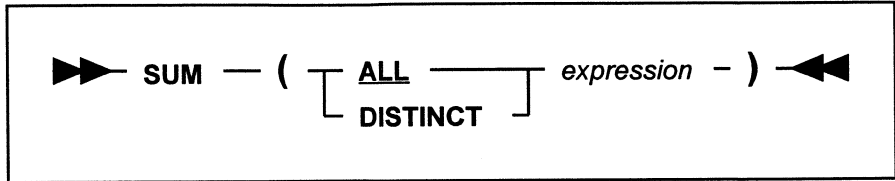
Null values are ignored.

Example

This example finds the highest and the lowest salary in department 10.

```
SQL> SELECT MAX(SALARY), MIN(SALARY)
FROM EMP WHERE DEPTNO = 10;
```

SUM



This function returns the sum of the values in the argument, which is a set of column values.

The data type of the argument may be numeric, date/time, or character. If an argument is a character data type, the value must form a valid numeric or date/time value (only digits and standard editing characters). SQLBase will automatically convert the value to the required data type.

The data type of the result is the same as the input argument.

The keyword **DISTINCT** eliminates duplicates. If **DISTINCT** is not specified, then duplicates are not eliminated.

Null values are ignored.

Example

Calculate the total orders.

```
SQL> SELECT SUM (PRICE * QUANTITY) FROM ORDERS;
```

This example totals the sales for each quarter and displays the first day of the quarter.

```
SQL> SELECT @QUARTERBEG(SALES_DATE), SUM(SALES)
FROM SALES_TABLE GROUP BY 1;
```

@ABS

@ABS (x)

This function returns the absolute value of x.

Example

```
@ABS (-1.1)
```

Returns 1.1.

```
SQL> SELECT @ABS (PRICE) FROM FINANCES;
```

Returns all price column entries as positive (absolute) values.

@ACOS

@ACOS (x)

This function returns the arc-cosine of x.

Example

```
@ACOS (.1)
```

Returns 1.47062891.

```
SQL> SELECT @ACOS (PVAL) FROM GEOM;
```

Returns the arc-cosine of all PVAL column entries in the GEOM table.

@ASIN

@ASIN (x)

This function returns the arc-sine of x .

Example

```
@ASIN (.1)
```

Returns .100167421.

```
SQL> SELECT @ASIN(PVAL) FROM GEOM;
```

Returns the arc-sine of all PVAL column entries in the GEOM table.

@ATAN

@ATAN (x)

This function returns the arc-tangent of x .

Example

```
@ATAN (.1)
```

Returns .099668652.

```
SQL> SELECT @ATAN(PVAL) FROM GEOM;
```

Returns the arc-tangent of all PVAL column entries in the GEOM table.

@ATAN2

@ATAN2 (*x,y*)

This function returns the arc-tangent of y/x .

The order of arguments is the opposite of C [atan2(y,x)].

Example

```
@ATAN2 (.1, .1)
```

Returns .785398163.

```
SQL> SELECT @ATAN2 (PVAL,QVAL) FROM GEOM;
```

Returns the arc-tangent of all QVAL and PVAL column entries in the GEOM table.

@CHAR

@CHAR (*number*)

This function returns the ASCII character for a decimal code. If the argument is outside the ASCII character set range, results depend on the display character set.

Example

```
@CHAR (65)
```

Returns the letter 'A'.

@CHOOSE

@CHOOSE (*selector-number, value 0, value 1, ..., value n*)

This function selects a value from a list based on a correlation between the selector-number and the sequence number of a value in the list.

You must specify a selector-number and at least one value. A negative selector-number maps to the first value in the list (value 0). If the selector number exceeds the number of values in the list, the result is the last value in the list. Every value in the list is cast to the data type of the first value (value 0).

Example

```
@CHOOSE (SEL_NUM, A, B, C, D, E, F, G)
```

Selector-number	Values
0	A
-1	A
2	C
12	G

This example finds the day of the week on which each employee was born.

```
SQL> SELECT @CHOOSE(@WEEKDAY(BIRTH_DATE), 'Sat','Sun',
'Mon', 'Tue', 'Wed', 'Thu', 'Fri'), @YEAR(BIRTH_DATE))
FROM EMP
WHERE @YEARNO(BIRTH_DATE) > 1900;
```

@CODE

@CODE (*string*)

This function returns the ASCII decimal code of the first character in a string.

Example

```
@CODE ( 'ABC' )
```

Returns the number 65, which is the code for 'A'.

@COS

@COS (*x*)

This function returns the cosine of x.

Example

```
@COS (.1)
```

Returns .995004165.

```
SQL> SELECT @COS(PVAL) FROM GEOM;
```

Returns the cosine of all PVAL column entries in the GEOM table.

@CTERM

@CTERM (*int*, *fv*, *pv*)

This function returns the number of compounding periods to an investment of present value *pv* to grow to a future value *fv*, earning a fixed periodic interest rate *int*.

@CTERM uses this formula to compute the term:

$$\frac{\ln (fv/pv)}{\ln (1+int)}$$

fv = future value
pv = present value
int = periodic interest rate
 ln = natural logarithm

Example

```
@CTERM (.10/12, 20000, 10000)
```

Returns 83.5237559, which is the number of months it will take to double a \$10,000 investment that earns a 10% annual interest rate compounded monthly.

@DATE

@DATE (*year-number*, *month-number*, *day-number*)

This function converts the arguments to a date.

The data type of the result is date/time.

Example

```
@DATE (1987, 1, 31)
```

Returns 31-JAN-1987

@DATETOCHAR

@DATETOCHAR (*date*, *picture*)

This function accepts a DATE, TIME, or DATETIME data type value (specified in *date*), applies the editing specified by *picture* and returns the edited value. For an explanation of *picture*, see the COLUMN command.

The data type of the result is character.

Example

```
SQL> SELECT @DATETOCHAR(SYSDATETIME, 'dd-mm-yy') FROM ...
```

Returns 31-01-46.

@DATEVALUE

@DATEVALUE (*date-string*)

This function converts the argument to a date.

@DATEVALUE is like @DATE, except its argument is a date string, or a portion of a date string. It converts the date string in any standard date string form (dd-mon-yyyy hh:mm:ss) to the date portion of the string.

The data type of the result is date.

Example

If a DATE column called APPT contains '18-JAN-1987 10:14:27 AM', then:

```
@DATEVALUE (APPT)
```

Returns 18-JAN-1987.

@DAY

@DAY (*date*)

This function returns a number between 1 and 31 that represents the day of the month.

Example

If BIRTHDATE contains '12/28/46', then:

```
@DAY (BIRTHDATE)
```

Returns 28.

@DECIMAL

@DECIMAL (*string*)

This function returns the decimal equivalent for the given hexadecimal number.

Example

```
@DECIMAL ('A')
```

Returns 10.

@DECODE

@DECODE (*expr*, *search1*, *return1*, *search2*, *return2*, ..., [default])

If *expr* equals any *search*, returns the following *return*; if not, returns default. If default is omitted and there is no match, NULL is returned. The *expr* may be any data type; *search* must be the same type. The value returned is forced to the same datatype as the first *return*.

Example

This returns employee's and manager's names. If no match is found, "None" is returned:

```
SQL> SELECT emp_name, @DECODE(mgr_code, 100, 'Lennon', 200,
'Starr 'None'), Mgr_code from emp_table;
```

<u>EMP_NAME</u>	<u>@DECODE(MGR_CODE, ...)</u>	<u>MGR_CODE</u>
Jagger	Lennon	100
Watts	Starr	200
Richards	None	888

@DECRYPT

@DECRYPT (password)

This function decrypts a password stored in the PASSWORD column in the SYSUSERAUTH system catalog table.

Example

This returns the decrypted passwords for a user and system administrator:

```
SQL> SELECT name, @DECRYPT("PASSWORD") from sysuserauth;
```

NAME	@DECRYPT ("PASSWORD")
=====	=====
SYSADM	SYSADM
HARRISON	HARRISON

@EXACT

@EXACT (*string1*, *string2*)

This function compares two strings.

If the strings are identical, the function returns 1; otherwise the function returns 0.

This function is case sensitive.

Example

```
@EXACT ('TRUDY', 'NOAH')
```

Returns 0.

If the column `NAME` contains the value 'TRUDY', then:

```
@EXACT ('TRUDY', NAME)
```

Returns 1.

@EXP

@EXP (x)

This function returns the natural logarithmic base (e) raised to the x power.

Example

```
@EXP (10)
```

Returns 22026.4658.

```
SQL> SELECT @EXP (PVAL) FROM GEOM;
```

Returns the e of all PVAL column entries in the GEOM table.

@FACTORIAL

@FACTORIAL (x)

This function computes the factorial of the argument. The argument must be an INTEGER (no decimal portion) and positive (≥ 0).

Example

```
@FACTORIAL (10)
```

Returns 3628800.

```
SQL> SELECT @FACTORIAL (PVAL) FROM GEOM;
```

Returns the factorial of all PVAL column entries in the GEOM table.

@*FOUND*

@*FOUND* (*string1*, *string2*, *start-pos*)

This function returns the position (offset) within *string1* that occurs in *string2*. The search begins with the character at *start-pos* in *string2*. If the pattern is not found, the function returns -1.

The starting position represents an offset within a string argument. The first character in a string is at position 0. For example, in the string 'RELATION', the character 'R' is at position 0, the final character 'N' is at position 7, and the string is 8 characters long. In other words, the last position in *string1* is calculated by subtracting one from the length of *string1*.

Example

```
@FOUND( 'TRIPLETT', 'NOAH TRIPLETT', 0)
```

Returns 5.

@FV**@FV** (*pmt*, *int*, *n*)

This function returns the future value of a series of equal payments (*pmt*) earning periodic interest rate (*int*) over the number of periods (*n*).

@FV uses this formula to compute the future value of an ordinary annuity:

$$\text{pmt} * \frac{(1 + \text{int})^n - 1}{\text{int}}$$

pmt = periodic payment
int = periodic interest rate
n = number of periods

Ordinary Annuity Example

```
@FV(2000, .10, 20)
```

Returns \$114,549.999, which is the value of an account after 20 years of depositing \$2,000 at the end of each year, at an annually compounded interest rate of 10%. Interest payments and deposits are transacted on the *last day of each year*.

Annuity Due Example

```
@FV(2000, .10, 20) * (1+.10)
```

Returns \$126,005, which is the value of an annuity amount due annually. Note that this is 10% over the ordinary annuity calculated in the above example.

@HEX

@HEX (*number*)

This function returns the hexadecimal equivalent for the given decimal number.

Example

```
@HEX(10)
```

Returns 'A'.

@HOUR

@HOUR (*date*)

This function returns a number between 0 and 23 that represents the hour of the day.

Example

```
@HOUR(12/28/46 03:52:00 PM)
```

Returns 15.

@IF

@IF (*number*, *value1*, *value2*)

This function tests *number* and returns *value1* if it is TRUE (non-zero) or *value2* if it is FALSE (zero).

A non-zero argument evaluates to TRUE, and an argument of zero evaluates to FALSE. A null value evaluates to FALSE. Each value in the list is cast to the data type of the first value.

Example

```
@IF (TEST1, 'M', 'F')
```

Returns 'M' if TEST1 is non-zero (≠1), and 'F' if TEST1 is 0.

@INT

@INT (*x*)

This function returns the integer portion of *x*.

Example

```
@INT (10.2)
```

Returns 10.

```
SQL> SELECT @INT (PVAL) FROM GEOM;
```

Returns the integer portion of all PVAL column entries in the GEOM table.

@ISNA

@ISNA (*argument*)

This function returns 1 (TRUE) if the argument is NULL. Any other value returns 0 (FALSE). The argument can be any value.

Example

```
@ISNA (NULL)
```

Returns 1.

```
@ISNA('hello')
```

Returns 0.

@LEFT

@LEFT (*string, length*)

This function returns a string for the specified length, starting with the first (leftmost) character in the string.

Example

```
@LEFT('P8-196', 2)
```

Returns 'P8'.

@LENGTH

@LENGTH (*string*)

This function returns the length of a string. The length is the number of characters in the string.

You cannot use this function to find the length of a LONG VARCHAR column.

Example

If the value in the column EMPNAME is 'JOYCE':

```
@LENGTH (EMPNAME)
```

Returns the number 5.

This example finds the entries in the EMP table where the length of the REMARKS column exceeds 40 characters.

```
SQL> SELECT EMPNAME, @SUBSTRING(REMARKS, 0, 40)
FROM EMP
WHERE @LENGTH(REMARKS) > 40
```

@LICS

@LICS (*string*)

This function uses an international character set for sorting its argument, instead of the ASCII character set. This is useful for sorting characters not in the English language. The translation table for this character set is shown below.

Example

```
SQL> SELECT @LICS(NAME) FROM TAB ORDER BY 1;
```

Code	Character	Description
0	0	Ctrl @
1	1	Ctrl A
2	2	Ctrl B
3	3	Ctrl C
4	4	Ctrl D
5	5	Ctrl E
6	6	Ctrl F
7	7	Ctrl G
8	8	Ctrl H
9	9	Ctrl I
10	10	Ctrl J line feed
11	11	Ctrl K
12	12	Ctrl L form feed
13	13	Ctrl M return
14	14	Ctrl N
15	15	Ctrl O
16	16	Ctrl P
17	17	Ctrl Q
18	18	Ctrl R
19	19	Ctrl S
20	20	Ctrl T
21	21	Ctrl U

22	22	Ctrl V
23	23	Ctrl W
24	24	Ctrl X
25	25	Ctrl Y
26	26	Ctrl Z
27	27	[Esc]
28	28	FS
29	29	GS
30	30	RS
31	31	US
32	32	Space
33	33	!
34	34	"
35	35	#
36	36	\$
37	37	%
38	38	&
39	39	Apostrophe
40	40	(
41	41)
42	42	*
43	43	+
44	44	
45	45	-

46	46	.
47	47	/
48	48	0
49	49	1
50	50	2
51	51	3
52	52	4
53	53	5
54	54	6
55	55	7
56	56	8
57	57	9
58	58	:
59	59	;
60	60	<
61	61	=
62	62	>
63	63	?
64	64	@
65	65	A
66	66	B
67	67	C
68	68	D
70	69	E
71	70	F
72	71	G
73	72	H
74	73	I
75	74	J
76	75	K
77	76	L
78	77	M
79	78	N
81	79	O
82	80	P
83	81	Q
84	82	R
85	83	S
87	84	T
88	85	U

89	86	V
90	87	W
91	88	X
92	89	Y
93	90	Z
99	91	[
100	92	\
101	93]
102	94	^
103	95	_
104	96	`
65	97	a
66	98	b
67	99	c
68	100	d
70	101	e
71	102	f
72	103	g
73	104	h
74	105	i
75	106	j
76	107	k
77	108	l
78	109	m
79	110	n
81	111	o
82	112	p
83	113	q
84	114	r
85	115	s
87	116	t
88	117	u
89	118	v
90	119	w
91	120	x
92	121	y
93	122	z
105	123	{
106	124	
107	125	}

108	126	~ (tilde)
109	127	DEL
110	128	Uppercase grave
111	129	Uppercase acute
112	130	Uppercase circumflex
113	131	Uppercase umlaut
114	132	Uppercase tilde
115	133	
116	134	
117	135	
118	136	
119	137	
120	138	
121	139	
122	140	
123	141	
124	142	
125	143	
126	144	Lowercase grave
127	145	Lowercase acute
128	146	Lowercase circumflex
129	147	Lowercase umlaut
130	148	Lowercase tilde
131	149	Lowercase i without dot
132	150	Ordinal indicator
133	151	Begin attribute (display)
134	152	End attribute (display only)
135	153	Unknown character (display)
136	154	Hard space (display only)
137	155	Merge character (display)

138	156	
139	157	
140	158	
141	159	
142	160	Dutch Guilder
143	161	Inverted exclamation mark
144	162	Cent sign
145	163	Pound sign
146	164	Low opening double quotes
147	165	Yen sign
148	166	Pesetas sign
149	167	Section sign
150	168	General currency sign
151	169	Copyright sign
152	170	Feminine ordinal
153	171	Angle quotation mark left
154	172	Delta
155	173	Pi
156	174	Greater-than-or-equals
157	175	Divide sign
158	176	Degree sign
159	177	Plus/minus sign
160	178	Superscript 2
161	179	Superscript 3
162	180	Low closing double quotes
163	181	Micro sign
164	182	Paragraph sign
165	183	Middle dot
166	184	Trademark sign
167	185	Superscript 1
168	186	Masculine ordinal
169	187	Angle quotation mark right

170	188	Fraction one quarter
171	189	Fraction one-half
172	190	Less-than-or-equals
173	191	Inverted question mark
65	192	Uppercase A with grave
65	193	Uppercase A with acute
65	194	Uppercase A with circumflex
65	195	Uppercase A with tilde
65	196	Uppercase A with umlaut
65	197	Uppercase A with ring
97	197	Uppercase A with ring
94	198	Uppercase AE with ligature
67	199	Uppercase C with cedilla
70	200	Uppercase E with grave
70	201	Uppercase E with acute
70	202	Uppercase E with circumflex
70	203	Uppercase E with umlaut
74	204	Uppercase I with grave
74	205	Uppercase I with acute
74	206	Uppercase I with circumflex
74	207	Uppercase I with umlaut

69	208	Uppercase eth (Icelandic)
80	209	Uppercase N with tilde
81	210	Uppercase O with grave
81	211	Uppercase O with acute
81	212	Uppercase O with circumflex
81	213	Uppercase O with tilde
81	214	Uppercase O with umlaut
80	215	Uppercase OE with diphthong
96	216	Uppercase O with slash
88	217	Uppercase U with grave
88	218	Uppercase U with acute
88	219	Uppercase U with circumflex
88	220	Uppercase u with umlaut
92	221	Uppercase Y with umlaut
98	222	Uppercase thorn (Icelandic)
86	223	Lowercase German sharp s
65	224	Lowercase a with grave
65	225	Lowercase a with acute
65	226	Lowercase a with circumflex
65	227	Lowercase a with tilde

65	228	Lowercase a with umlaut
65	229	Lowercase a with ring
95	230	Lowercase ae with ligature
67	231	Lowercase c with cedilla
70	232	Lowercase e with grave
70	233	Lowercase e with acute
70	234	Lowercase e with circumflex
70	235	Lowercase e with umlaut
74	236	Lowercase i with grave
74	237	Lowercase i with acute
74	238	Lowercase i with circumflex
74	239	Lowercase i with umlaut
69	240	Lowercase eth (Icelandic)
80	241	Lowercase n with tilde
81	242	Lowercase o with grave
81	243	Lowercase o with acute
81	244	Lowercase o with circumflex
81	245	Lowercase o with tilde
81	246	Lowercase o with umlaut
80	247	Lowercase oe with diphthong

81	248	Lowercase o with slash
88	249	Lowercase u with grave
88	250	Lowercase u with acute
88	251	Lowercase u with circumflex
88	252	Lowercase u with umlaut
92	253	Lowercase y with umlaut
174	254	Lowercase thorn (Icelandic)

@LN

@LN (x)

This function returns the natural logarithm (base e) of (positive) *x*. The log of a zero or negative argument is handled as an overflow error.

Example

```
@LN(.1)
```

Returns -2.3025851.

```
SQL> SELECT @LN(PVAL) FROM GEOM;
```

Returns the natural logarithm of all PVAL column entries in the GEOM table.

@LOG

@LOG (x)

This function returns the (positive) base-10 logarithm of *x*. The log of a zero or negative argument is handled as an overflow error.

Example

```
@LOG(.1)
```

Returns -1.

```
SQL> SELECT @LOG(PVAL) FROM GEOM;
```

Returns the natural logarithm of all PVAL column entries in the GEOM table.

@LOWER

@LOWER (*string*)

This function converts upper-case alphabetic characters to lower-case. Other characters are not affected.

Example

```
@LOWER ( ' JOYCE ' )
```

Returns the string 'joyce'.

@MEDIAN

▶▶▶ @MEDIAN — ([ALL —] expression —) ◀◀◀
 [DISTINCT]

This function returns the middle value in a set of values. An equal number of values lie above and below the middle value.

The data type of the argument may be numeric, date/time, or character. If an argument is a character data type, the value must form a valid numeric or date/time value (only digits and standard editing characters). SQLBase will automatically convert the value to the required data type.

The data type of the result is the same as the input argument.

If the set has an *odd* number of items, @MEDIAN finds the middle value with this formula:

$$(n + 1) / 2$$

For example, if there are 5 items, then the middle item is the third:

$$(5 + 1) / 2 = 6 / 2 = 3$$

If the set has an *even* number of items, @MEDIAN finds the middle value with this formula:

$$((n / 2) + ((n / 2) + 1)) / 2$$

For example, if there are 6 items, then the middle item is the third:

$$((6 / 2) + ((6 / 2) + 1)) / 2 = (3 + (3 + 1)) / 2 = (3 + 4) / 2 = 7 / 2 = 3.5$$

Any decimal portion is dropped.

The keyword DISTINCT eliminates duplicates. If DISTINCT is not specified, then duplicates are not eliminated. Be cautious when using DISTINCT because the result may lose its statistical meaning.

Null values are ignored.

Example

This example finds the middle salary for department 10.

```
SQL> SELECT @MEDIAN(SALARY) FROM EMP WHERE DEPTNO = 10;
```

@MICROSECOND

@MICROSECOND (*date*)

This function returns the microsecond value in a DATETIME or TIME value. If a microsecond quantity was not specified on input, zero is returned.

```
@MICROSECOND (12:44:01:500000)
```

Returns 500000.

@MID

@MID (*string, start-pos, length*)

This function returns a string of specified length from a string, starting with the character at *start-pos*.

Example

This example returns the second character from a string.

```
@MID('P9-186', 1, 1)
```

Returns the character '9'.

@MINUTE

@MINUTE (*date*)

This function returns a number between 0 and 59 that represents the minute of the hour.

Example

```
@MINUTE (12/28/46 03:52:00 PM)
```

Returns 52.

@MOD

@MOD (*x*, *y*)

This function returns the modulo (remainder) of x/y . Division by zero is an overflow error.

Example

```
@MOD (5, 10)
```

Returns 5.

```
SQL> SELECT @MOD(PVAL,QVAL) FROM GEOM;
```

Returns the remainder of all PVAL/QVAL column entries in the GEOM table.

@MONTH

@MONTH (*date*)

This function returns a number between 1 and 12 that represents the month of the year.

Example

```
@MONTH (25-OCT-86)
```

Returns 10, representing October.

@MONTHBEG

@MONTHBEG (*date*)

This function returns the first day of the month represented by the date.

Example

If the value in BIRTHDATE is '16-FEB-1947', then:

```
@MONTHBEG (BIRTHDATE)
```

Returns 01-FEB-1947.

@NOW

@NOW

This function returns the current date and time. It returns the same value as the system keyword SYSDATETIME.

For example, if the date and time is January 12, 1986, 3:15 PM, this function would return 12-jan-1986 03:15:00 PM.

@NULLVALUE

@NULLVALUE (x, y)

This function returns the string or number specified by *y* if *x* is null.

The data type of the returned value is the same as the data type of the *y* argument.

Example

Return "N/A" when the column is null.

```
@NULLVALUE (name, 'N/A')
```

```
SQL> SELECT @NULLVALUE(deptno, 'NOT ASSIGNED') FROM DEPT;
```

Returns the string 'NOT ASSIGNED' if the deptno column value is null, and deptno is a character column. If the column is numeric, the replacement value must be a number. For example:

```
SQL> SELECT @NULLVALUE(BONUS, 0) FROM EMP;
```

Returns a 0 if a null exists in the BONUS column. BONUS is a numeric data type.

@PI

@PI

This function returns the value Pi (3.14159265). This function has no arguments but could be used as a numeric constant in a nested set of math functions.

Example

```
10*@PI
```

Returns 31.4159265.

```
SQL> SELECT (PVAL)*@PI FROM GEOM;
```

Returns all PVAL column entries multiplied by the value Pi in the GEOM table.

@PMT

@PMT (*prin*, *int*, *n*)

This function returns the amount of each periodic payment needed to pay off a loan principal (*prin*) at a periodic interest rate (*int*) over a number of periods (*n*). @PMT uses this formula:

$$\frac{\text{prin} * \text{int}}{(1 - (1 + \text{int})^{-n})}$$

prin = principal
int = periodic interest rate
n = term

Example

```
@PMT(50000, .125/12, 30*12)
```

Returns \$533.628881, which is the value of a monthly mortgage payment for a \$50,000, 30-year mortgage at an annual interest rate of 12.5%.

@PROPER

@PROPER (*string*)

This function converts the first character of each word in a string to uppercase and other characters to lower case.

The argument must be a CHAR or VARCHAR data type.

Example

```
@PROPER('JOHANN SEBASTIAN BACH')
```

Returns 'Johann Sebastian Bach'.

@PV

@PV (*pmt*, *int*, *n*)

This function returns the present value of a series of equal payments (*pmt*) discounted at periodic interest rate (*int*) over the number of periods (*n*).

This function is useful when trying to decide the best way to receive a payment option, over time or immediately.

@PV uses this formula:

$\text{pmt} * \frac{(1-(1+\text{int})^{-n})}{\text{int}}$	<p>pmt = periodic payment int = periodic interest rate n = term</p>
---	---

Ordinary Annuity Example

@PV(50000, .12, 20)

Returns \$373,472.181, which is what \$1,000,000 paid equally (\$50,000 at the end of each year) over 20 years at 12% is worth today.

Annuity Due Example

@PV(50000, .12, 20) * (1+.12)

Returns \$418,289, which is what \$1,000,000 paid equally (\$50,000 at the beginning of each year) over 20 years at 12% is worth today.

@QUARTER

@QUARTER (*date*)

This function returns a number between 1 and 4 that represents the quarter. The first quarter of the year is January through March, etc.

Example

```
@QUARTER (12-MAR-80)
```

Returns 1, representing the first quarter.

@QUARTERBEG

@QUARTERBEG (*date*)

This function returns the first day of the quarter represented by the date.

Example

```
@QUARTERBEG (04-JUL-1776)
```

Returns 01-JUL-1776.

This example totals the sales for each quarter and displays the first day of the quarter.

```
SQL> SELECT @QUARTERBEG(SALES_DATE), SUM(SALES)
FROM SALES_TABLE GROUP BY 1;
```

@RATE

@RATE (*fv*, *pv*, *n*)

This function returns the interest rate for an investment of present value (*pv*) to grow to a future value (*fv*) over the number of compounding periods (*n*).

@RATE uses this formula:

$$\left(\frac{fv}{pv}\right)^{\frac{1}{n}} - 1$$

fv = future value
pv = present value
n = term

Example

@RATE(18000,10000,5*12)

Returns .009844587, which is the periodic (monthly) interest rate calculated for a \$10,000 investment for 60 months (5 years) with a maturity value of \$18,000 (compounded monthly).

@REPEAT

@REPEAT (*string*, *number*)

This function concatenates a string with itself for the specified number of times. This creates a string of pattern repetitions.

This function returns nulls if specified in a select list. However, it can be used in a WHERE clause and in other contexts.

Example

```
@REPEAT ('$', 5)
```

Returns the value '\$\$\$\$\$'.

@REPLACE

@REPLACE (*string1*, *start-pos*, *length*, *string2*)

This function returns a string in which characters from *string1* have been replaced with characters from *string2*. The replacement *string2* begins at *start-pos*, the position at which characters of the specified *length* have been removed.

The first position in the string is 0.

Example

```
@REPLACE('RALF', 3, 1, 'PH')
```

Returns the value 'RALPH'.

@RIGHT

@RIGHT (*string, length*)

This function returns a specified number of characters starting from the end, or rightmost part, of a string.

Example

```
@RIGHT('P4-186', 3)
```

Returns '186'.

@ROUND

@ROUND (*x*, *n*)

This function rounds the number *x* with *n* decimal places. The rounding can occur to either side of the decimal point.

Example

```
@ROUND (@PI*10, 2)
```

Returns 31.42.

```
@ROUND (1234.1234, -2)
```

Returns 1200.

```
SQL> SELECT @ROUND (PVAL, 2) FROM GEOM;
```

Returns the value of all PVAL column entries in the GEOM table, rounded to 2 decimal places to the RIGHT of the decimal point.

```
SQL> SELECT @ROUND (PVAL, -2) FROM GEOM;
```

Returns the value of all PVAL column entries in the GEOM table, rounded to 2 decimal places to the LEFT of the decimal point.

@SCAN

@SCAN (*string, pattern*)

This function searches a given string for a specified pattern and returns a number indicating the numeric position of the first instance of the pattern.

This function returns null if the column being scanned is null.

The first position in the string is position 0. The match is performed without regard to case.

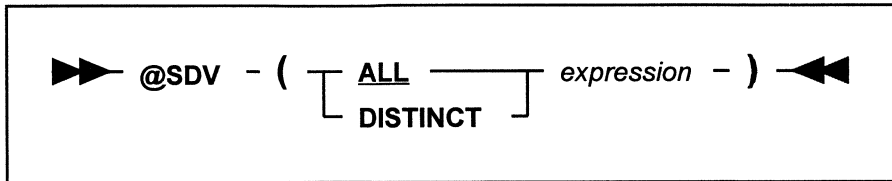
If the result is -1, it indicates no match was found.

The @SCAN function can perform a case-insensitive match on columns of type CHAR, VARCHAR, and LONG VARCHAR.

Example

```
@SCAN('P-186', '-')
```

Returns 1 as the start position of the character '-'.

@SDV

This function computes the standard deviation for the set of values specified by the argument.

The data type of the argument may be numeric, date/time, or character. If an argument is a character data type, the value must form a valid numeric or date/time value (only digits and standard editing characters). SQLBase will automatically convert the value to the required data type.

The keyword `DISTINCT` eliminates duplicates. If `DISTINCT` is not specified, then duplicates are not eliminated.

Example

```
SQL> SELECT @SDV(SALARY) FROM DEPT;
```

Returns the standard deviation of the SALARY column in the table DEPT.

```
SQL> SELECT @SDV(PRICE * QUANTITY) FROM ORDERS;
```

Returns the standard deviation of the order totals for the table ORDERS.

@SECOND

@SECOND (date)

This function returns a number between 0 and 59 that represents the second of the minute.

Example

```
@SECOND (12/28/46 03:52:58)
```

Returns 58.

@SIN

@SIN (x)

This function returns the sine of x.

Example

```
@SIN (1)
```

Returns .841470985

```
SQL> SELECT @SIN(PVAL) FROM GEOM;
```

Returns the value of all PVAL column entries in the GEOM table.

@SLN

@SLN (*cost, salvage, life*)

This function returns the straight-line depreciation allowance of an asset for each period, given the base cost, predicted salvage value, and expected life of the asset.

@SLN uses this formula to compute depreciation:

$$\frac{(c-s)}{n}$$

c = cost of the asset
s = salvage value of the asset
n = useful life of the asset

Example

```
@SLN(10000,1200,8)
```

Returns \$1100, which is the yearly depreciation allowance for a machine purchased for \$10,000, with a useful life of 8 years, and a salvage value of \$1200 after the 8 years.

@SQRT

@SQRT (*x*)

This function returns the square root of *x* (which must be zero or positive). The square root of a negative argument is handled as an overflow error.

Example

```
@SQRT(10)
```

Returns 3.16227766.

```
SQL> SELECT @SQRT(PVAL) FROM GEOM;
```

Returns the square root of all PVAL column entries in the GEOM table.

@STRING

@STRING (*number*, *scale*)

This function converts a number into a string with the number of decimal places specified by *scale*. Numbers are rounded where appropriate.

Example

```
@STRING(123.456, 2)
```

Returns the character string '123.46'.

@SUBSTRING

@SUBSTRING (*string*, *start-pos*, *length*)

This function returns a desired portion of a string from a given argument string. The substring starts at the specified start position and is of the specified length. If the start position and length define a substring that exceeds the actual length of the string, the result is truncated to the actual length of the string. If the start position is beyond length of the string, a null string (") is returned. The first character in a string is at start-pos 0.

Example

```
@SUBSTRING('DR. SMITH', 4, 20)
```

Returns 'SMITH'.

This example returns the first 40 characters of the REMARKS column in the EMP table where the length of the REMARKS column exceeds 40 characters.

```
SQL> SELECT ENAME, @SUBSTRING(REMARKS, 0, 40)
FROM EMP
WHERE @LENGTH(REMARKS) > 40;
```

@SYD

@SYD (*cost, salvage, life, period*)

This function returns the Sum-of-the-Years'-Digits depreciation allowance of an asset for a given period, given the base cost, predicted salvage value, expected life of the asset and specific period.

@SLN uses this formula to compute depreciation:

$$\frac{(c-s) * (n-p+1)}{(n * (n+1) / 2)}$$

c = cost of the asset
s = salvage value of the asset
p = period for which depreciation is being computed
n = useful life of the asset

Example

@SYD (10000, 1200, 8, 5)

Returns \$978, which is the depreciation allowance for the fifth year for a \$10,000 machine with a useful life of 8 years, and a salvage value of \$1200 after the 8 years.

@TAN

@TAN (x)

This function returns the tangent of x.

Example

```
@TAN (10)
```

Returns .648360827.

```
SQL> SELECT @TAN(PVAL) FROM GEOM;
```

Returns the tangent of all PVAL column entries in the GEOM table.

@TERM

@TERM (*pmt*, *int*, *fv*)

This function returns the number of payment periods for an investment, given the amount of each payment *pmt*, the periodic interest rate *int*, and the future value *fv* of the investment.

@TERM uses this formula to compute the term:

$$\frac{\ln(1+(fv*int/pmt))}{\ln(1+int)}$$

pmt = periodic payment
fv = future value
int = periodic interest rate
ln = natural logarithm

Example

@TERM(2000, .10, 100000)

Returns 18.7992455, which is the number of years it will take for an investment to mature to an amount of \$100,000. This is based on a yearly deposit of \$2,000 at the end of each year to an account that earns 10% compounded annually.

@TIME

@TIME (*hour, minute, second*)

This function returns a date/time value given the *hour*, *minute*, and *second*. An hour is a number from 0 to 23; a minute is a number from 0 to 59; a second is a number from 0 to 59.

The date '30-DEC-1899' represents day 0 in the internal floating point format. It can be truncated (through SQLTalk) with a COLUMN picture format.

Example

```
@TIME (13, 0, 0)
```

Returns 13:00:00.

@TIMEVALUE

@TIMEVALUE (*time*)

The function returns a date/time value, given a string in the form HH:MM:SS [AM or PM]. If the AM or PM parameter is omitted, military time is used.

Example

If the CHAR column APPT contains '18-JAN-1987 10:14:27 AM', then:

```
@TIMEVALUE (APPT)
```

Returns 10:14:27.

@TRIM

@TRIM (*string*)

This function strips leading and trailing blanks from a string and compresses multiple spaces within the string into single spaces.

Example

```
@TRIM('   JOHN   DEWEY  ')
```

Returns 'JOHN DEWEY'.

@UPPER

@UPPER (*string*)

This function converts lower-case letters in a string to upper-case. Other characters are not affected.

Example

```
@UPPER('e.e. cummings')
```

Returns 'E.E. CUMMINGS'.

@VALUE

@VALUE (*string*)

This function converts a character string that has the digits (0-9) and an optional decimal point into the number represented by that string.

Example

```
@VALUE ('123456')
```

Returns the number 123456 which will be interpreted strictly as a numeric data type by any function to which it is passed.

@WEEKBEG

@WEEKBEG (*date*)

This function returns the date of the Monday of the week containing the date. This is the previous Monday if the date is not a Monday and the date value itself if it is a Monday.

Example

If the value in DATECOL is 01/FEB/87, then:

```
@WEEKBEG (DATECOL)
```

Returns 01/26/87.

@WEEKDAY

@WEEKDAY (<i>date</i>)

This function returns a number between 0 and 6 (Saturday = 0) that represents the day of the week.

Example

```
@WEEKDAY (12/28/86)
```

Returns 1, representing SUNDAY.

This example finds the day of the week on which each employee was born.

```
SQL> SELECT @CHOOSE (@WEEKDAY(BIRTH_DATE), 'Sat','Sun',  
'Mon', 'Tue', 'Wed', 'Thu', 'Fri'), @YEAR(BIRTH_DATE)  
FROM EMP  
WHERE @YEARN0 (BIRTH_DATE) > 1900;
```

@YEAR

@YEAR (*date*)

This function returns a number between -1900 and +200 that represents the year relative to 1900. The year 1900 is 0, 1986 is 86, and 2000 is 100. Years before 1900 are negative numbers and 1899 is -1.

Example

```
@YEAR (12/28/1923)
```

Returns 23.

This example finds the day of the week on which each employee was born.

```
SQL> SELECT @CHOOSE (@WEEKDAY (BIRTH_DATE), 'Sat', 'Sun',  
'Mon', 'Tue', 'Wed', 'Thu', 'Fri'), @YEAR (BIRTH_DATE)  
FROM EMP  
WHERE @YEARNO (BIRTH_DATE) > 1900;
```

@YEARBEG

@YEARBEG (*date*)

This function returns the first day of the year represented by the date.

Example

If the value in BIRTHDATE is '16-FEB-1947', then:

```
@YEARBEG (BIRTHDATE)
```

Returns 01-JAN-1947.

@YEARNO

@YEARNO (*date*)

This function returns a 4-digit number that represents a calendar year.

Example

If the column HISTORIC_DATE contains the value 04/JUL/1776, then:

```
@YEARNO (HISTORIC_DATE)
```

Returns 1776.

This example finds the day of the week on which each employee was born.

```
SQL> SELECT @CHOOSE (@WEEKDAY(BIRTH_DATE), 'Sat','Sun',  
'Mon', 'Tue', 'Wed', 'Thu', 'Fri'), @YEAR(BIRTH_DATE)  
FROM EMP  
WHERE @YEARNO (BIRTH_DATE) > 1900;
```


Chapter 9

SQL Reserved Words

About this Chapter

The following words are reserved in SQL.

You can use a reserved word as an identifier if it is enclosed in double quotes, but this is *not* recommended.

@ABS	@LOG
@ACOS	@LOWER
@ASIN	@MEDIAN
@ATAN	@MICROSECOND
@ATAN2	@MID
@CHAR	@MINUTE
@CHOOSE	@MOD
@CODE	@MONTH
@COS	@MONTHBEG
@CTERM	@NOW
@DATE	@NULLVALUE
@DATETOCHAR	@PI
@DATEVALUE	@PMT
@DAY	@PROPER
@DECIMAL	@PV
@DECODE	@QUARTER
@DECRYPT	@QUARTERBEG
@EXACT	@RATE
@EXP	@REPEAT
@FACTORIAL	@REPLACE
@FIND	@RIGHT
@FV	@ROUND
@HEX	@SCAN
@HOUR	@SDV
@IF	@SECOND
@INT	@SIN
@ISNA	@SLN
@LEFT	@SQRT
@LENGTH	@STRING
@LICS	@SUBSTRING
@LN	@SYD

@TAN
@TERM
@TIME
@TIMEVALUE
@TRIM
@UPPER
@VALUE
@WEEKBEG
@WEEKDAY
@YEAR
@YEARBEG
@YEARNO
ABORTxxxDBSxxx
ADD
ADJUSTING
ALL
ALTER
AND
ANY
AS
ASC
AVG
BETWEEN
BY
CASCADE
CHAR
CHARACTER
CHECK
CLUSTERED
COLUMN
COMMENT
COMMIT
COMPUTE
CONNECT
COUNT
CREATE
CURRENT
DATABASE
DATE
DATETIME
DAY
DAYS
DBA
DBAREA
DEC
DECIMAL
DEFAULT
DEINSTALL
DELETE
DESC
DIRECT
DISTINCT
DOUBLE
DROP
EXISTS
FLOAT
FOR
FOREIGN
FROM
GRANT
GROUP
HASHED
HAVING
HOUR
HOURS
IDENTIFIED
IN
INDEX
INSERT
INSTALL
INT
INTEGER
INTO
IS
KEY
LABEL
LIKE
LOG

LONG	SIZE
MAX	SMALLINT
MICROSECOND	STATISTICS
MICROSECONDS	STOGROUP
MIN	SUM
MINUTE	SYNONYM
MINUTES	SYSDATE
MODIFY	SYSDATETIME
MONTH	SYSTEM
MONTHS	SYSTEMTIMEZONE
NOT	TABLE
NULL	TIME
NUMBER	TIMESTAMP
OF	TIMEZONE
ON	TO
OPTION	UNION
OR	UNIQUE
ORDER	UPDATE
PASSWORD	USER
PCTFREE	USERERROR
POST	USING
PRECISION	VALUES
PRIMARY	VARCHAR
PUBLIC	VIEW
REAL	WAIT
REFERENCES	WHERE
RENAME	WITH
RESOURCE	WORK
RESTRICT	YEAR
REVOKE	YEARS
ROLLBACK	
ROWCOUNT	
ROWID	
ROWS	
SAVEPOINT	
SECOND	
SECONDS	
SELECT	
SET	

Chapter 10

System Catalog

About this Chapter

This chapter contains descriptions of the tables in the system catalog, organized alphabetically by name. The order of the columns in the system catalog tables is the same as in DB2.

System Catalog Summary

The system catalog is a set of tables that contain information about objects in the database. The tables are maintained by SQLBase. The system catalog is also called the *data dictionary*.

Table Name	Contents
SYSCOLAUTH	Each user's column update privileges.
SYSCOLUMNS	Each column in the database.
SYSCOMMANDS	Each stored command in the database.
SYSFKCONSTRAINTS	Each constraint on foreign keys in the database.
SYSINDEXES	Each index in a table.
SYSKEYS	Each column in an index.
SYSPKCONSTRAINTS	Each constraint on primary keys in the database.
SYSSYNONYMS	Each synonym of a table or view.
SYSTABAUTH	Each user's table privileges.
SYSTABCONSTRAINTS	Each constraint on tables in the database.
SYSTABLES	Each table or view in the database.
SYSUSERAUTH	Each user's database authority level.
SYSVIEWS	Each view in the database.

System Table Operations

The above tables are all owned by SYSADM. Users with the proper privileges can perform the following operations on system tables:

- Select rows.
- Create a view.
- Create a synonym.
- Create an index on a column.
- Add user-defined columns.
- Drop user-defined columns.
- Update user-defined columns.

You cannot DROP a system table or system-defined column, nor can you INSERT or DELETE rows in a system table.

The privilege to perform any of the above operations must be granted explicitly to a user by the SYSADM or by a user with DBA authority. Users who do not have these privileges cannot access the system tables directly.

System Catalog Views

Views for a particular user can be created from the system tables. Use the keyword USER in the search condition for the view. For example, a view called TABLES could contain the names of all the tables that can be accessed with a given authorization-id; a view called COLUMNS could contain the names of all the columns in those tables, and so forth. These views would be owned by SYSADM and other users can select from these views but cannot modify them.

Although you can add columns to a system catalog table, SQLBase does not maintain them. For example, the UNLOAD command ignores user-defined columns.

SYSADM.SYSCOLAUTH
(Column Authorizations)

This table contains the UPDATE privileges of users for individual columns of a table or view.

Column Name	Description
GRANTEE	The authorization-id of the user who holds update privileges.
CREATOR	The authorization-id of the user who created the table on which the update privileges are held.
TNAME	The name of the table or view on which privileges are held.
COLNAME	The name of the column to which the UPDATE privilege applies.

SYSADM.SYSCOLUMNS **(Columns)**

This table contains one row for every column of each table and view (including the columns of the system catalog tables).

Column Name	Description
TBCREATOR	The authorization-id of the user who created the table or view in which the column exists.
NAME	The name of the column.
TBNAME	The name of the table or view that contains this column.
COLNO	The relative column number in the table. This number remains the same even after a column is dropped.
COLTYPE	The data type of the column. Column data types are the same as those used in DB2.
LABEL	A user-specified label about each row. The maximum length of a label is 30 characters.

Column Name	Description
LENGTH	<p>The length of the data in the column, or for a DECIMAL column, the precision.</p> <p>The standard lengths for the data types are:</p> <p>INTEGER 4 SMALLINT 2 FLOAT 8 CHAR Length of string VARCHAR Maximum length of string DECIMAL Precision of number</p>
NULLS	<p>Contains a "Y" if nulls are allowed in the column; "N" if defined with NOT NULL; "D" if defined with NOT NULL WITH DEFAULT.</p>
UPDATES	<p>Contains a "Y" if the column can be updated; "N" if the column is read-only.</p>
REMARKS	<p>A user-specified comment about each row. The maximum length of a comment is 254 characters.</p>
SCALE	<p>The scale for a DECIMAL data type column.</p>

SYSADM.SYSCOMMANDS **(Stored Commands)**

This table contains one row for every stored command.

Column Name	Description
CREATOR	Authorization-id of the creator of the stored command.
NAME	The name of the stored command.
TEXT	The text of the stored command.

SYSADM.SYSFKCONSTRAINTS (Foreign Key Constraints)

This table contains the foreign key columns of a table.

Column Name	Description
CONSTRAINT	The name of the foreign key constraint.
CREATOR	The authorization-id of the user who created the table.
FKCOLSEQNUM	The sequence number of the foreign key column within the foreign key.
NAME	The name of the column to which the foreign key applies.
REFDCOLUMN	The column name of the referenced column in the parent table.
REFDTBCREATOR	The authorization-id of the user who created the parent table being referenced by the foreign key.
REFDTBNAME	The name of the parent table which is referenced by the foreign key.
REFSCOLUMN	The relative column number of the foreign key column.

SYSADM.SYSINDEXES
(Indexes)

This table contains one row for every index, including indexes on catalog tables.

Column Name	Description
TBCREATOR	Authorization-id of the creator of the table which contains the index.
NAME	The name of the index.
TBNAME	The name of the table on which the index is defined.
CREATOR	Authorization-id of the creator of the index.
UNIQUERULE	Indicates if the index is unique. "D" if duplicates are allowed. "U" if each index must be unique.
COLCOUNT	The number of columns in the index.
IXTYPE	Indicates the type of index. "H" if clustered hashed, "B" if B-tree.
CLUSTERRULE	"N" if B-tree, "Y" if clustered-hashed.
IXSIZE	The number of rows for the table as specified by the user.
PERCENTFREE	The amount of free space to leave in each index page when the index is first built. If specified by the user, the default of 10 percent is used.

SYSADM.SYSKEYS
(Index Keys)

This table contains one row for each column in an index.

Column Name	Description
IXCREATOR	Authorization-id of the creator of the index.
IXNAME	The name of the index.
COLNAME	The name of the column of the key.
COLNO	The numerical position of the column in the row; for example 2 (out of 7).
COLSEQ	The numerical position of the column in the key; for example 2 (out of 3).
ORDERING	The order of the column in the key. "A" if ascending and "D" if descending.
FUNCTION	The function definition used to define the key.

SYSADM.SYSPKCONSTRAINTS ***(Primary Key Constraints)***

This table contains the primary key columns of a table.

Column Name	Description
COLNAME	The name of the column to which the primary key applies.
CREATOR	The authorization-id of the user who created the table.
NAME	The name of the table to which the primary key applies.
PKCOLSEQNUM	The relative column number of the primary key column.

SYSADM.SYSSYNONYMS (Synonyms)

This table contains one row for each synonym of a table or view.

Column Name	Description
NAME	Synonym for the table or view.
CREATOR	Authorization-id of the creator of the synonym.
TBNAME	Name of the table or view.
TBCREATOR	Authorization-id of the creator of the table or view.

SYSADM.SYSTABAUTH
(Table and View Privileges)

This table contains the privileges of users for tables or views.

Column Name	Description
GRANTEE	Authorization-id of the user who holds the privileges described in this row.
TCREATOR	Authorization-id of the user who created the table or view on which privileges are held.
TTNAME	The name of the table or view on which privileges are held. "T" stands for target.
UPDATECOLS	Contains a "*" if the user has update privileges on only some of the columns in the target table. The column names for which privileges are held are contained in the SYSCOLAUTH table. If the user holds no update privileges or if update is allowed on the entire table, then this column contains a blank.
ALTERAUTH	Contains a "Y" if the user is allowed to alter the target table. Otherwise it contains a blank.
DELETEAUTH	Contains a "Y" if the user is allowed to delete rows from the target table or view. Otherwise it contains a blank.
INDEXAUTH	Contains a "Y" if the user is allowed to create or drop indexes on the target table. Otherwise it contains a blank.

Column Name	Description
INSERTAUTH	Contains a "Y" if the user is allowed to insert rows into the target table or view. Otherwise it contains a blank.
SELECTAUTH	Contains a "Y" if the user is allowed to read rows from the target table or view. Otherwise it contains a blank.
UPDATEAUTH	Contains a "Y" if the user is allowed to update rows in the target table or view. Otherwise it contains a blank.

SYSADM.SYSTABCONSTRAINTS

(Table Constraints)

This table contains the table constraints.

Column Name	Description
CONSTRAINT	The name of the constraint which is being used by the database. This column also shows the name of the foreign key. For the primary key, it is set to "PRIMARY."
CREATOR	The authorization-id of the user who created the table.
DELETERULE	<p>The DELETE rule being followed by the database. This column is not used for primary key constraints.</p> <p>This column can be set to:</p> <p>CASCADE C SET NULL N RESTRICT R</p> <p>The default is RESTRICT.</p>
NAME	The name of the table to which the constraint applies.
TYPE	The type of constraint. This column is set to 'P' for a primary key, and 'F' for a foreign key constraint.
USRERRINSDEP	The user-specified error message number for an INSERT_DEPENDENT violation. If there was no user-specified error, this column is set to 0.

Column Name	Description
USRERRUPDDEP	The user-specified error message number for an UPDATE_DEPENDENT violation. If there was no user-specified error, this column is set to 0.
USRERRDELPAR	The user-specified error message number for a DELETE_PARENT violation. If there was no user-specified error, this column is set to 0.
USRERRUPDPAR	The user-specified error message number for an UPDATE_PARENT violation. If there was no user-specified error, this column is set to 0.

SYSADM.SYSTABLES **(Tables and Views)**

This table contains one row for each table or view.

Column Name	Description
CREATOR	Authorization-id of the user who created the table or view.
NAME	The name of the table or view.
COLCOUNT	The number of columns in the table or view.
TYPE	"T" if the row describes a table or "V" if the row describes a view.
REMARKS	A user-specified comment about each row. The maximum length of a comment is 254 characters.
SNUM	Serial number used by UNLOAD DATABASE.
LABEL	A user-specified label about each table and row. The maximum length of a label is 30 characters.
PERCENTFREE	The amount of free space left in each table row page when it is initially filled. The default value is 10 percent.

SYSADM.SYSUSERAUTH
(User Authority)

This table contains the database authority level of each user.

Column Name	Description
NAME	Authorization-id of each valid user of the database. A user is valid if he has CONNECT authority.
RESOURCEAUTH	"Y" if the user can create or drop tables or "G" if the user is SYSADM. Otherwise blank.
DBAAUTH	"Y" if the user has DBA authority, a "G" if the user is SYSADM. Otherwise blank.
PASSWORD	The encrypted password associated with the authorization-id. Use the @DECRYPT function to decode the password.

SYSADM.SYSVIEWS

(Rows in View)

This table contains one or more rows for each view.

Column Name	Description
NAME	Name of the view.
CREATOR	Authorization-id of the creator of the view.
SEQNO	Sequence number of this row. The first text portion of the view is on row one and successive rows have increasing values of SEQNO.
CHECKFLAG	Whether the CHECK option was specified in the CREATE VIEW command. "Y" means yes and "N" means no.
TEXT	The text of the CREATE VIEW command. This column is 40 characters long. If the CREATE VIEW command is longer than 40 characters, then each row contains a portion of the text.

Server-Independent System Catalog Views

The following views on the system catalog tables can be used by application programs to access catalog information in a server-independent fashion.

These views are a common subset of the system catalog tables supported by most SQL database vendors. Commands such as LOAD and UNLOAD use these views to access non-SQLBase databases (such as DB2).

View Name	Columns
SYSSQL.SYSCOLAUTH	GRANTEE CREATOR TNAME COLNAME
SYSSQL.SYSCOLUMNS	TBCREATOR NAME TBNAME COLNO COLTYPE LENGTH NULLS UPDATES REMARKS SCALE
SYSSQL.SYSINDEXES	TBCREATOR NAME TBNAME CREATOR UNIQUERULE COLCOUNT IXTYPE CLUSTERRULE IXSIZE PERCENTFREE

View Name	Columns
SYSSQL.SYSKEYS	IXCREATOR IXNAME COLNAME COLNO COLSEQ ORDERING FUNCTION
SYSSQL.SYSSYNONYMS	NAME CREATOR TBNAME TBCREATOR
SYSSQL.SYSTABAUTH	GRANTEE TCREATOR TTNAME UPDATECOLS ALTERAUTH DELETEAUTH INDEXAUTH INSERTAUTH SELECTAUTH UPDATEAUTH
SYSSQL.SYSTABLES	CREATOR NAME COLCOUNT TYPE REMARKS PERCENTFREE

Chapter 11

Referential Integrity

About this Chapter

This chapter explains referential integrity in SQLBase.

About Referential Integrity

Referential integrity means that all references from one database table to another are valid and accurate. Referential integrity prevents problems that occur because of changes in one table which are not reflected in another. For example, suppose you tried to insert a new row into the GUEST_ROSTER table on the next page. This table and the ROOMS table are related by sharing the column ROOM, which contains unique values for room numbers.

```
SQL> INSERT INTO GUEST_ROSTER VALUES ('Harvey Forman', 33,
'Todd', '1991-03-01', '1991-04-01', .1);
```

At this point, the database shows that Harvey Forman is a guest in room 33, even though there is no room 33 listed in the ROOMS table. This newly inserted row violates the relationship between the GUEST_ROSTER and ROOMS tables, because Harvey Forman was assigned to a non-existent room number.

Every value in the ROOM column of one table should match the values in the ROOM column of the other table. This rule is known as a referential integrity *constraint*. SQLBase lets you build these constraints, which removes the requirement of maintaining the integrity and accuracy at the *application* end.

Database Examples

The examples in this chapter primarily use the sample Spa database. Spa is an imaginary health club with guests, trainers, and facilities. The Spa database contains the following tables.

GUEST_ROSTER Table

Name	Room	Trainer	Arrival	Departure	Discount
Bette Midriff	1	Michael	14-jun-1990	16-jun-1990	
Marcello Histrioni	2	Todd	26-jun-1990	18-aug-1990	.1
Jean-Paul Rotundo	3	Julio	15-jun-1990	17-jun-1990	
Michael Johnson	4	Julio	15-jun-1990	19-jun-1990	
Clint Westwood	5	Michael	25-jun-1990	28-jun-1990	.15
Sean Pencil	6	Todd	15-jun-1990	20-jun-1990	
Hetaera	7	Serena	18-jul-1990	19-aug-1990	.05
Joannie Rivulets	8	Serena	25-jun-1990	28-jun-1990	.05
Warren Amoroso	9	Todd	16-jun-1990	23-jun-1990	
Marlon Spandex	10	Julio	21-jun-1990	28-jun-1990	

ROOMS Table

Room	Name	Rate
1	Annatto Rm	300
2	Carmine Rm	300
3	Cerise Rm	250
4	Cherry Rm	250
5	Maroon Rm	250
6	Puce Rm	250
7	Peach Rm	250
8	Marigold Suite	250
9	Gold Rm	250
10	Amber Rm	250

What is a Primary Key?

The primary key of a table is the column or set of columns that are used to uniquely identify each row. In the ROOMS table on the next page, the ROOM column is the primary key. If you want to find a specific row in a table, you refer to it with the primary key. Primary keys ensure the integrity of your data. If the primary key is correctly used and maintained, every row will be different from every other row, and there will be no empty rows.

The following rules apply to primary keys:

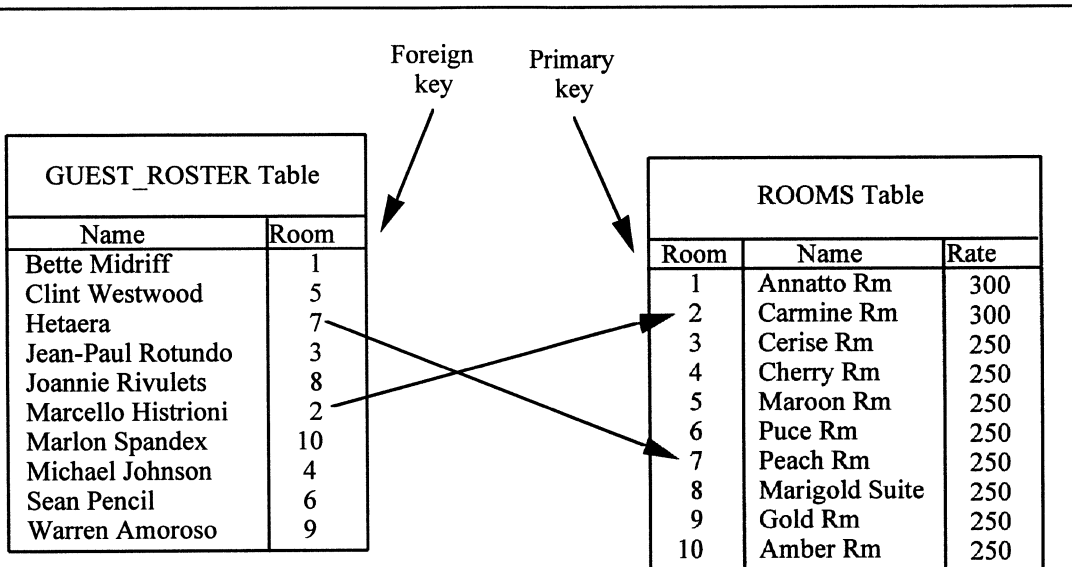
- The values of the primary key must be unique — no two rows of a table can have the same key values.
- A table can have only one primary key.
- The primary key can be made up of one or more columns in a table. If multiple columns, this is called a *composite* primary key.
- Each column in the primary key must have a NOT NULL constraint.

You can create a primary key as shown in the following example, using the ROOMS table:

```
SQL> CREATE TABLE ROOMS (ROOM VARCHAR(3) NOT NULL, NAME  
CHAR(25), RATE VARCHAR(3), PRIMARY KEY (ROOM));
```

What is a Foreign Key?

A foreign key references a primary key in the same table or another table. The ROOM column of the ROOMS table is an example of a primary key. The ROOM column of the GUEST_ROSTER table is an example of a foreign key. There is a direct relationship between the data in the two columns. When one table's column values are present in another table's column, the column from the first table refers to the second.



There are a few rules necessary to maintain referential integrity:

- A primary and a foreign key can be made up of one or more columns of a table.
- The foreign key and primary key must contain the same number of columns.
- The foreign key's data types must match those of the primary key on a one-to-one basis, and the matching columns must be in the same order.
- The primary key must have a unique index on it.

You can use the ALTER TABLE statement to add a foreign key to an existing table. To add a foreign key, you must have the ALTER privilege on both the table containing the foreign key and the table containing the primary key.

For example, if you wanted to add a foreign key to the sample GUEST_ROSTER table that references the ROOMS table, you would use this statement:

```
SQL> ALTER TABLE GUEST_ROSTER
FOREIGN KEY ROOM_REF (ROOM) REFERENCES ROOMS
ON DELETE RESTRICT;
```

When you add a foreign key, the primary key must already exist.

The following rules apply to foreign keys:

- A table can have many foreign keys.
- A foreign key column value can be NULL.
- A foreign key value is NULL if any part is NULL.

The foreign key can be assigned a *constraint name* in a SQLBase statement. If no name is assigned by the user, the default name is the first column name of the foreign key column list.

Parent and Dependent Tables and Rows

In a database with referential integrity, the table containing the primary key is called the parent table, while the table containing the foreign key is a dependent table. A dependent of a dependent is called a descendent.

A row of a parent table that is referred to by a particular row of the dependent table is a parent row. The row that refers to it is a dependent row.

How to Create Tables with Referential Constraints

You can use the CREATE TABLE or ALTER TABLE statement to create or alter tables with primary keys or foreign keys, and to establish referential constraints.

You can define a single primary key composed of specific columns by using the CREATE TABLE statement. However, the definition of the table is incomplete until you generate its unique index. Unique indexes are explained on the next page.

For example, you can add a referential constraint to the definition of the tables GUEST _ ROSTER and ROOMS, as follows:

```
SQL> CREATE TABLE GUEST_ROSTER (NAME CHAR(25),  
ROOM VARCHAR(3), TRAINER VARCHAR(8),  
ARRIVAL DATE, DEPARTURE DATE, DISCOUNT  
DECIMAL(2,2),  
FOREIGN KEY ROOM_REF (ROOM) REFERENCES ROOMS  
ON DELETE RESTRICT);  
  
CREATE TABLE ROOMS (ROOM VARCHAR(3) NOT NULL,  
NAME CHAR(25), RATE VARCHAR(3),  
PRIMARY KEY (ROOM));
```

A delete rule, RESTRICT, was specified in the above example. Delete rules are discussed later in this chapter.

Once you have defined the referential constraint, SQLBase will enforce the constraint on every UPDATE, INSERT, and DELETE operation. This is discussed later in this chapter.

Creating a UNIQUE Index

If a table is created with a primary key, it must also have a unique index created on its primary key columns, with the same order of columns as the primary key columns. The unique index can be in either ascending or descending order. The table is in an *incomplete* state until the unique index is created. Use of a table in an incomplete state is limited — you cannot load the table, insert data, retrieve data, or create foreign keys that reference the primary key.

Because of these limitations, you should plan to create the unique index soon after creating the table. For example, to create the unique index for the ROOMS table, enter:

```
SQL> CREATE UNIQUE INDEX ROOM_IDX  
ON ROOMS (ROOM);
```

If the primary key is added later with ALTER TABLE, then a unique index on the key columns must already exist.

If a unique index is dropped later, the table becomes incomplete again, and no data manipulation operations are possible until the unique index is recreated.

Deleting from Tables

You can specify a delete rule for each parent/dependent relationship created by a foreign key in a SQLBase application. The delete rule tells SQLBase what to do when a user tries to delete a row of the parent table. You can specify one of three delete rules, as explained in the following sections.

DELETE RESTRICT

This rule prevents you from deleting a row from the parent table if the row has any dependents. A DELETE statement that attempts to delete such a parent row causes an error message.

For example, you cannot delete a room from the ROOMS table if a guest is still occupying it.

```
SQL> ALTER TABLE GUEST_ROSTER
FOREIGN KEY ROOM_REF (ROOM) REFERENCES ROOMS
ON DELETE RESTRICT;
```

If you do not specify a DELETE rule, RESTRICT is the default.

DELETE CASCADE

This rule specifies that when a parent row is deleted, all of its dependent rows should be automatically deleted from the dependent table. In this way, deletions from the parent table *cascade* to the dependent table.

For example, you can delete a room by deleting its row in the ROOMS table. That also deletes the guest who occupies the room.

```
SQL> ALTER TABLE GUEST_ROSTER
FOREIGN KEY ROOM_REF (ROOM) REFERENCES ROOMS
ON DELETE CASCADE;
```

The CASCADE rule should be used with caution, because it can cause extensive automatic deletion of data if it is used incorrectly.

DELETE SET NULL

This rule specifies that when a parent row is deleted, the foreign key values in all of its dependent rows should automatically be set to NULL. For example, if a room is deleted from the ROOMS table, the room assignment for its guest is unknown.

```
SQL> ALTER TABLE GUEST_ROSTER  
FOREIGN KEY ROOM_REF (ROOM) REFERENCES ROOMS  
ON DELETE SET NULL;
```

Implications for SQLBase Operations

Referential constraints have special implications for some SQLBase operations that depend on whether the table is a parent or dependent. This section describes the effects of the rules for referential constraints on the SQLBase INSERT, UPDATE, and DROP operations.

INSERT Implications

There are several rules to follow when you insert data into tables. If you are inserting into a dependent table with foreign keys:

- Each non-null value you insert into a foreign key column must be equal to some value in the primary key, otherwise an error is returned. See the section *Customizing SQLBase Error Messages* later in this chapter.
- If any column in the foreign key is null, the entire foreign key is regarded as null. If all foreign key columns are null, the INSERT will succeed.
- If the index enforcing the primary key of the parent table has been dropped, the INSERT into either the parent table or dependent table will fail with an error message.

For example, the GUEST_ROSTER table has a foreign key on the ROOM column, referencing the ROOMS table. Every row inserted into the GUEST_ROSTER table must have a value that is equal to a value in the ROOM column in the ROOMS table. (The null value is not allowed because the ROOM column in the ROOMS table was defined as NOT NULL.)

UPDATE Implications

The UPDATE statement updates tables with primary or foreign keys. Any non-null foreign key values that you enter must match the primary key for each relationship in which the table is a dependent.

The only UPDATE rule that can be applied to a parent table is RESTRICT. This means that any attempt to update the primary key of the parent table is restricted to cases where there are no matching values in the dependent tables.

SQLBase enforces these rules on UPDATES:

- An UPDATE statement that assigns a value to a primary key *cannot* specify more than one record.
- The UPDATE WHERE CURRENT OF statement is not allowed to update a primary key.

If an UPDATE against a table with a referential constraint fails, an error message is returned. See the section *Customizing SQLBase Error Messages* later in this chapter.

DROP Implications

Dropping a table drops its primary key and also drops any foreign keys in other tables that reference the parent table. When the parent table of the relationship is dropped, or when the primary key of the parent table is dropped, the referential constraint is also dropped.

□ *Dropping a Primary Key*

If you have the ALTER privilege on both the parent and dependent tables, you can drop a primary key with a statement such as this one:

```
SQL> ALTER TABLE ROOMS  
DROP PRIMARY KEY;
```

The statement drops the primary key of the sample ROOMS table. If the table has a dependent, the relationship between them is dropped.

Dropping a primary key drops all the referential relationships in which the table is a parent. Before you drop a primary key, you should consider the effect this will have on your application programs. The primary key of a table is intended to serve as a permanent, unique identifier of the entities it describes. It is likely that some of your programs depend on the primary key. Without it, your programs must enforce referential constraints.

□ **Dropping a Foreign Key**

To drop a foreign key, name its relationship in a statement like this one:

```
SQL> ALTER TABLE GUEST_ROSTER  
DROP FOREIGN KEY ROOM_REF;
```

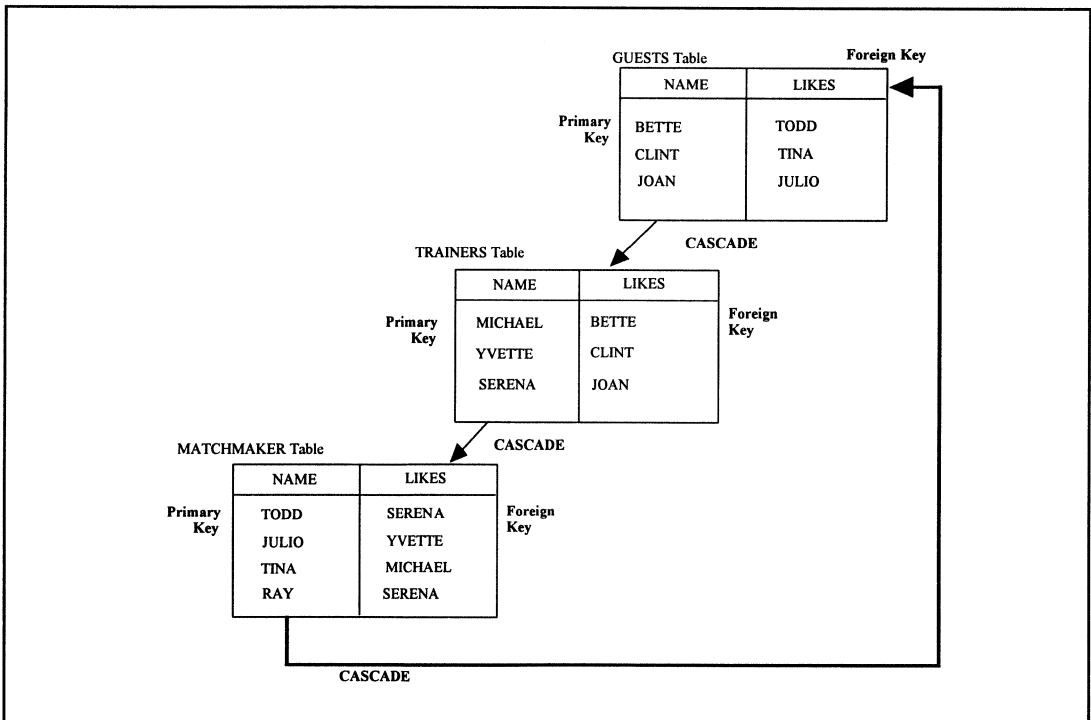
To drop a foreign key, you must have the ALTER privilege on both the parent and dependent tables.

Before you drop a foreign key, consider the effect this will have on your application programs. Dropping a foreign key drops the corresponding referential relationship and delete rule. Without this key, your programs must enforce these constraints.

Cycles of Dependent Tables

In the example below there are three tables which form a cycle. The GUESTS table has three guests and the trainers they like, the TRAINERS table has three trainers and the guests they like, and the MATCHMAKER table has four trainers who like other trainers. There is a foreign key matching the NAMES column of the MATCHMAKER table with the LIKES column of the GUESTS table.

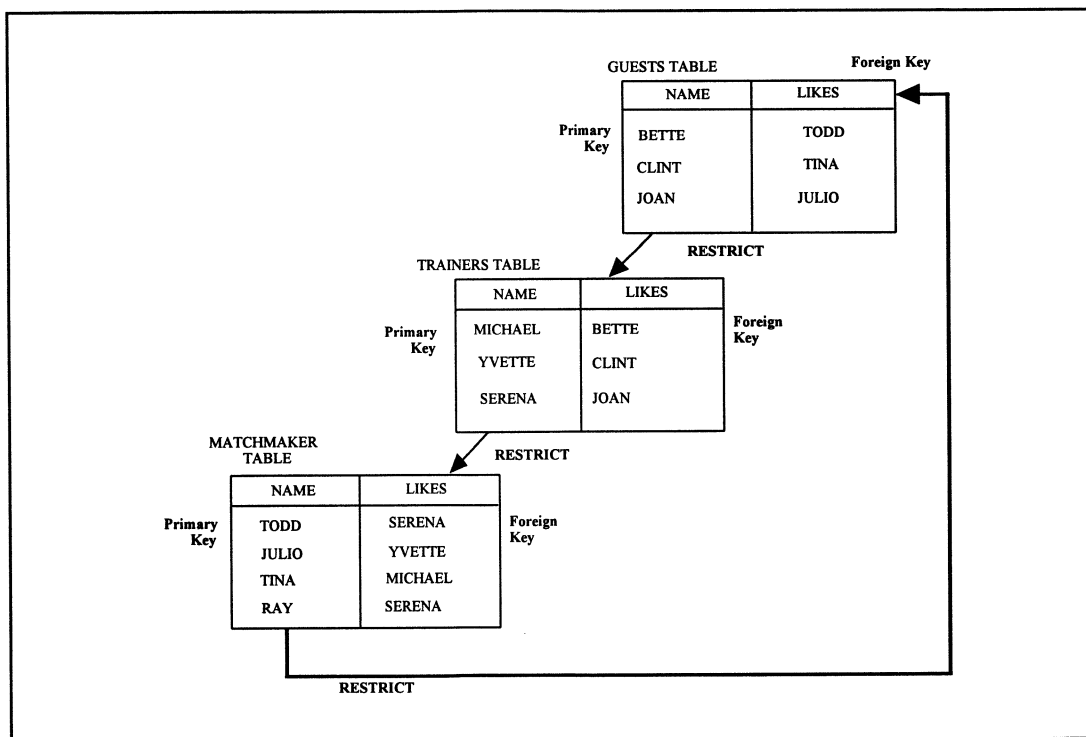
The example below shows what happens if all the tables are using the DELETE CASCADE rule. According to the diagram, MATCHMAKER is a parent of GUESTS, GUESTS is a parent of TRAINERS, and TRAINERS is a parent of MATCHMAKER. If you delete Julio from the MATCHMAKER table, you delete Joan from the GUESTS table, causing Serena to be deleted from the TRAINERS table, and thus Todd and Ray are deleted from the MATCHMAKER table, and so on.



To avoid the widespread deletion of data, SQLBase enforces the following rule:

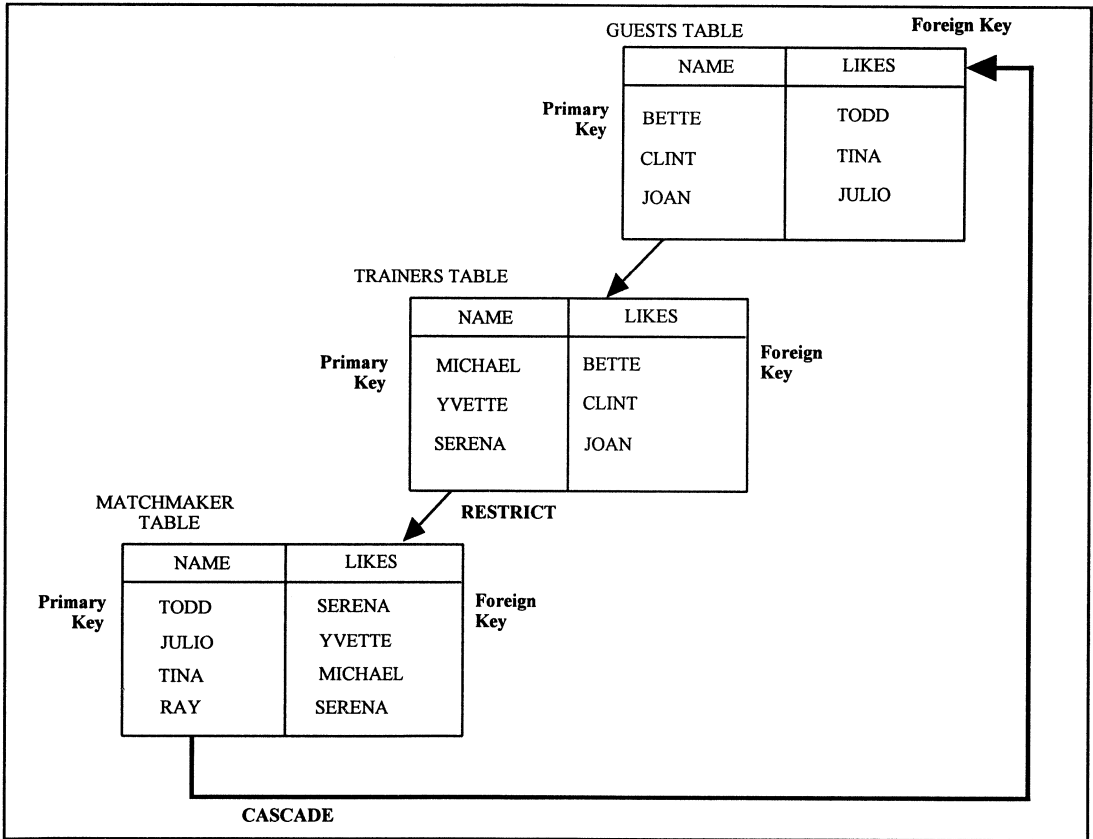
- In cycles of two or more tables, one relationship in the cycle must have a RESTRICT or SET NULL delete rule to break the cycle of cascaded deletions.

The table below shows how the RESTRICT rule affects the relationships of all three tables. Because of this rule, Julio's row is the only row you can delete from the MATCHMAKER and GUESTS tables. Every other row is the parent in some relationship, so you are restricted from deleting them.



It is important that you *not* specify the RESTRICT rule for *all* the relationships in a referential cycle, unless you want users to be barred from deleting *any* data.

The diagram below shows a correct implementation of a cycle, because the last two tables in the cycle do *not* have DELETE rules of CASCADE. The DELETE rules from the last table to the beginning table must not be CASCADE, and all the DELETE rules except the last one *must* be CASCADE.



Restrictions on Self-Referencing Tables

A self-referencing table is one that has foreign and primary keys with matching values within the same table. In the table below, the value of the foreign key GUESTNUM (guest number) matches the value of the primary key, ROOM. This is an example of a self-referencing table.

Room	Name	Rate
1	Annatto Rm	300
2	Carmine Rm	300
3	Cerise Rm	250
4	Cherry Rm	250
5	Maroon Rm	250
6	Puce Rm	250
7	Peach Rm	250
8	Marigold Suite	250
9	Gold Rm	250
10	Amber Rm	250

The following restrictions apply to self-referencing tables:

- The DELETE rule should be CASCADE. Otherwise, data cannot be deleted from the tables.
- An INSERT *subselect* is allowed if it *only* inserts one row.
- A DELETE WHERE CURRENT OF must not be used.

These restrictions guarantee that only one row will be involved when updating occurs, and the integrity of your data is ensured.

Delete-Connected Table Restrictions

Any table that is involved in a delete operation is delete-connected. For example, in cycles of dependent tables with a CASCADE delete rule, a deletion from one table directly affects the contents of other tables.

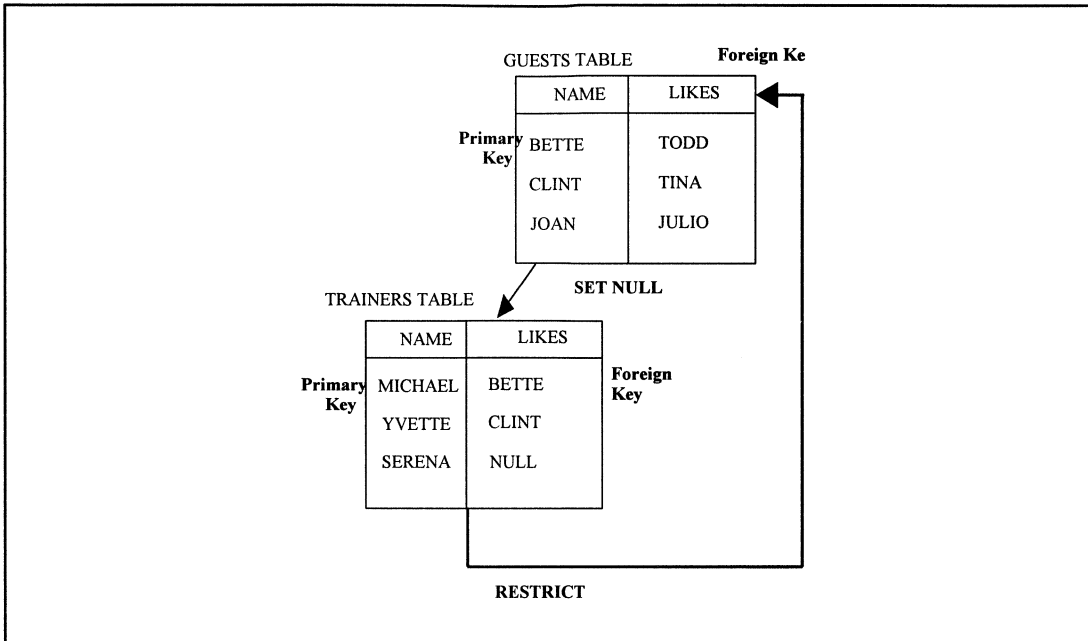
The following restrictions apply to delete-connected tables.

- If a DELETE operation involves a table that is referenced in a subquery, the last delete rule in the path to that table must be RESTRICT. A basic rule of SQL is that the result of an operation must *not* depend on the order in which rows of a table are accessed. That rule leads to the restriction that a subquery of a DELETE statement must not reference the same table that rows are deleted from. For example, without referential constraints, the following statement deletes rooms from the ROOMS table whose guests are not listed correctly in the GUESTS table.

```
SQL> DELETE FROM ROOMS
WHERE NOT ROOM = (SELECT NAME FROM GUESTS
WHERE GUESTNUM = GUESTS.GUESTNUM);
```

If referential constraints were defined for the sample tables, the statement would violate the rule that the last delete in the path to the tables must be RESTRICT. If the statement could be run, its results would depend on the order in which rows were accessed. SQLBase forces this statement to fail with an error message.

- If tables are delete-connected via two or more distinct referential paths, then the last delete rule should *not* be SET NULL. In the tables below, if a DELETE SET NULL was performed on the row JOAN in the GUESTS table, the value of JOAN in the TRAINERS table would be set to NULL. This violates the DELETE RESTRICT rule which prevents the deletion of a row from a parent table if the row has dependents.



Since different foreign keys can share the same column, disallowing SET NULL for the delete rule guarantees that applying the delete rule on any foreign key in GUESTS will not affect the values of other foreign keys. By only allowing CASCADE and RESTRICT, the dependent row will either be deleted (CASCADE) or remain the same (RESTRICT).

- If a referential cycle involves more than one table, any table in the cycle must *not* be delete-connected to itself. This restriction guarantees that the CASCADE rule will not cause widespread deletion of data.

Customizing SQLBase Error Messages

There are several error messages in SQLBase for referential integrity. This section shows how you can change the error messages to make them understandable to users.

The following are examples of error messages *before* editing error.sql:

- “EXE UFV - unmatched foreign key values”

This message is caused by an insert into a dependent table which failed because there was no parent row in the parent table.

- “EXE UFV - unmatched foreign key values”

This message is caused by an update into a dependent table which failed because there was no parent row in the parent table containing the new set of values.

- “EXE CDR - cannot delete row until all the dependent rows are deleted”

This message is caused by deleting rows from the parent table when there were dependent rows in the dependent table.

- “EXE CUR - cannot update row until all the dependent rows are deleted”

This message is caused by updating rows in a parent table when there were dependent rows in the dependent table.

Editing the Error Messages

To change the error messages for referential integrity, you can edit the *error.sql* file and use ALTER TABLE statements.

The following sections use an example that creates two tables, MANAGER and MEMBER, to show how to create user-specific error messages for primary keys.

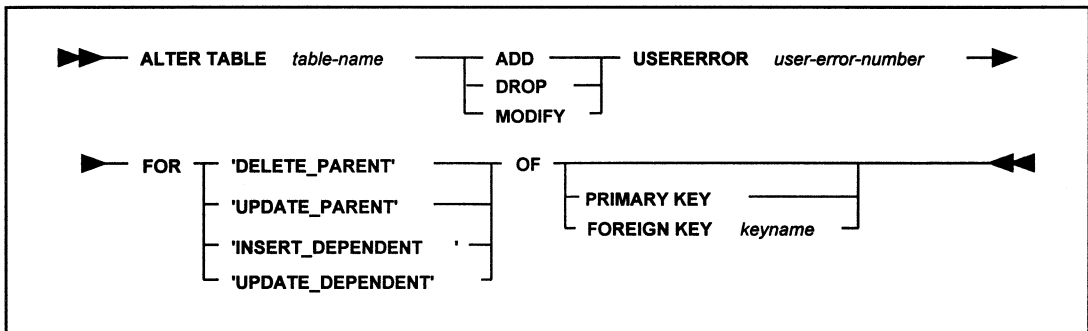
```
SQL> CREATE TABLE manager (name char(20) not null,
PRIMARY KEY (name));

CREATE UNIQUE INDEX manager_idx ON manager (name);

INSERT INTO manager VALUES ('Chris')
CREATE TABLE member (name char(20), manager char (20),
FOREIGN KEY mgr_ref (manager) REFERENCES manager);

INSERT INTO member VALUES ('Carol', 'Chris');
```

The following is the syntax of the ALTER TABLE statement to add, drop, or modify user specified error messages for primary keys:



The *user_error_number* in the syntax diagram above is a user-specified error message in the modified *error.sql*. Note that you do not need to enter the *user_error_number* for a DROP command.

□ **Primary Key Error Messages**

If a user attempts to delete a manager from the MANAGER table, it would fail with the default error message.

```
SQL> DELETE FROM manager where name = 'Chris';  
Error : EXE CDR - cannot delete row until all the dependent  
rows are deleted
```

However, if you add the following line to the *error.sql* file:

```
20000 xxx xxx Manager cannot be reassigned unless he/she  
has no member working for him/her.
```

Then you can use the ALTER TABLE statement to add the new error message:

```
SQL> ALTER TABLE manager ADD USERERROR 20000 FOR  
'DELETE_PARENT' OF PRIMARY KEY;
```

If a user tries to delete a manager from the table, the new error message will appear:

```
SQL> DELETE FROM manager where name = 'Chris';  
Error: Manager cannot be reassigned unless he/she has no  
member working for him/her.
```


□ Foreign Key Error Messages

In the following example, if a user attempts to insert a new member into the table, it would fail with the default error message.

```
SQL> INSERT INTO member VALUES ('John', 'Mike');  
Error : EXE UFV unmatched foreign key values
```

However, if you add the following line to the *error.sql* file:

```
20001 xxx xxx Member has to have a manager being assigned.
```

Then you can use the ALTER TABLE statement to add the new error message:

```
SQL> ALTER TABLE member ADD USERERROR 20001 FOR  
'INSERT_DEPENDENT' OF FOREIGN KEY mgr_ref;
```

If a user tries to insert a new member into the table, the following error message will occur:

```
SQL> INSERT INTO member VALUES ('John', 'Mike');  
Error: Member has to have a manager being assigned.
```

